CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assesment Test – III

Sub: Operating Systems Code: 16MCA24

Date: 26.05.2017 Duration: 90 mins Max Marks: 50 Sem: II Branch: MCA

Answer Any FIVE FULL Questions

		Marks	OBE CO	OBE RBT
1	What do you mean by deadlock avoidance? Explain the banker's algorithm for deadlock avoidance.	10	CO3	L2
2(a)	Explain fragmentation and its types with neat diagram.	5	CO3	L1
(b)	Brief first fit, best fit, worst fit concepts with examples.	5	CO3	L2
3	Discuss different page table architectures and explain each with a neat diagram.	10	CO3	L2
4 (a)	Under what circumstances do page faults occur? Describe the action taken by the operating system when a page fault occurs.	7	CO3	L2
(b)	What do you mean by thrashing?	3	CO3	L1
5 (a)	Explain the various file allocation methods in detail.	6	CO4	L2
(b)	Define i) Rotational Latency ii) Seek Time	4	CO4	L1
6(a)	Define disk scheduling.	2	CO4	L2
(b)	What are the disk scheduling methods available? Explain any four in detail with examples.	8	CO4	L2
7	Explain the components of LINUX operating system.	10	CO5	L2
8	Write short notes on the following:	5	CO5	L2
(a)	fork() and exec() process model.			
(b)	Free space management.	5	CO5	L2

Internal Assesment Test – III

Sub : Operating Systems **Code: 16MCA24**

1. What do you mean by deadlock avoidance? Explain the banker's algorithm for deadlock avoidance

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Banker's Algorithm

The resource-allocation graph algorithm is not applicable when there are multiple instances for each resource. The banker's algorithm addresses this situation, but it is less efficient. The name was chosen 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 resource type that it may need. This number may not exceed the total number 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, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Data structure for Banker's algorithms is as below –
Let n be the number of processes in the system and m be the number of resource types.
 Available*:* Vector of length *m* indicating number of available resources. If Available[*j*] = *k*, there are *k* instances of resource type *Rj* available.
 Max*:* An *n x m* matrix defines the maximum demand of each process. If *Max* [*i,j*] = *k*, then process *Pi* may request at most *k* instances of resource type *Rj*.
 Allocation*:* An *n x m* matrix defines the number of resources currently allocated to each process. If Allocation[*i, j*] = *k* then *Pi* is currently allocated *k* instances of *Rj*.
 Need*:* An *n x m* matrix indicates remaining resource need of each process. If *Need*[*i,j*] = *k*, then *Pi* may need *k* more instances of *Rj* to complete its task. Note that, *Need* [*i,j]* = *Max*[*i,j*] – *Allocation* [*i,j*].

The Banker's algorithm has two parts:
1. **Safety Algorithm:** It is for finding out whether a system is in safe state or not. The steps are as given below –
1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
Work = *Available*

Finish [*i*] = *false* for *i* = 1, 2, 3, …, *n.*
2. Find an *i* such that both:
(a) *Finish* [*i*] = *false*
(b) *Needi* □ *Work*
If no such *i* exists, 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.

2. **Resource – Request Algorithm:** Let *Requesti* be the request vector for process *Pi*. If *Requesti* [*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj*.
1. If *Requesti* □ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If *Requesti* □ *Available*, go to step 3. Otherwise *Pi* must wait, since resources are not available.
3. Pretend to allocate requested resources to *Pi* by modifying the state as follows:
Available = Available - Requesti;
Allocationi = Allocationi + Requesti;
Needi = Needi – Requesti;
If the resulting resource allocation is safe, then the transaction is complete and the process Pi is allocated its resources. If the new state is unsafe, then Pi must wait for *Requesti* , and the old resource-allocation state is restored

Example for Banker's algorithm:
Consider 5 processes *P*0 through *P*4 and 3 resources *A* (10 instances), *B* (5 instances), and *C* (7 instances). Snapshot at time *T*0 the snapshot of the system is as given in Table 5.1.

Table 5.1 Snapshot of the system at time To

Process	Allocation (A B C)	Max (A B C)	Available (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)	

The matrix *Need* = *Max* – *Allocation*. It is given by the Table 5.2.

