


CMR									
INSTITUTE OF TECHNOLOGY		USN	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>		
Internal Assessment Test – II									
Sub:	Operating Systems						Code:	18MCA25	
Date:	15.05.2019	Duration:	90 mins	Max Marks:	50	Sem:	II	Branch:	MCA
Answer Any FIVE FULL Questions									
Part-1							Marks	OBE	
								CO	RBT
1	Explain file allocation methods						[10]	CO4	L1
(OR)									
2(a)	What is Critical Section?						[2]	CO2	L1
	(b) Explain reader's writer's problem and write the solution using semaphore.						[8]	CO2	L2
Part-2									
3	Explain Demand paging in detail						[10]	CO3	L2
(OR)									
4	Explain the segmentation memory management. Describe the hardware support this is required for its implementation.						[10]	CO3	L2
Part-3							[5]	CO4	L2
5 (a)	What are the different file access methods? Explain briefly.								
(b)	Discuss directory implementation using						[5]	CO4	L1
	(i) Linear list								
	(ii) Hash table								
(OR)									

6	Explain Bankers algorithm in detail.	[10]	CO3	L2
Part-4				
7	State the dining philosophers' problem and give solution for the same, using semaphores.	[10]	CO3	L3
(OR)				
8 (a)	Explain i) Fragmentation ii) Thrashing	[5]	CO3	L2
(b)	Explain Linked List and grouping with respect to free space management.	[5]	CO4	L2
Part-5				
9(a)	Explain various file operations Mention the different page table structures. Explain in brief	[5]	CO4	L1
(b)		[5]	CO4	L2
(OR)				
10(a)	What is deadlock? Explain with a small diagram. What are the conditions occurring when deadlock arises?	[5]	CO3	L1
(b)	Explain how resource allocation graph is used to describe deadlocks.	[5]	CO3	L2

Internal Assessment Test 2 – May 2019

Sub:	Operating Systems						Code:	18MCA25	
Date:	15/05/2019	Duration:	90mins	Max Marks:	50	Sem:	II	Branch:	MCA

Note: Answer Any One FULL Question from each part.

1. Explain file allocation methods

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.

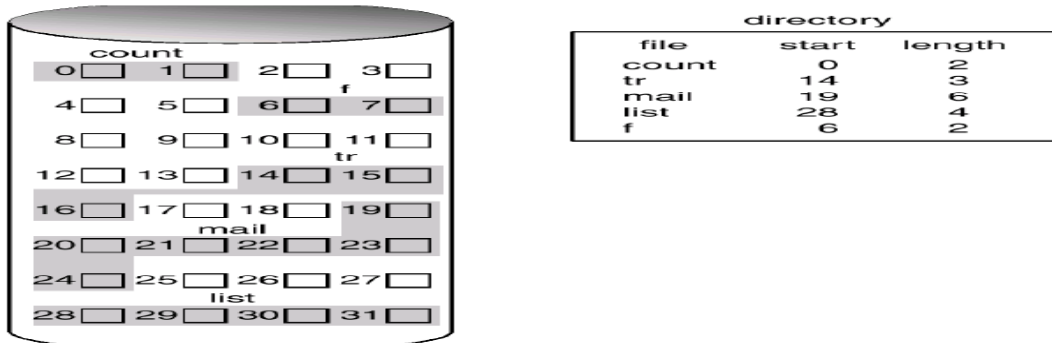


Figure 7.11 Contiguous allocation of disk space

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.

2.

