Question Paper Set-2

Note:-Attempt questions of all sections as directed

Part-1

Note:-All questions are compulsory. Each question carry 2 Marks

Objective Type[2X10]

1. Translator for low level programming language were termed as
   A. Assembler B. Compiler C. Linker D. Loader
2. Shell is the exclusive feature of
   A. UNIX B. DOS C. System Software D. Application Software
3. A program in execution is called
   A. Process B. Instruction C. Procedure D. Function
4. Interval between the time of submission and completion of the job is called
   A. Waiting time B. Turnaround time C. Throughput D. Response time
5. The scheduling in which CPU is allocated to the process with least CPU burst time is called
   A. Priority Scheduling B. Shortest Job First Scheduling C. Round Robin Scheduling D. Multilevel queue Scheduling
6. System calls are usually invoked by using
   A. Software interrupt B. Polling C. An indirect jump D. A privileged instruction
7. A null process has a process identifier
   A. -1 B. 0 C. 1 D. Null
8. CPU performance measured through
   A. Throughput B. MHz C. Flaps D. None
9. .... Scheduler selects the jobs from the pool of jobs and loading to the ready queue
   A. Long term B. Short term C. Medium term D. None
10. Which scheduling policy is most suitable for a time shared operating system
    A. SJF B. Round Robin C.FCFS D. Priority

Part-2[8X5]

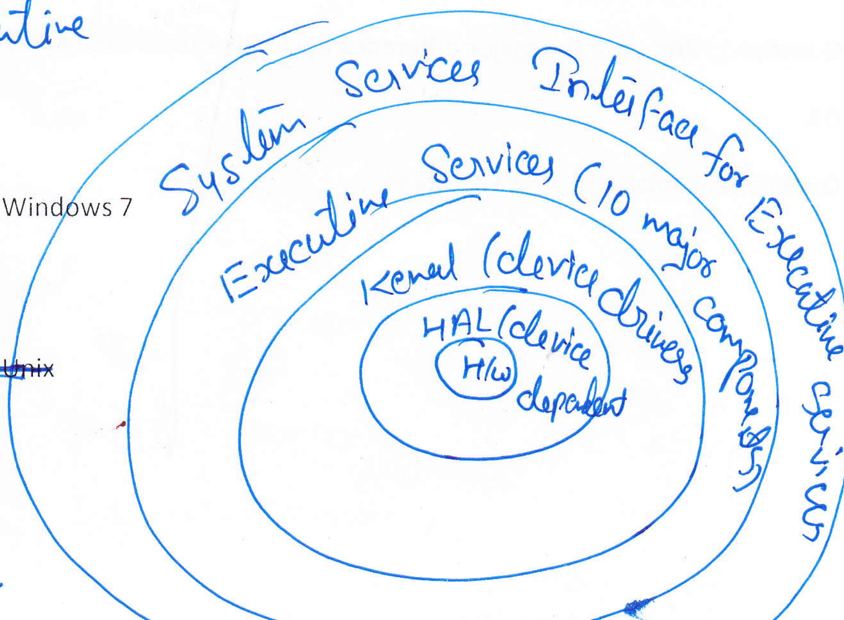
Solve any one from each unit

Unit-1

Question1:-Give introduction of Windows 7

OR

Question2:-Give introduction of Unix



UNIX

Unix is a multitasking, multi-user computer operating system originally developed in 1969 by a group of AT&T employees at Bell Labs, including Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy and Joe Ossanna.

The system qualifying the resemblance with either UNIX Version 7 or UNIX System V are called the Traditional UNIX systems. All others are called UNIX like Systems (LINUX, MINIX, BSD descendants etc.)

Company/Developer	AT&T Bell Labs
Programmed in	C
OS-family	UNIX
Available programming	C, C++
Kernal	Monolithic
User Interface	CLI & X-Window System

Peter Neumann Unics (Uniplexed Information and Computing ~~System~~ Service) Multics (Multiplexing information and Computer Services) gave rise to the name UNIX Multiple Users.

Kant Singh

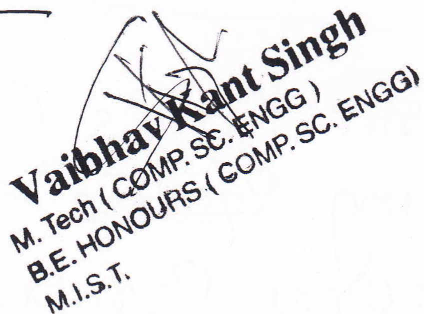
Components

Kernal
- conf — configuation of m/c dependent parts
- dev — device drivers
- sys — management routines [functions]
- h — header files

Development Environments
- cc – C compiler
- as – assembler
- ld – linker
- lib – libraries

Commands
- sh – shell commands
- utilities – system, user

Documentations
- man – manual pages for commands
- doc – longer documents



WINDOWS 7

DEVELOPER :- Microsoft
SOURCE MODEL :- Closed / shared source
Kernel type :- Hybrid
STATUS :- Running

Windows 7 is the latest release of Microsoft Windows, a series of operating systems produced by Microsoft for use on personal computers, including home and business desktops, laptops, netbooks, tablet-PC's and media center PCs. Windows 7 was released 3 years after its predecessor, Windows Vista. Windows 7 was intended more towards the incremental upgrade to the windows line, with the goal of being compatible with applications and hardware with which Windows Vista was already compatible.

Blackcomb → Vienna → Windows 7
Longhorn → Windows Vista

Vaibhav Kant Singh
M. Tech (COMP. SC. ENGG)
B.E. HONOURS (COMP. SC. ENGG)

New features

1) touch and hand writting recognition.
2) Support for virtual hard disks.
3) —— " —— multi-core processors.
4) Improved boot performance
5) Direct Access and Kernal improvements
6) Redesigned Calculator (Programmer & statistics
7) New Control Panel [Text Tuner, Display Calibration wizard]

9) Window 7 includes Internet Explorer 8
Windows Media Player 12.