Table 5.2 Need Matrix

Process	Need (A B C)
P0	(7 4 3)
P1	(1 2 2)
P2	(6 0 0)
P3	(0 1 1)
P4	(4 3 1)

Table 5.3 New State

Process	Allocation (A B C)	Need (A B C)	Available (A B C)
P0	(0 1 0)	(7 4 3)	(2 3 0)
P1	(3 0 2)	(0 2 0)	
P2	(3 0 2)	(6 0 0)	
P3	(2 1 1)	(0 1 1)	
P4	(0 0 2)	(4 3 1)	

We can apply Safety algorithm to check whether the system is safe. We can find that the sequence <P1, P3, P4, P2, P0> is one of the safety sequences. Suppose, now the process P1 makes a request (1, 0, 2). To check whether this

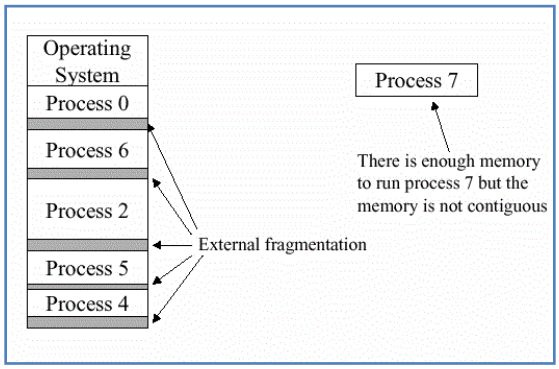
request can be immediately granted, we can apply Resource-Request algorithm. If we assume that this request is fulfilled, the new state would be as shown in Table 5.3. Now, by checking using safety algorithm, we see that the sequence <P1, P3, P4, P0, P2> is in safe state. Hence, this request can be granted.

2.a) Explain fragmentation and its types with neat diagram

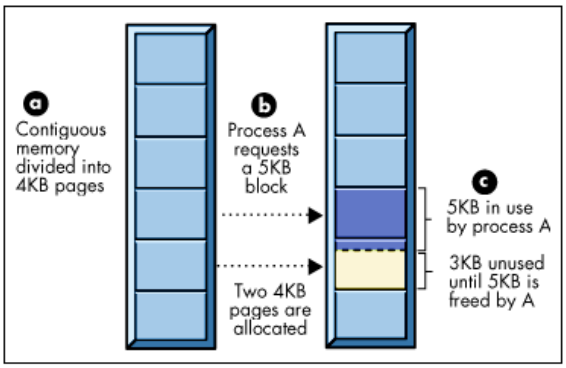
External Fragmentation – when total memory space exists to satisfy a request, but the available spaces are not contiguous.

- Storage is fragmented into a large number of small holes caused by first-fit and best-fit strategies for memory allocation plus variable-partition scheme

Internal Fragmentation – when allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used



External Fragmentation Example



Internal Fragmentation Example

2.b) Brief first fit, best fit, worst fit concepts with examples

First Fit

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition.

Advantage : Fastest algorithm because it searches as little as possible.

Disadvantage : The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

Best Fit

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.

Advantage : Memory utilization is much better than first fit as it searches the smallest free partition first available.

Disadvantage : It is slower and may even tend to fill up memory with tiny useless holes.

Worst fit

In worst fit approach is to locate largest available free portion so that the portion left will be big enough to be useful. It is the reverse of best fit.

Advantage : Reduces the rate of production of small gaps.

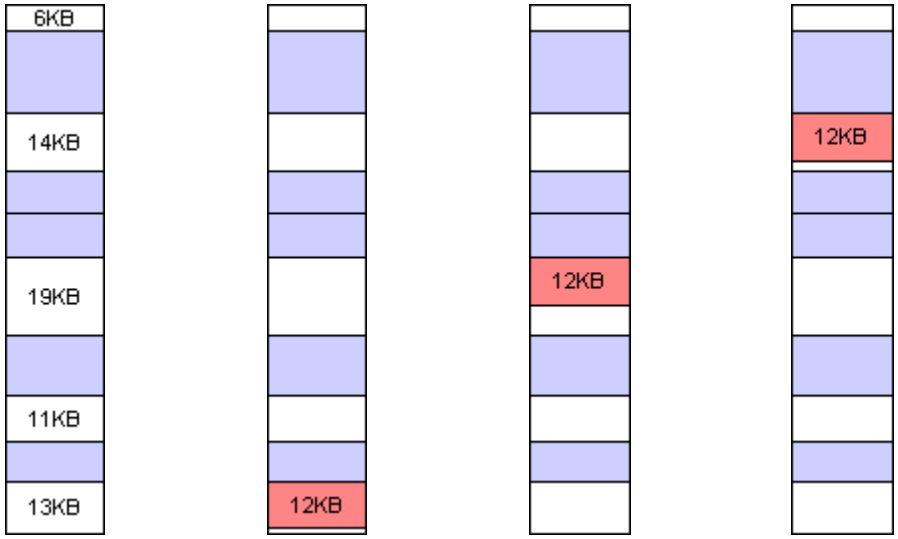
Disadvantage : If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

Example

- **Best fit:** The allocator places a process in the smallest block of unallocated memory in which it will fit. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The best-fit strategy will allocate 12KB of the 13KB block to the process.