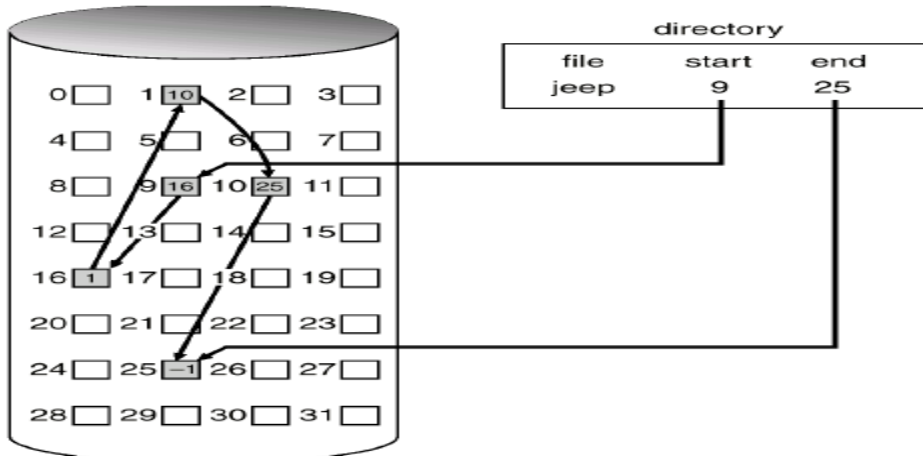


Figure 7.12 Linked Allocation of disk space

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 *i*th 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 *i*th entry in the index block points to the *i*th block of the file. The directory contains the address of the index block as shown in Figure 7.13.

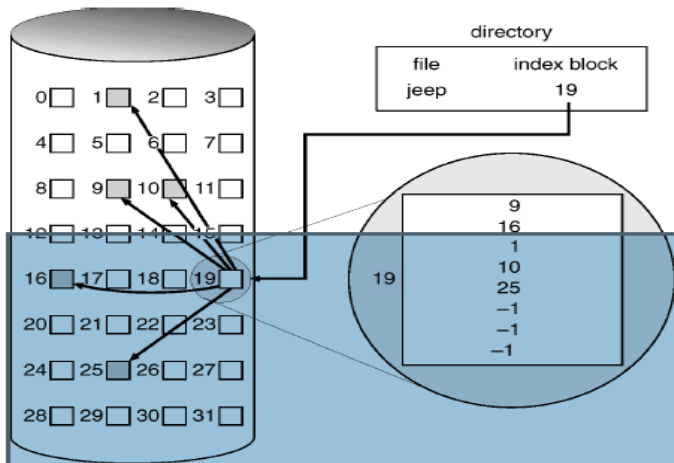


Figure 7.13 Indexed allocation of disk space

2a) What is Critical Section?

If *n* processes are competing to use some shared data. Each process has a code segment, called *critical section*, in which the shared data is accessed e.g. changing common variables, updating a table, writing a file etc. It is necessary to ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

2.b) Explain reader's writer's problem and write the solution using semaphore.

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it
- If there is at least one reader reading the data area, no writer may write to it.
- Readers only read and writers only write

Reader's have priority

Unless a writer has permission to access the object, any reader requesting access to the object will get it. Note this may result in a writer waiting indefinitely to access the object.

```

/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true) {
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}

```

Writers Have Priority

When a writer wishes to access the object, only readers which have already obtained permission to access the object are allowed to complete their access; any readers that request access after the writer has done so must wait until the writer is done. Note this may result in readers waiting indefinitely to access the object

The following semaphores and variables are added:

- A semaphore *rsem* that inhibits all readers while there is at least one writer desiring access to the data area
- A variable *writecount* that controls the setting of *rsem*
- A semaphore *y* that controls the updating of *writecount*

- A semaphore z that prevents a long queue of readers to build up on *rsem*

```

/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}

```

```

void writer ()
{
    while (true) {
        semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}

```

3. What is demand paging? Explain how TLB improves the performance of demand paging with neat diagram.

Demand paging is similar to paging system with swapping. Whenever process needs to be executed, only the required pages are swapped into memory. This is called as *lazy swapping*. As, the term *swapper* has a different meaning of 'swapping entire process into memory', another term *pager* is used in the discussion of demand paging.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. The pager brings only those necessary pages into memory. Hence, it decreases the swap time and the amount of physical memory needed.

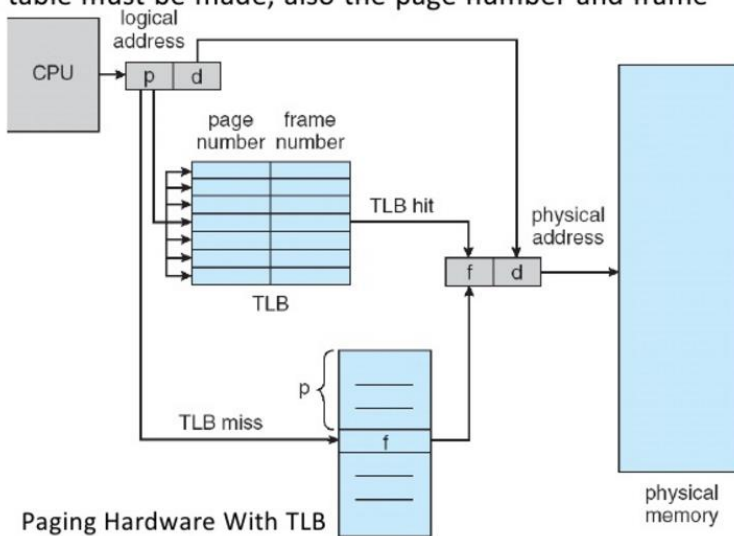
Demand Paging: Bring a page into memory only when it is needed.