10) Internet Spades, Internet Backgammon and Internet Checkers are retained (VISTA Removed)

11) ~~Taskbar~~ Changed taskbar. (Superbar)

12) 13 additional Sound Schemes etc.

13) Multiple Windows environments

14) Heterogeneous Graphics card support.

Physical Memory limits for WINDOWS 7 VERSIONS

VERSION	LIMIT in 32-bit Windows	Limit in 64 bit Windows
Windows 7 Ultimate	4GB	192GB
Windows 7 Enterprise	4GB	192GB
Windows 7 Professional	4GB	192GB
Windows 7 Home Premium	4GB	16 GB
Windows 7 Home Basic	4GB	8 GB
Windows 7 Starter	2GB	N/A

Vaibhav Kant Singh
M. Tech (COMP. SC. ENGG)
B.E. HONOURS (COMP. SC. ENGG)
M.I.S.T.

Unit-2

Question1:-Solve the Readers and Writers Problem using semaphores

Readers Writers Problem

A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**. Obviously, if two readers access the shared data simultaneously, no adverse affects will result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the *readers–writers problem*. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers–writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first readers–writers problem.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

The semaphores `mutex` and `wrt` are initialized to 1; `readcount` is initialized to 0. The semaphore `wrt` is common to both reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `readcount` is updated. The `readcount` variable keeps track of how many processes are currently reading the object. The semaphore `wrt` functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
}while (TRUE);
```

Figure The structure of a writer process.

```
do {
    wait(mutex);
    readcount++;
    if (readcount == 1)
       wait(wrt);
    signal(mutex);
       . . .
    // reading is performed
       . . .
    wait(mutex);
    readcount--;
    if (readcount == 0)
       signal(wrt);
    signal(mutex);
}while (TRUE);
```

Figure The structure of a reader process.

reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure ; the code for a reader process is shown in Figure . Note that, if a writer is in the critical section and *n* readers are waiting, then one reader is queued on `wrt`, and n − 1 readers are queued on `mutex`. Also observe that, when a writer executes `signal(wrt)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers–writers problem and its solutions has been generalized to provide **reader–writer** locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either *read* or *write* access. When a process only wishes to read shared data, it requests the reader–writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode; only one process may acquire the lock for writing as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which threads only write shared data.
- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual exclusion locks, and the overhead for setting up a reader–writer lock is compensated by the increased concurrency of allowing multiple readers.

OR

Question2:-Solve the Dining Philosophers Problem using semaphores

Answer2 :- **DiningPhilosopher Problem using Semaphore** Consider five philosophers who spend their lives thinking and eating.The philosophers share a circular table surrounded by five chairs each belonging to one philosophers. In the center of the table is a rice bowl, and the table is laid

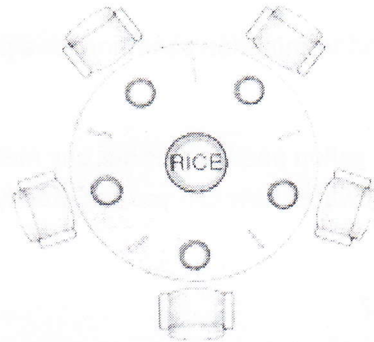


Figure The situation of the dining philosophers.

with five single chopsticks (Figure). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The *dining-philosophers problem* is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore; she releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of `chopstick` are initialized to 1. The structure of philosopher *i* is shown in Figure

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of `chopstick` will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

- Allow at most four philosophers to be sitting simultaneously at the table.

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
}while (TRUE);
```

Figure The structure of philosopher *i*.

- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Unit-3

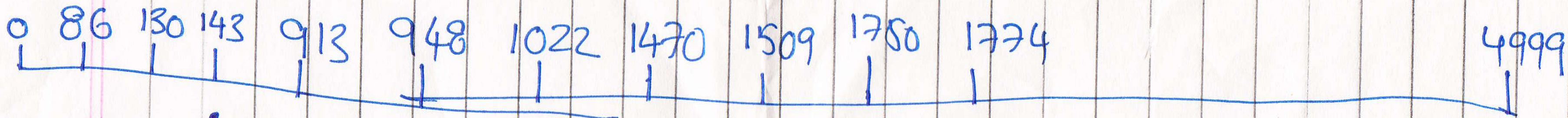
Question1:-Suppose that a disk drive has 5000 cylinders, numbered 0-4999.The drive is currently serving a request at cylinder 143 and the previous request was at cylinder 125.The queue of pending request in FIFO order is:

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Starting from the current head position, what is the total distance that the disk arm moves to satisfy all the pending results for each of the following disk scheduling algorithms:-