- **Worst fit:** The memory manager places a process in the largest block of unallocated memory available. The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that, compared to best fit, another process can use the remaining space. Using the same example as above, worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.

- **First fit:** There may be many holes in the memory, so the operating system, to reduce the amount of time it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request. Using the same example as above, first fit will allocate 12KB of the 14KB block to the process.



3.a) Discuss different page table architectures and explain each with a neat diagram

There are three common techniques for structuring a page table. They are:

Hierarchical Paging - Break up the logical address space into multiple page tables. A simple technique is a two-level page table.

A logical address (on 32-bit machine with 1K page size) is divided into:

- a page number consisting of 22 bits
- a page offset consisting of 10 bits

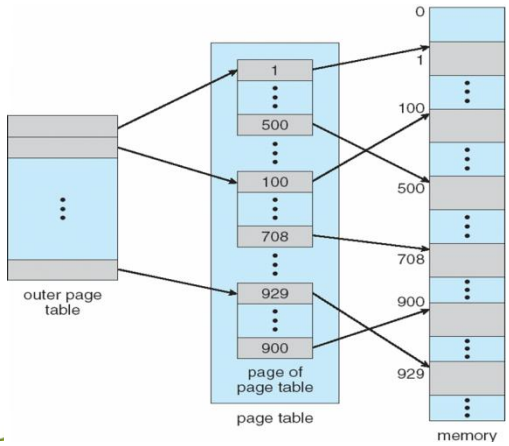
Since the page table is paged, the page number is further divided into:

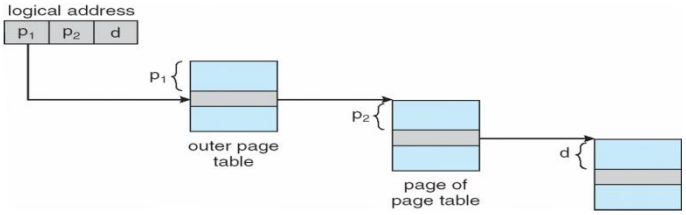
- a 12-bit page number
- a 10-bit page offset

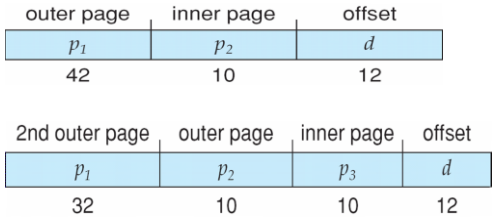
Thus, a logical address is as follows:

page number		page offset
p_i	p_2	d
12	10	10

where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table







Hashed Page Table –

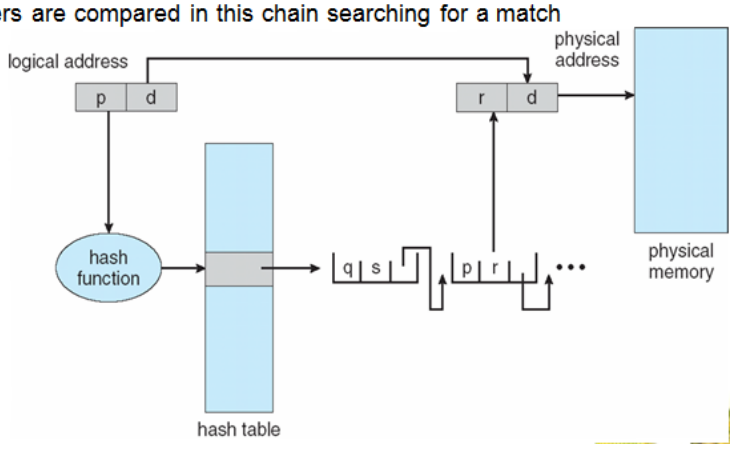
Common in address spaces > 32 bits

The virtual page number is hashed into a page table

- This page table contains a chain of elements hashing to the same location

Virtual page numbers are compared in this chain searching for a match

- If a match is found, the corresponding physical frame is extracted



Inverted Page Table –

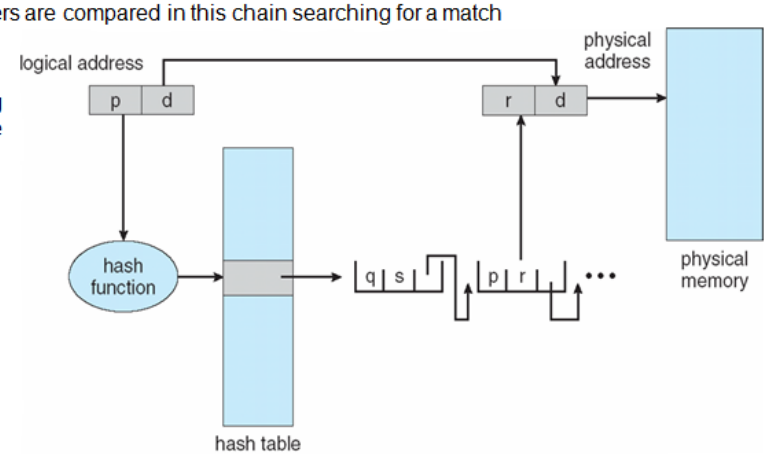
Common in address spaces > 32 bits

The virtual page number is hashed into a page table

- This page table contains a chain of elements hashing to the same location

Virtual page numbers are compared in this chain searching for a match

- If a match is found, the corresponding physical frame is extracted



4.a) Under what circumstances do page faults occur? Describe the action taken by the operating system when a page fault occurs.

In Demand pagubg , we need to distinguish the pages which are in the memory and pages which are there on the disk. For this purpose, the valid – invalid bit is used. When this bit is set to *valid*, it indicates the page is in the memory. Whereas, the value of bit as *invalid* indicates page is on the disk.

If the process tries to access a page which is not in the memory (means, it is on the disk), ***page fault*** occurs. The paging hardware notices the *invalid* bit in the page table and cause a trap to the OS. This trap is the result of the failure of OS to bring the desired page into memory. This error has to be corrected. The procedure for handling this page fault is as shown in Figure 6.16. The steps are explained below:

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 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 into memory, it is brought now.
3. We find a free frame.
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 illegal address trap. The process can now access the page as though it had always been in memory.

It is important to realize that, because we save the state (registers, condition code, instruction counter etc.) of the interrupted process when the page fault occurs, we can restart the process in exactly the same place and state In an extreme situation, a process may starts executing with no page in the memory. So, each time an instruction has to be executed, page fault occurs and the required page needs to be brought into the memory. This situation is called as ***pure demand paging***.

That is, no page is brought into the memory until it is required.