- Less I/O needed
- Less memory needed
- Faster response
- More users

- Hardware implementation of Page Table is a set of high speed dedicated Registers
- Page table is kept in main memory and
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- The CPU dispatcher reloads these registers, instructions to load or modify the page-table registers are privileged
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware **cache** called **associative memory** or **translation look-aside buffers (TLBs)**
- TLB entry consists a key (or tag) and a value, when it is presented with an item, the item is compared with all keys simultaneously

Page #	Frame #

- When page number from CPU address is presented to the TLB, if the page number is found, its frame number is immediately available and is used to access memory.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made, also the page number and frame number to the TLB.
- If the TLB is full, the OS select one entry for replacement.
- Replacement policies range from LRU to random
- TLBs allow entries (for kernel code) to be wired down, so that they cannot be removed from the TLB.



4. Explain the segmentation memory management. Describe the hardware support this is required for its implementation

Segmentation is a memory-management scheme that supports user view of memory. A logical-address space is a collection of segments. Each address is specified in terms of the segment number and the offset within the segment. Here, a two dimensional user-defined addresses are mapped into one-dimensional physical addresses. This mapping is affected by a segment table. Each entry of the segment table has

- *segment base* – contains the starting physical address where the segment resides in memory
- *segment limit* – specifies the length of the segment

A logical address consists of two parts: a segment number, s and an offset d . The segment number is used as an index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the OS (logical addressing attempt beyond end of segment). If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs. The structure is as given in Figure 6.14.

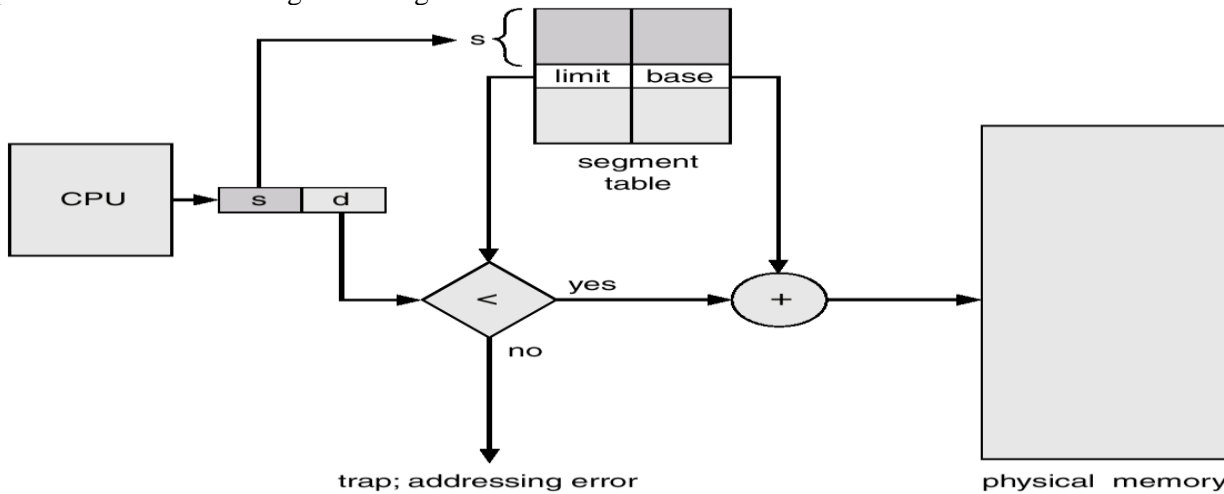


Fig 1 Segmentation Hardware

Example: Consider the situation shown in Figure 6.15. We have five segments numbered from 0 through 4. The segment table has a separate entry for each segment. It is giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at the location 4300.