1. FCFS
2. SSTF

OR

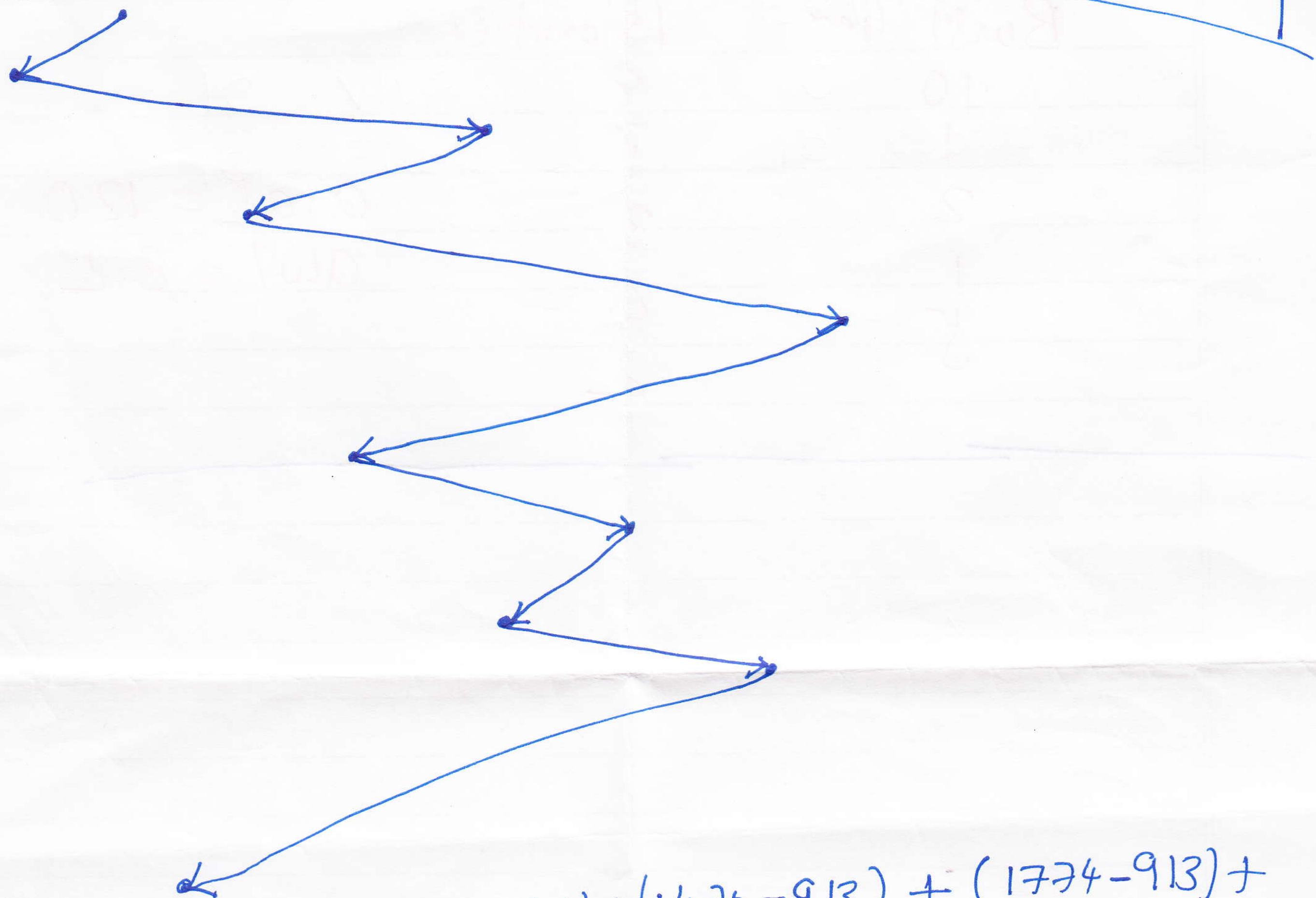


$$= (143 - 130) + (130 - 86) + (913 - 86) + (948 - 913) + (1022 - 948) + (1470 - 1022) + (1509 - 1470) + (1750 - 1509) + (1774 - 1780)$$

$$= 13 + 44 + 827 + 35 + 74 + 448 + 39 + 241 + 24$$

$$= 1745$$

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130



(143 - 86) + (1470 - 86) + (1470 - 913) + (1774 - 913) +
(1774 - 948) + (1509 - 948) + (1509 - 1022) + (1750 - 1022) +
(1750 - 130)

57 + 1384 + 557 + 861 + 826 + 561 + 487
+ 728 + 1620

4733 + 728 + 1620

7081

Explain Multilevel queue Scheduling and Multilevel feedback queue Scheduling? Give algorithm evaluation techniques? Explain Multiprocessor scheduling in brief?

BRICE
Page No.
Date: / /

MULTILEVEL QUEUE SCHEDULING

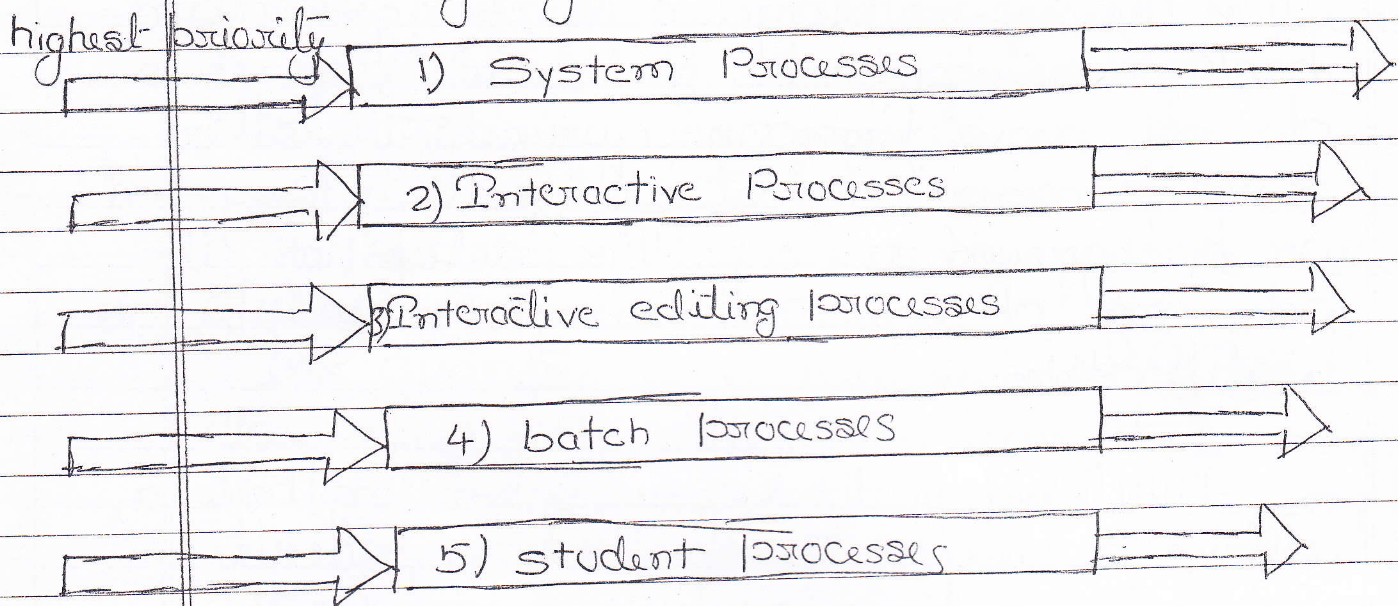
Another class of scheduling algorithm has been created for situations in which processes are easily classified into different groups. For example a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue-scheduling algorithm partitions the ready queue into several separate 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. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm. (Voidicidated)