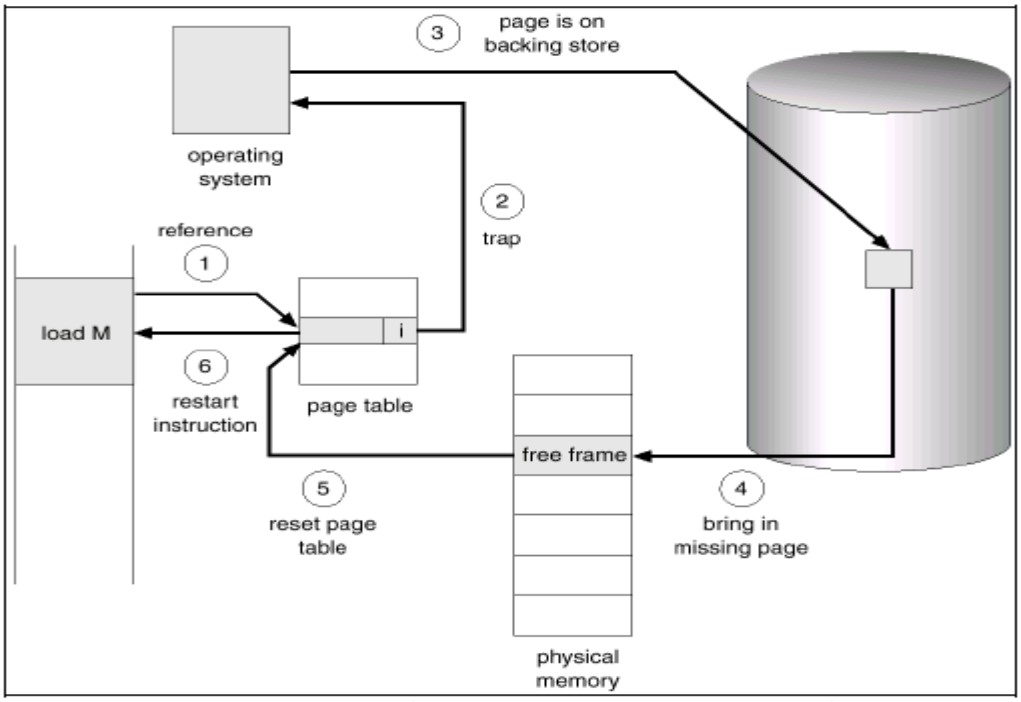


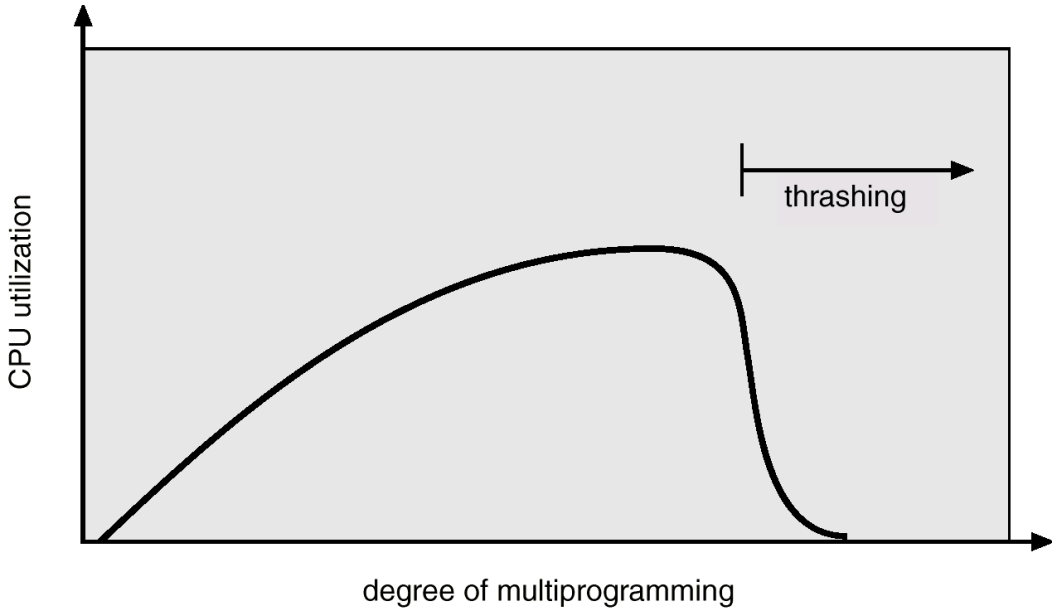
Figure 6.16 Steps in handling page fault

4.b) What do you mean by thrashing?

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend the execution of that process. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling. Whenever any process does not have enough frames, it will page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again. The process continues to fault, replacing pages for which it then faults and brings back in right away. This high paging activity is called ***thrashing***.

A process is thrashing if it is spending more time paging than executing. Thrashing affects the performance of CPU as explained below:

If the CPU utilization is low, we normally increase the degree of multiprogramming by adding a new process to the system. A global page-replacement algorithm is used, and hence, the new process replaces the frames belonging to other processes as well. As the degree of multiprogramming increases, obviously there will be more page faults leading to thrashing. When every process starts waiting for paging rather than executing, the CPU utilization decreases. This problem is shown in Figure 6.18. The effects of thrashing can be limited by using local replacement algorithm.

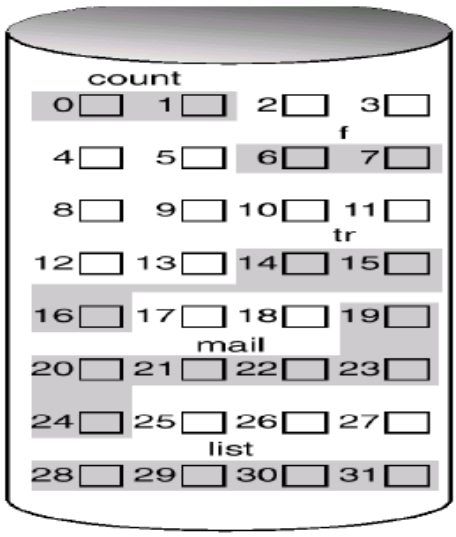


5. a) Explain the various file allocation methods in detail.

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are: contiguous, linked, and indexed.

Contiguous Allocation

In contiguous allocation, files are assigned to contiguous areas of secondary storage. A
user specifies in advance the size of the area needed to hold a file to be created. If the desired amount of contiguous space is not available, the file cannot be created. A contiguous allocation of disk space is shown in Figure 7.11.



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Figure 7.11 Contiguous allocation of disk space

One advantage of contiguous allocation is that all successive records of a file are normally physically adjacent to each other. This increases the accessing speed of records. It means that if records are scattered through the disk it is accessing will be slower. For sequential access the file system remembers the disk address of the last block and when necessary reads the next block. For direct access to block B of a file that starts at location L, we can immediately access block L+B. Thus contiguous allocation supports both sequential and direct accessing. The disadvantage of contiguous allocation algorithm is, it suffers from external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file as shown in Figure 7.12.