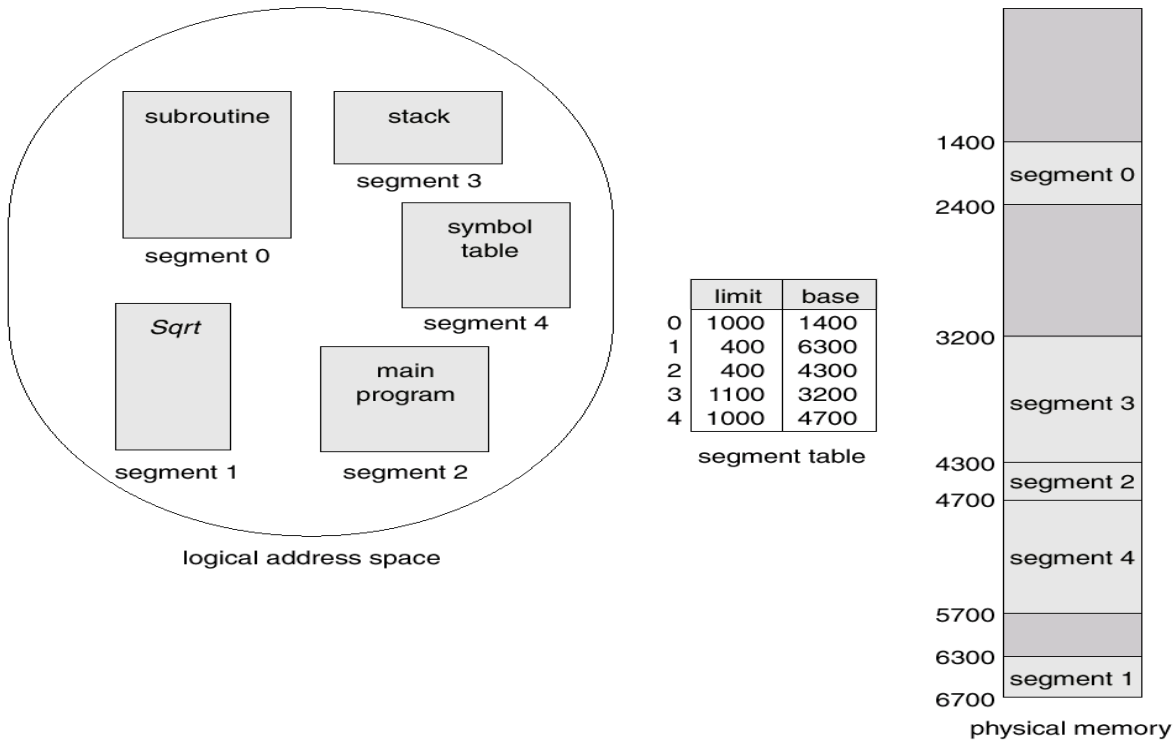


Fig : 2 Example of Segmentation

5. a) What are the different file access methods? Explain briefly.

There are several methods to access the information stored in the file. Some techniques are discussed here.

Sequential Access

It is the simplest method of file access. Here, information in the file are accessed one record after the other in an order. It works on the logic: *read next*, *write next*, *reset*. It is shown in Figure 7.1. (Note that, in programming languages like C, the functions like *fseek()*, *rewind()* etc can be used).