In addition, there must be scheduling between the queues, which is commonly implemented as a fixed-priority preemptive scheduling. For example the foreground

the background queue

Let us look at an example of a multilevel queue scheduling algorithm with five queues



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

Another possibility is to time slice between the queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background 20 percent of the CPU to give

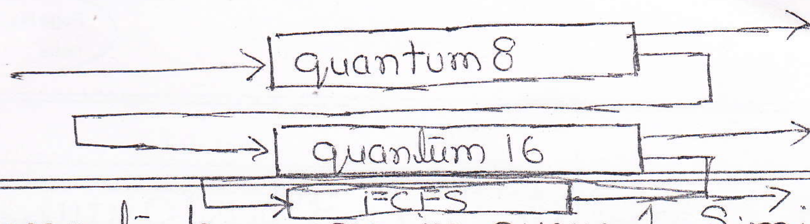
BRICE
Page No.
Date: / /

MULTILEVEL FEEDBACK QUEUE SCHEDULING

Normally, in a multilevel queue scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but is inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU burst characteristics. 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.

For example, consider a multilevel feedback queue schedular with three queues, numbered from 0 to 2. The scheduler first executes all processes in queue 0.



execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis, only when queues 0 and 1 are empty.

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

In general, a multilevel feedback queue schedular is defined by the following parameters :-

1 The number of queues,

2 The scheduling algorithm for each queue

3 The method used to determine when to upgrade a process to a higher priority queue

4 The method used to determine when to demote a process to a lower-priority queue

5 The method used to determine which queue a process will enter when that process needs service

BRICE
Page No.
Date: / /

MULTIPLE PROCESSOR SCHEDULING

If multiple CPUs are available, the scheduling problem is correspondingly more complex. Many possibilities have been tried and as we saw with single processor CPU scheduling, there is no one best solution. We concentrate on systems where the processors are identical (homogeneous) in terms of their functionality; Any available processor can then be used to run any processes in the queue.

Even within homogeneous multiprocessor, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes wishing to use that device must be scheduled to run on that processor. Processes wishing to use that device must be scheduled to run on that processor, otherwise the device would not be available.

If several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case. However, one processor could be idle, with an empty queue, while another processor was very busy. To prevent this situation, we use a common ready queue. All processes go into one queue and are scheduled onto any available processor.

In such a scheme, one of two scheduling approaches may be used. In one approach, each processor is self-scheduling. Each processor examines the common ready queue and selects a process to execute. If we have multiple processors trying to access and update a common data structures, each processor must be programmed very carefully. We must ensure that two processors do not choose the same process, and that processes are not lost from the queue. The other approach avoids this problem by appointing one processor as schedular for the other processors, thus creating a master-slave structure.

Some systems carry this structure one step further, by having all scheduling decisions I/O processing, and other system activities handled by one single processor - the master server. The other processors only execute user code. This asymmetric multiprocessing is far simpler than symmetric multiprocessing because only one processor accesses the system data structures, alleviating the need for data sharing.

BRICE

ALGORITHM EVALUATION

To select an algorithm, we must first define the relative importance of these measures. Our criteria may include several measures, such as

a) Maximize CPU utilization under the constraint that the maximum response time is 1 second

b) Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time.

There are a number of different evaluation methods given below:-

1. Deterministic Modeling
2. Queueing Models
3. Simulations
4. Implementation

Deterministic Modeling

One major class of evaluation methods is called analytic evaluation. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.

One type of analytic evaluation is deterministic modeling. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload.

Deterministic modeling is simple and fast. It gives exact numbers for input, and its answers apply to only those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we may be running the same programs over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm.

Queueing Models

The Computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time and so on. This area of study is called queueing-network analysis.

As an example, let n be the average queue length (excluding the process being serviced), let W be the average waiting time

in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). Then we expect that during the time W that a process waits, $\lambda \times W$ new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

$$n = \lambda \times W$$ Little's formula

SIMULATIONS

To get a more accurate evaluation of scheduling algorithms we can use simulations. Simulations involve programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

BRICE
Page No.
Date: / /

The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator, which is programmed to generate processes, CPU burst times, arrivals, departures and so on, according to probability distributions. The distributions may be defined mathematically or empirically. If the distribution is to be defined empirically, measurements of the actual system under study are taken. The results are used to define the actual distribution of events in the real system, and this distribution can then be used to drive the simulation

A distribution-driven simulation may be inaccurate, however, due to relationships between successive events in the real system. The frequency distribution indicates only how many of each event occur; it does not indicate anything about the order of their occurrence. To correct this problem, we can use trace tapes. We create a trace tape by monitoring the real system, recording the sequence of actual events. This sequence is then used to drive the simulation. Trace tapes provide an excellent way to compare two algorithms on exactly the same set of real inputs. This

Page No. ____
Date: / /

Not dictated

Simulations can be expensive; however, often requiring hours of computer time. A more detailed simulation provides more accurate results, but also requires more computer time. In addition, trace tapes can require large amounts of storage space. Finally, the design, coding and debugging of the simulator can be a major task.