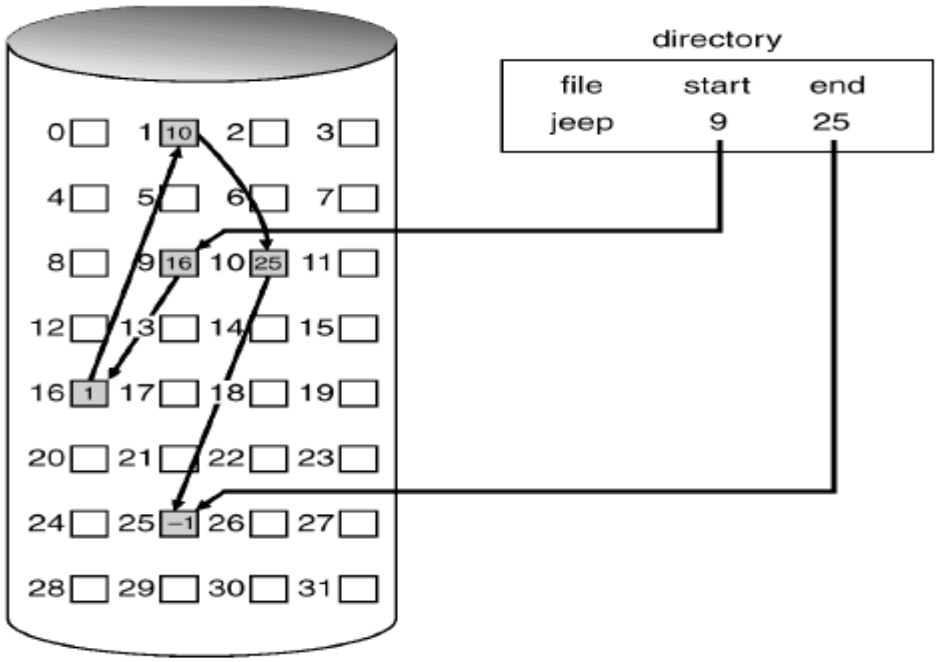


Figure 7.12 Linked Allocation of disk space

Linked allocation solves the problem of external fragmentation, which was present in contiguous allocation. But, still it has a disadvantage: Though it can be effectively used for sequential-access files, to find ith file, we need to start from the first location. That is, random-access is not possible.

Indexed Allocation

This method allows direct access of files and hence solves the problem faced in linked allocation. Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block as shown in Figure 7.13.

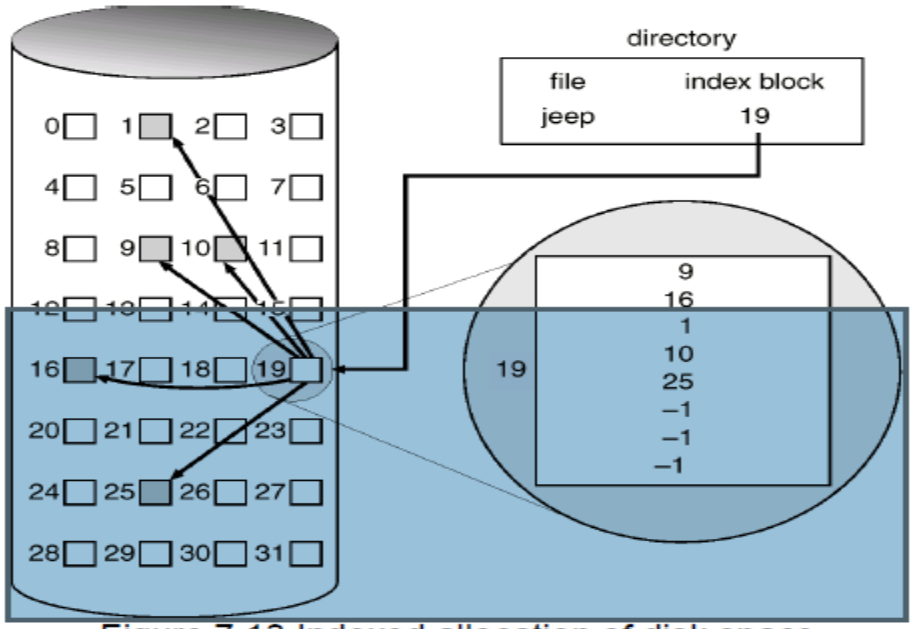


Figure 7.13 Indexed allocation of disk space

Figure 7.13 Indexed allocation of disk space

5.b) Define i) Rotational Latency ii) Seek Time

The **rotational latency** is the additional time waiting for the disk to rotate the desired sector to the disk head. The **rotational latency** is the additional time waiting for the disk to rotate the desired sector to the disk head.