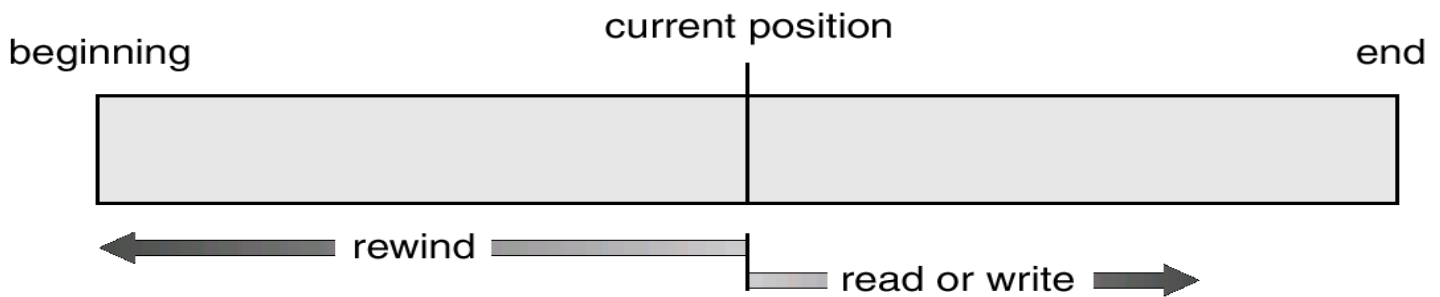


Figure 7.1 Sequential Access

Direct Access

Another method is *direct access* (or *relative access*). A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Other Access Methods

There are several access methods based on direct access method. One of such methods uses an index for a file. Index contains pointers to various blocks. To find a record in the file, we first search the index, and then use the pointer to access the file directly and to find the desired record. The working of indexed file is shown in Figure 7.2

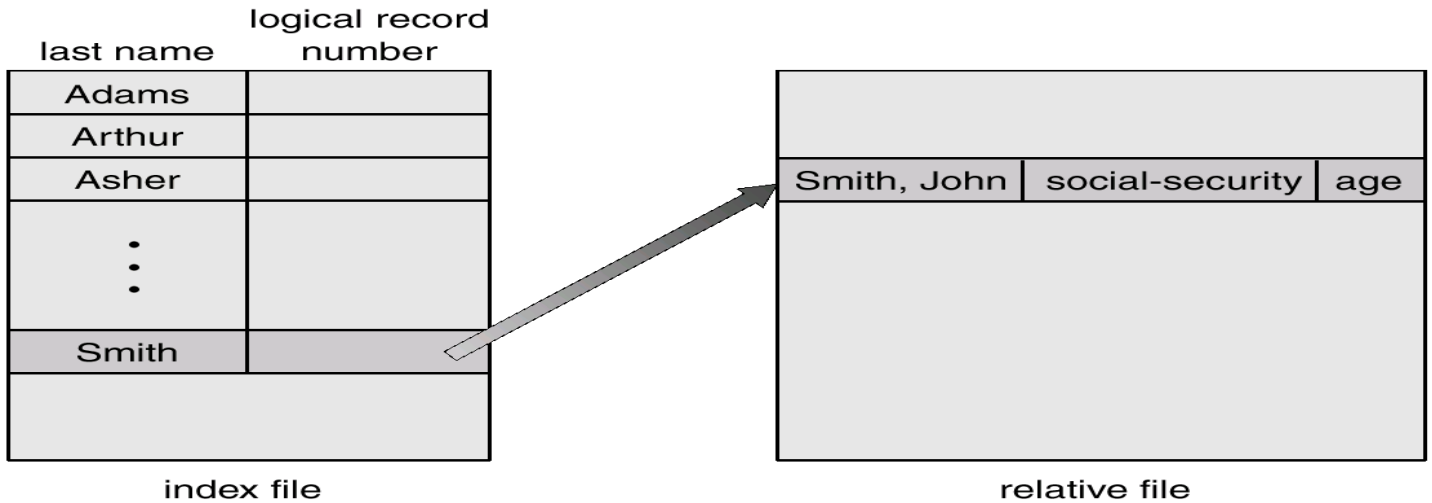


Figure 7.2 Example of index and relative files

b) Describe the methods used for implementing directories.

The selection of directory-allocation and directory-management algorithms has a effect on the efficiency, performance, and reliability of the file system.

7.9.1 Linear List

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. A linear list of directory entries requires a linear search to find a particular entry. This method is simple to program but time-consuming to execute.

- To **create** a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory.
 - To **delete** a file, we search the directory for the named file, then release the space allocated to it.
- To reuse the directory entry, we can do one of several things.
- Mark the entry as unused (by assigning it a special name, such as an all-blank name, or with a used-unused bit in each entry)
 - Attach it to a list of free directory entries
 - Copy the last entry in the directory into the freed location, and to decrease the length of the directory.

A linked list can also be used to decrease the time to delete a file.

The real disadvantage of a linear list of directory entries is the linear search to find a file.

Directory information is used frequently, and users would notice a slow implementation of access to it.