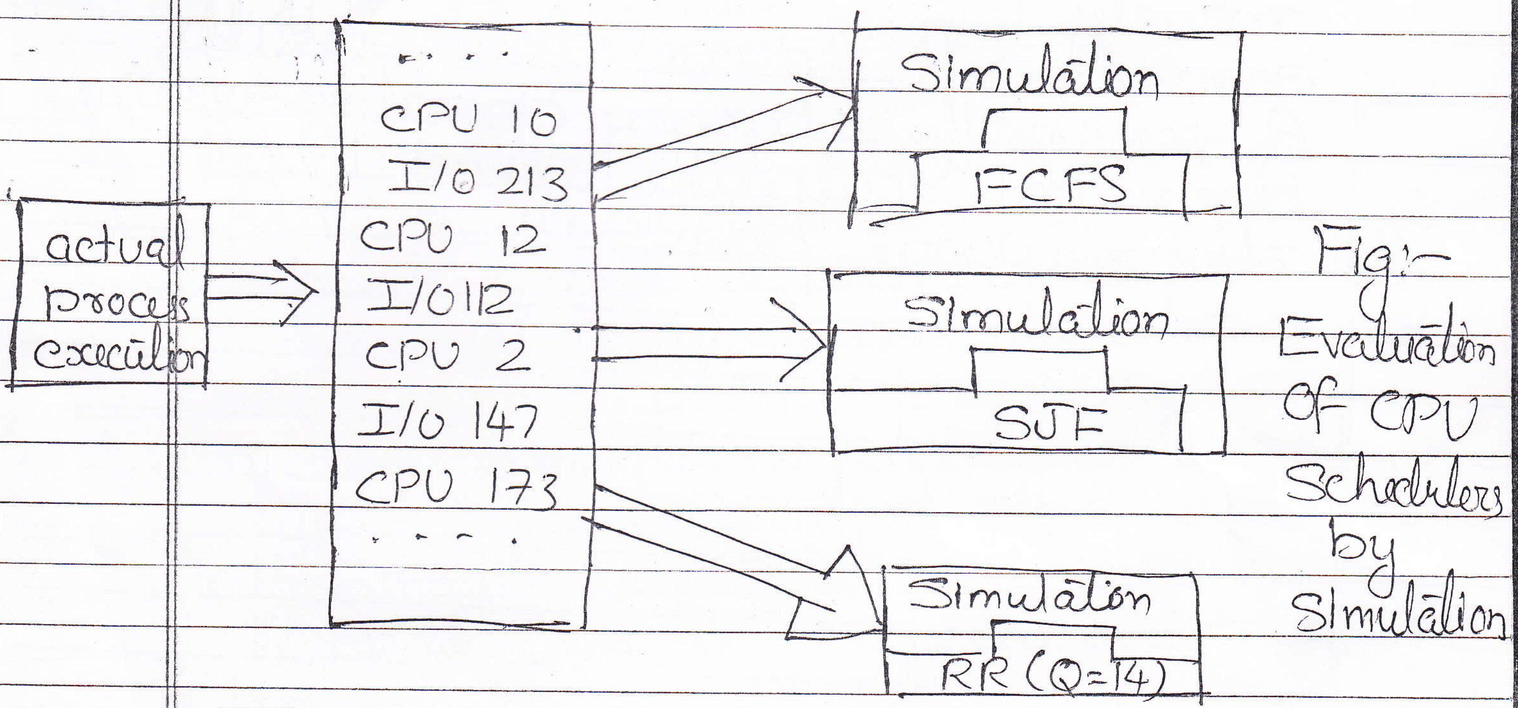


Fig:- Evaluation of CPU Schedulers by Simulation

IMPLEMENTATION

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, to put it in the operating system and to see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions

BRICE
Page No.
Date: / /

The major difficulty is the cost of this approach. The expense is incurred not only in coding the algorithm and modifying the operating system to support it as well as its required data structures, but also in the reaction to the users to a constantly changing operating system. Most users are not interested in building a better operating system; they merely want to get their processes executed and to use their results. A constantly changing operating system does not help the users to get their work done.

Unit-4

Question1:-Explain Process Termination and Resource Preemption in deadlock recovery?

Recovery From Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system *recover* from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim**. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.

2. **Rollback**. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

   Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. **Starvation**. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

   In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Explain Combined Approach to Deadlock handling

COMBINED APPROACH TO DEADLOCK HANDLING :-

Researchers have argued that none of the basic approaches for handling deadlocks (prevention, avoidance and detection) alone is appropriate for the entire spectrum of resource-allocation problems encountered in operating systems. One possibility is to combine the three basic approaches, allowing the use of the optimal approach for each class of resources in the system. The proposed method is based on the notion that resources can be partitioned into classes that are hierarchically ordered. A resource-ordering technique is applied to the classes. Within each class, the most appropriate technique for handling deadlocks can be used.

It is easy to show that a system that employs this strategy will not be subjected to deadlocks. Indeed, a deadlock cannot involve more than one class, since the resource-ordering technique is used. Within each class, one of the basic approaches is used. Consequently the system is not subject to deadlocks.

To illustrate this technique, we co a system that consists of the following four classes of resources :-

* Internal Resources

Resources used by the system, s as a process control block.

* Central Memory

Memory used by a user job.

* Job Resources

Assignable devices (such as a tape drive) and files.

* Swappable Space

Space for each user job on t backing store.