The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector

6.a)Define disk scheduling

I/O request issues a system call to the OS. ‰If desired disk drive or controller is available, request is served immediately. ‰If busy, new request for service will be placed in the queue of pending requests. When one request is completed, the OS has to choose which pending request to service next

6.b)What are the disk scheduling methods available? Explain any four in detail with examples

FCFS Scheduling

„ Simplest, perform operations in order requested

„ no reordering of work queue

„ no starvation: every request is serviced

 Doesn't provide fastest service

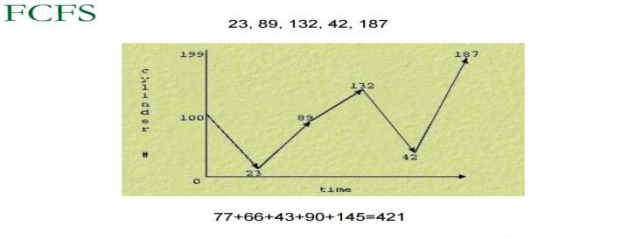
„ Ex: a disk queue with requests for I/O to blocks on cylinders

23, 89, 132, 42, 187

With disk head initially at 100

.

FCFS



If the requests for cylinders 23 and 42 could be serviced together, total head movement could be decreased substantially.

SSTF Scheduling

„ Like SJF, select the disk I/O request that requires the least movement of the disk arm from its current position, regardless of direction

„ reduces total seek time compared to FCFS.

„ Disadvantages

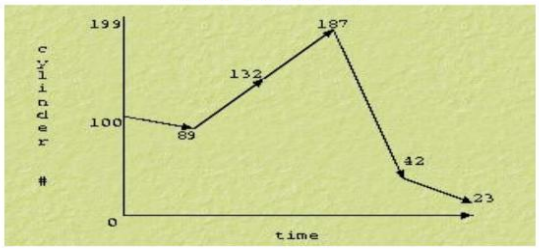
‰starvation is possible; stay in one area of the disk if very busy

‰switching directions slows things down

‰Not the most optimal

SSTF

23, 89, 132, 42, 187



11+43+55+145+19=273

SCAN

go from the outside to the inside servicing requests and then back from the outside to the inside servicing requests.

„ Sometimes called the elevator algorithm.

„ Reduces variance compared to SSTF.

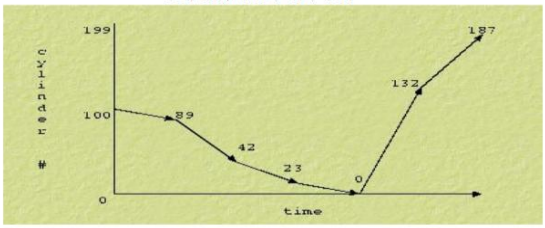
„ If a request arrives in the queue

‰just in front of the head

‰Just behind

SCAN

23, 89, 132, 42, 187



11+47+19+23+132+55=287

C-SCAN

„ Circular SCAN

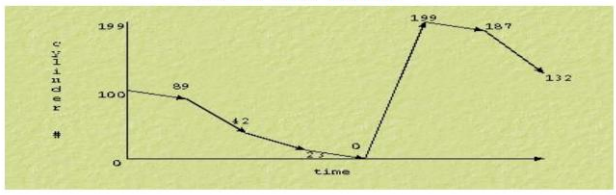
„ moves inwards servicing requests until it reaches the innermost cylinder; then jumps to the outside cylinder of the disk without servicing any requests.

„ Why C-SCAN?

‰Few requests are in front of the head, since these cylinders have recently been serviced. Hence provides a more uniform wait time.

C-SCAN

23, 89, 132, 42, 187



11+47+19+23+199+12+55=366

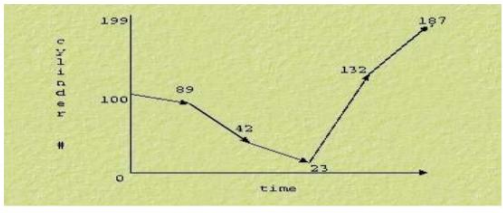
Head movement can be reduced if the request for cylinder 187 is serviced directly after request at 23 without going to the disk 0

LOOK

„ like SCAN but stops moving inwards (or outwards) when no more requests in that direction exist

LOOK

23, 89, 132, 42, 187



11+47+19+109+55=241

Compared to SCAN, LOOK saves going from 23 to 0 and then back.
Most efficient for this sequence of requests