7.9.2 Hash Table

Another data structure that has been used for a file directory is a hash table. In this method, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also straightforward, although some provision must be made for collisions situations where two file names hash to the same location. The major difficulties with a hash table are its generally fixed size (if collision is resolved using linear probing) and the dependence of the hash function on that size

6. Explain Bankers algorithm in detail

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 R_j available.

□ **Max:** An $n \times m$ matrix defines 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 $n \times m$ matrix defines the number of resources currently allocated to each process. If $Allocation[i, j] = k$ then P_i is currently allocated k instances of R_j .

□ **Need:** An $n \times m$ matrix indicates remaining resource need of each process. If $Need[i, j] = k$, then P_i may need k more instances of R_j 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 $i = 1, 2, 3, \dots, n$.

2. Find an i such that both:

(a) $Finish[i] = false$

(b) $Need_i \leq Work$

If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$

$Finish[i] = true$

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 *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 .

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available - Request_i;$

$Allocation_i = Allocation_i + Request_i;$

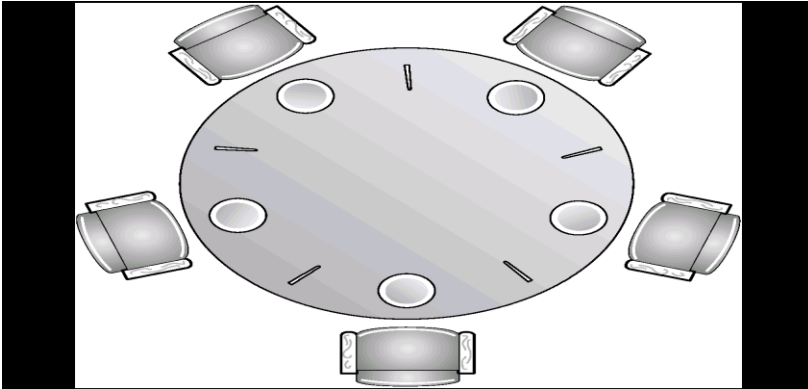
$Need_i = Need_i - Request_i;$

If the resulting resource allocation is safe, then the transaction is complete and the process P_i is allocated its resources. If the new state is unsafe, then P_i must wait for *Request_i*, and the old resource-allocation state is restored

7.a) State the dining philosophers' problem and give solution for the same, using semaphores.

Five philosophers spend their lives thinking and eating.

- Philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher.
- In center of the table is a bowl of rice (or spaghetti), and the table is laid with five single chopsticks.
- From time to time, 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.

- She cannot pick up a chopstick that is already in hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
- When she finishes eating, she puts down both of her chopsticks and start thinking again.

The problem is to ensure that no philosopher will be allowed to starve because he cannot ever pick up both forks.

The dining philosopher problem is considered a classic problem because it is an example of a large class of concurrency-control problems.

- Shared data
- semaphore chopstick[5];
- Initially all values are 1
- A philosopher tries to grab the chopstick by executing wait operation and releases the chopstick by executing signal operation on the appropriate semaphores.

```

/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher
(2),
            philosopher (3), philosopher (4));
}

```

Figure 6.12 A First Solution to the Dining Philosophers Problem

```

/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
             philosopher (3), philosopher (4));
}

```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

This solution guarantees that no two neighbors are eating simultaneously but it has a possibility of creating a deadlock and starvation.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks if both chopsticks are available.
- An odd philosopher picks up her left chopstick first and an even philosopher picks up her right chopstick first.
- Finally no philosopher should starve.

8.a Explain

i) Fragmentation

ii) Thrashing

Sometimes, the size of the hole is much smaller than the overhead required to track it. Hence, normally, physical memory is divided into fixed-size block. Then memory is allocated in terms of units. So, sometimes, the memory allocated for a process may be slightly larger than what it requires. Such a difference is known as **internal fragmentation**.

For example, if the physical memory is divided into block of 4 bytes, and the process requests for 6 bytes, then it will be allocated with 2 blocks. Hence, the total allocation is 8 bytes leading to 2 bytes of internal fragmentation.

One of the solutions to external fragmentation is **compaction**: shuffling of memory contents to place all free memory together into one large block. But, compaction is possible only when relocation is dynamic and is done during execution time. That is, if the relocation is static, we cannot apply this technique.