One mixed deadlock solution for this system orders the classes as shown, and uses the following approaches to each class:

* Internal Resources :-
Prevention through resource ordering can be used, since run-time choices between pending requests are necessary.

* Central Memory :-
Prevention through preemption can be used, since a job can always be swapped out, and the central memory can be preempted.

* Job Resources
Avoidance can be used, since the information needed about resource requirements can be obtained from the job-control cards.

* Swappable Space
Preallocation can be used, since the maximum storage requirements are usually known.

The above example shows how vari[ous] basic approaches can be mixed within the framework of resource ordering, to obtain an effective solution to the deadlock problem.

WORKING of COMBINED APPROACH

In this approach all the resource types on the basis of features that they exhibit are arranged into different classes. Say if R_1, R_2, R_3, R[...] $R_6, R_7, R_8, R_9, R_{10}$ resource types are p[...] in the system then these resource[s] on the basis of certain criteria could be arranged into suppose th[e] following set of classes

$C_1 = \{R_3, R_5, R_8\}$

$C_2 = \{R_2, R_4, R_9\}$

$C_3 = \{R_1, R_{10}\}$

$C_4 = \{R_6, R_7\}$

Now, the set of created classes ar[e] arranged in heirarchical order (ma[y] be on the basis of priority of class[es]) on the basis of certain crite[ria]

Class	Resources
C_3	R_1, R_{10}
C_1	R_3, R_5, R_8
C_4	R_6, R_7
C_2	R_2, R_4, R_9

← Suppose the classes C_1, C_2, C_3, C_4 are arranged in the ~~fashi~~ given fashion

Then two strategies are taken in account for allocation of resource by the System. The resource type of the individual classes could be managed by any of the core techniques i.e. prevention, avoidance and detection. But if a ~~Resource type~~ request comes for Resource types which are present in two different classes than the allocation is made on the basis of increasing order of enumeration. It means if a process requests for R_7 and currently it is holding R_8 than in that case the process has to first release R_8 than only it can made request for R_7.

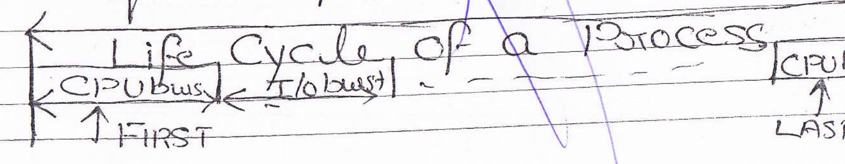
CPU Scheduling :-

Def :-
CPU scheduling is the basis of multipro[cut off] operating system. By switching the CPU among processes, the operating syste[cut off] can make the computer more produ[cut off]

Basic Concepts

CPU - I/O Burst Cycle :-

During the life cycle or time of the process the process toggles between two stages. The time interval wh[cut off] the CPU is holded by the process and the time period for which the process is engaged in some other I/O operation. The time which the process is engaged CPU operation and holding CPU is called CPU burst whereas th[cut off] time in which the I/O operation is accomplished is called I/O Burs[cut off] The execution of every process sta[cut off] from CPU burst and ends als[cut off] CPU burst. In between the tim[cut off] bursts ~~we a~~ every process is h[cut off] sequence of I/O and CPU burst[cut off]



Unit-5

Question1:-Explain Virtual Memory?

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files easily and to implement shared memory. In addition, it provides an efficient mechanism for process creation. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly.

Background

The memory-management algorithms outlined are necessary because of one basic requirement: The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Dynamic loading can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.

The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements. An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget are only rarely used.

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

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large *virtual* address space, simplifying the programming task.

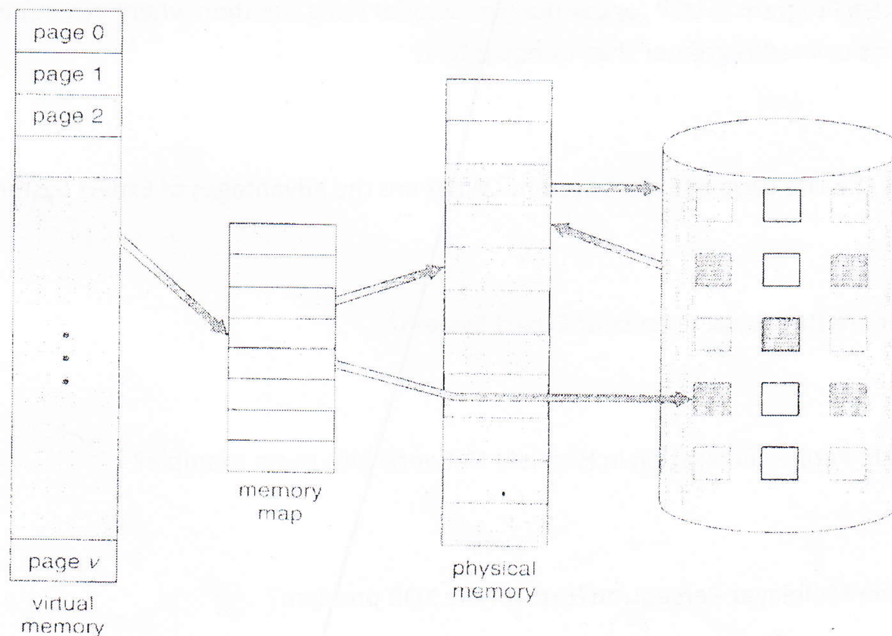


Figure Diagram showing virtual memory that is larger than physical memory.

- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and the user.

Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on the problem to be programmed.

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address—say, address 0—and exists in contiguous memory, as shown in Figure , though, that in fact physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory-management unit (MMU) to map logical pages to physical page frames in memory.