7. Explain the components of LINUX operating system.

Ans: The Linux system is composed of three main bodies of code, in line with most traditional UNIX implementations:

1. Kernel. The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.

2. System libraries. The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not needthe full privileges of kernel code.

3. System utilities. The system utilities are programs that perform individual, specialized management tasks. Some system utilities may be invoked just once to initialize and configure some aspect of the system; others—
known as *daemons* in UNIX terminology—may run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

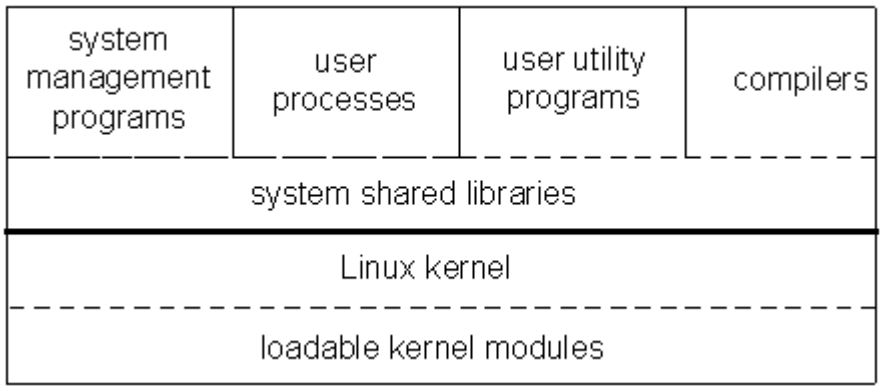


Figure 21.1 illustrates the various components that make up a full Linux system. The most important distinction here is between the kernel and everything else. All the kernel code executes in the processor's privileged mode with full access to all the physical resources of the computer. Linux refers to this privileged mode as **kernel mode.**

KERNEL MODULES

Kernel modules are the sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel. A kernel module may typically implement a device driver, a file system, or a networking protocol. The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL (General Public Library). Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in. Three components to Linux module support are discussed here.

 Module Management: It supports to load modules into memory and letting them communicate to the rest of the kernel. Module loading is split into two separate sections: o Managing sections of module code in kernel memory
o Handling symbols that modules are allowed to reference The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

 Driver Registration: Allows modules to tell the rest of the kernel that a new driver has become available. The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time. Registration tables include the following items:

o Device drivers
o File systems
o Network protocols
o Binary format
 Conflict Resolution: A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver. The conflict resolution module aims to:
o Prevent modules from clashing over access to hardware resources
o Prevent *autoprobes* from interfering with existing device drivers
o Resolve conflicts with multiple drivers trying to access the same hardware

8. Write short notes on the following:

a) fork() and exec() process model.

The basic principle of UNIX process management is to separate two operations: the creation of a process and the running of a new program. A new process is created by the fork() system call, and a new program is run after a call to exec(). These are two distinctly separate functions. A new process may be created with forkO without a new program being run—the new subprocess simply continues to execute exactly the same program that the first, parent process was running. Equally, running a new program does not require that a new process be created first: Any process may call exec 0 at any time. The currently running program is immediately terminated, and the new program starts executing in the context of the existing process. This model has the advantage of great simplicity. Rather than having to specify every detail of the environment of a new program in the system call that runs that program, new programs simply run in their existing environment. If a parent process wishes to modify the environment in which a new program is to be run, it can fork and then, still running the original program in a child process, make any system calls it requires to modify that child process before finally executing the new program. Under UNIX, then, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program. Under Linux, we can break down this context into a number of specific sections. Broadly, process properties fall into three groups: the process identity, environment, and context.

Free space management.

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a free-space list. The freespace list records all free disk blocks-those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. Free-space management is done using different techniques as explained hereunder.

Bit Vector

Frequently, the free-space list is implemented as a **bit map** or **bit vector**. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0. For example, consider a disk where blocks 2, 3,4,5, 8, 9,

10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be – 00111100111111000110000011100000

This technique is simple and efficient in finding the first free block, or *n* consecutive free blocks on the disk. But, bit vectors are inefficient unless the entire vector is kept in main memory. Keeping it in main memory is possible for smaller disks, such as on microcomputers, but not for larger ones.

Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. However, this scheme is not efficient. To traverse the list, we must read each block, which requires substantial I/O time. Fortunately, traversing the free list is not a frequent action. Usually, the OS simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used.

Grouping

A modification of the free-list approach is to store the addresses of *n* free blocks in the first free block. The first *n-1* of these blocks are actually free. The last block contains the addresses of another *n* free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly, unlike in the standard linked-list approach.

Counting

Another approach is to keep the address of the first free block and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count.