Note in Figure that we allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address

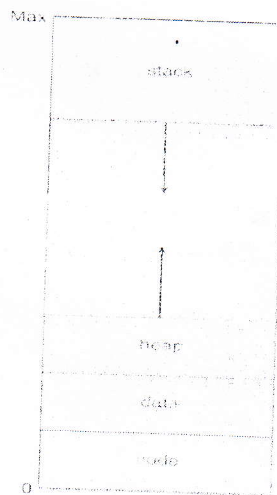


Figure Virtual address space.

space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **sparse** address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by two or more processes through page sharing This leads to the following benefits:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes Typically, a library is mapped read-only into the space of each process that is linked with it.
- Similarly, virtual memory enables processes to share memory. Recall that two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Figure
- Virtual memory can allow pages to be shared during process creation with the `fork()` system call, thus speeding up process creation.

We will further explore these—and other—benefits of virtual memory later in this chapter. First, we begin with a discussion of implementing virtual memory through demand paging.

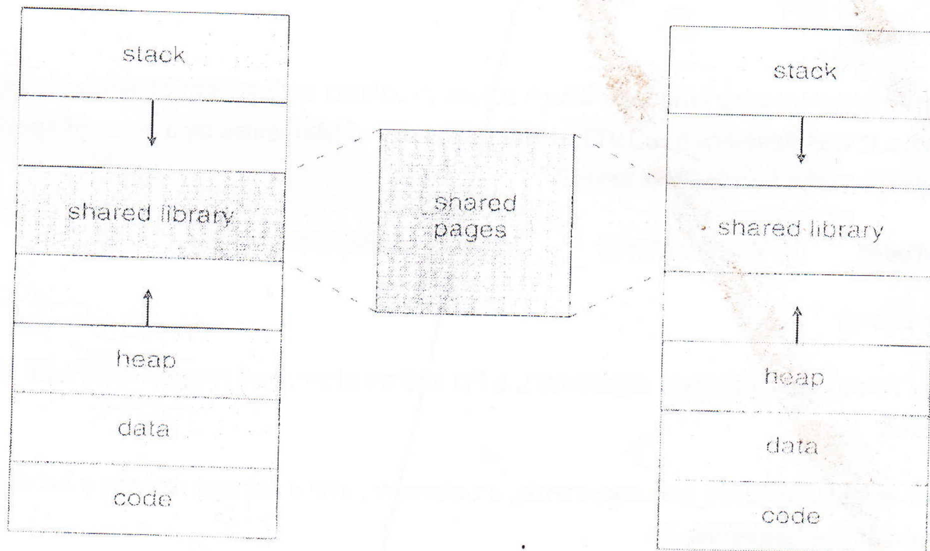


Figure Shared library using virtual memory.

OR

Question2:-Describe Demand Paging?

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially *need* the entire program in memory. Consider a program that starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for *all* options, regardless of whether an option is ultimately selected by the user or not. An alternative strategy is to initially load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.

A demand-paging system is similar to a paging system with swapping (Figure) where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term *swapper* is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use *pager,* rather than *swapper,* in connection with demand paging.

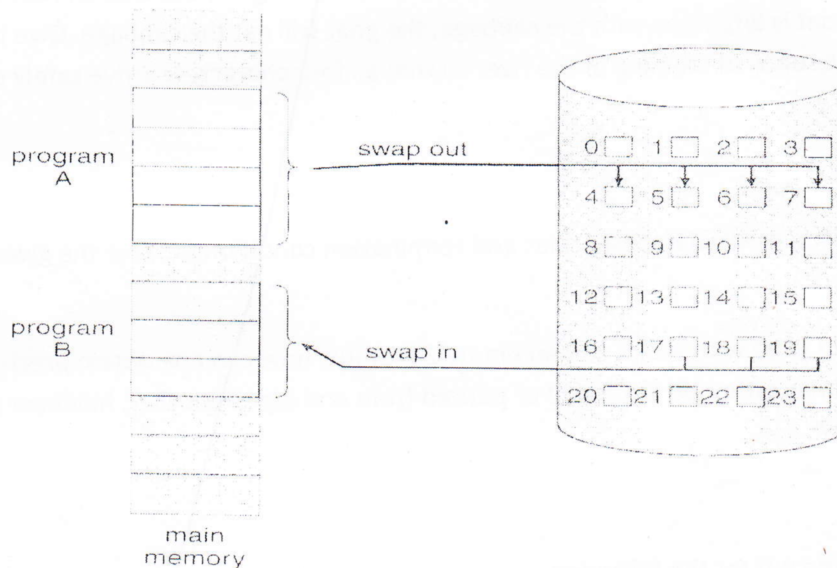


Figure Transfer of a paged memory to contiguous disk space.

Basic Concepts

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid–invalid bit scheme described can be used for this purpose. This time, however, when this bit is set to "valid," the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is depicted in Figure

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are **memory resident**, execution proceeds normally.

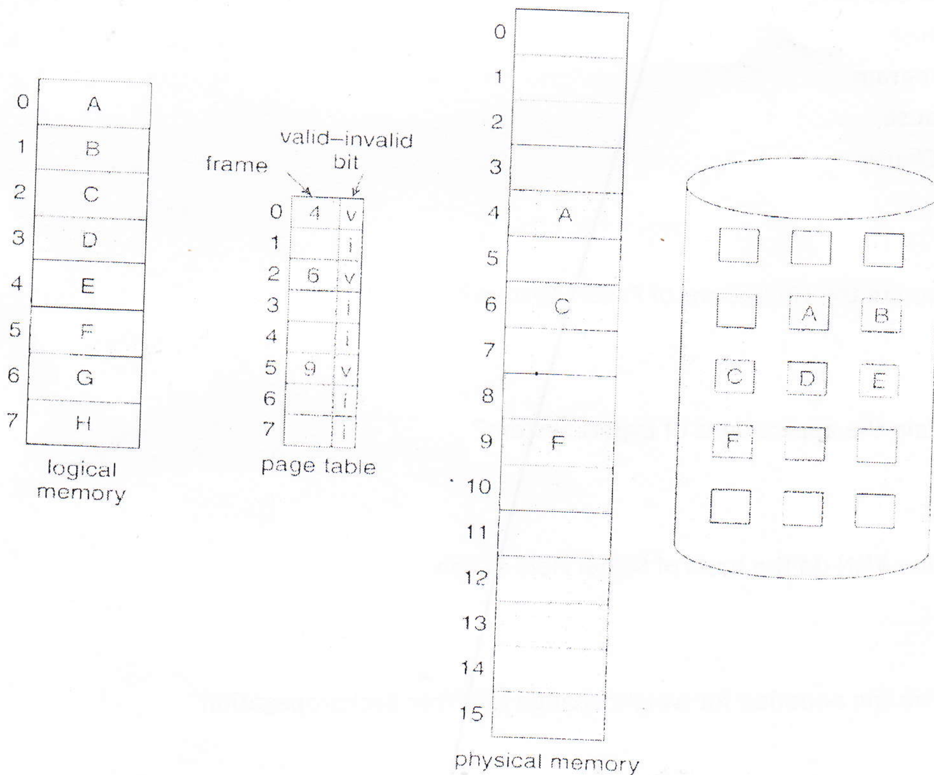


Figure Page table when some pages are not in main memory.

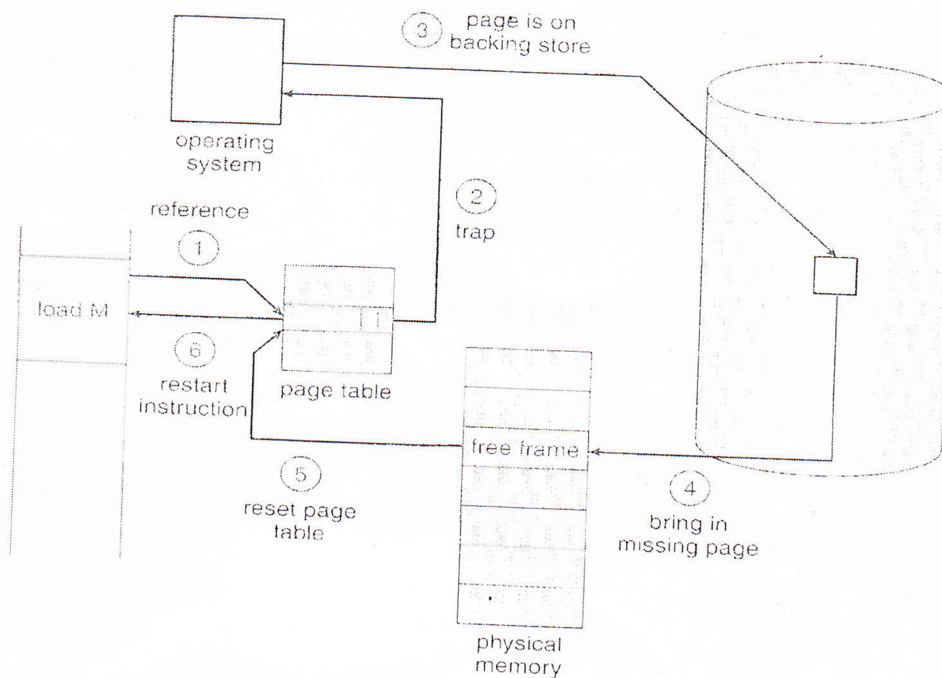


Figure Steps in handling a page fault.

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page-fault trap**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with *no* pages in memory. When the operating system sets the instruction pointer to the first

instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging**: Never bring a page into memory until it is required.

Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely. Programs tend to have **locality of reference**, , which results in reasonable performance from demand paging.

The hardware to support demand paging is the same as the hardware for paging and swapping:

- **Page table**. This table has the ability to mark an entry invalid through a valid–invalid bit or special value of protection bits.
- **Secondary memory**. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space**.

A crucial requirement for demand paging is the need to be able to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in *exactly* the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and

then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs.

The major difficulty arises when one instruction may modify several different locations. For example, consider the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another (possibly overlapping) location. If either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done. In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction.

This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place; we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

This is by no means the only architectural problem resulting from adding paging to an existing architecture to allow demand paging, but it illustrates some of the difficulties involved. Paging is added between the CPU and the memory in a computer system. It should be entirely transparent to the user process. Thus, people often assume that paging can be added to any system. Although this assumption is true for a non-demand-paging environment, where a page fault represents a fatal error, it is not true where a page fault means only that an additional page must be brought into memory and the process restarted.

Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the **effective access time** for a demand-paged memory. For most computer systems, the memory-access time, denoted *ma*, ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let p be the probability of a page fault ($0 \le p \le 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults. The **effective access time** is then

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}.$$

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.

3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
   a. Wait in a queue for this device until the read request is serviced.
   b. Wait for the device seek and/or latency time.
   c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization but requires additional time to resume the page-fault service routine when the I/O transfer is complete.

In any case, we are faced with three major components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page-switch time, however, will probably be close to 8 milliseconds. A typical hard disk has an average latency of 3 milliseconds, a seek of 5 milliseconds, and a transfer time of 0.05 milliseconds. Thus, the total paging time is about 8 milliseconds, including hardware and software time. Remember also that we are looking at only the device-service time. If a queue of processes is waiting for the device (other processes that have caused page faults), we have to add device-queueing time as we wait for the paging device to be free to service our request, increasing even more the time to swap.

If we take an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, then the effective access time in nanoseconds is