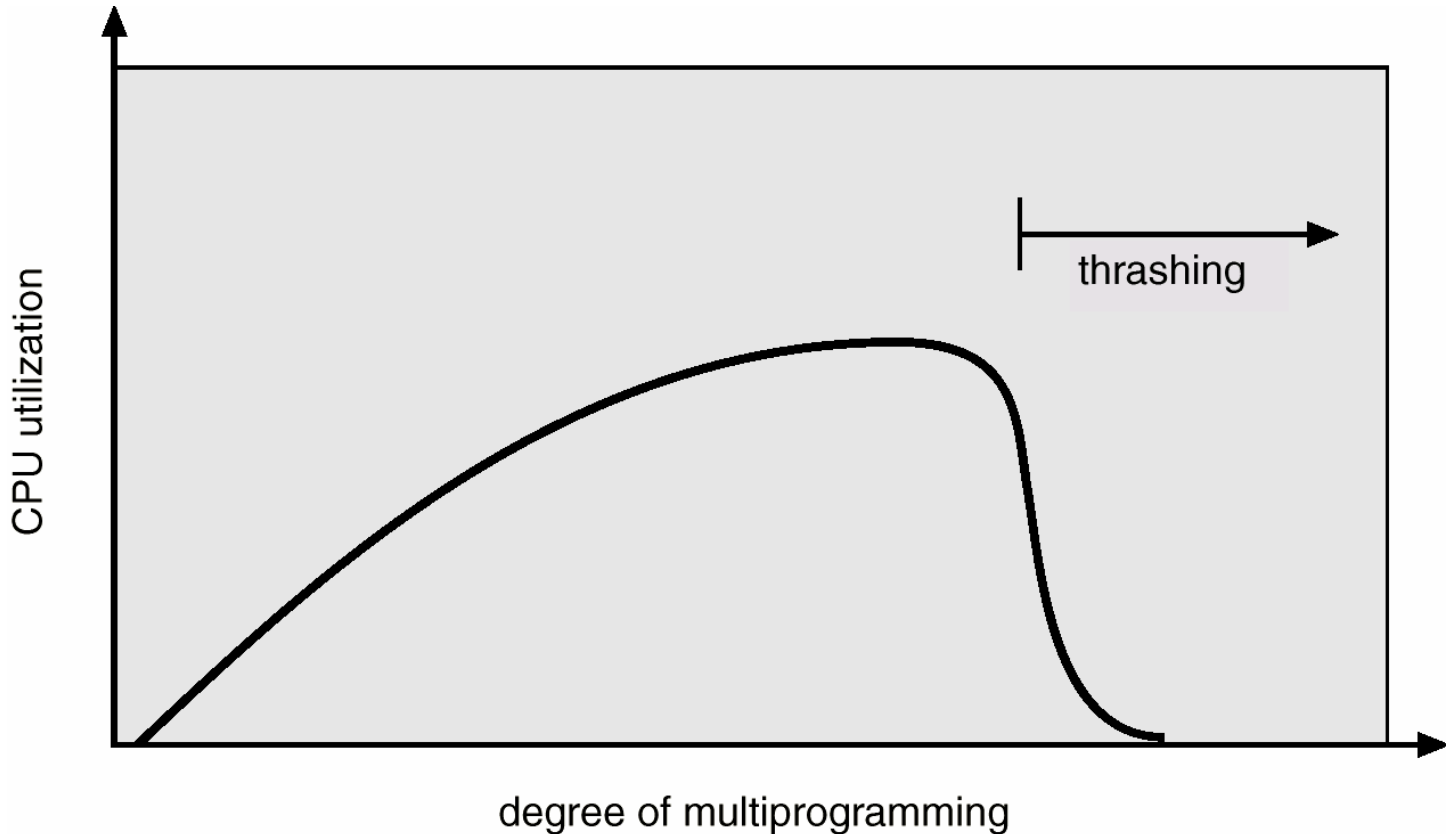
Another solution for external fragmentation is to permit logical-address space of a process to be non-contiguous. This allows a process to be allocated physical memory wherever it is available. To do so, two techniques are there:

- Paging
- Segmentation

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 3.24. The effects of thrashing can be limited by using local replacement algorithm.



8.b. Explain Linked List and grouping with respect to 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.

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.

9(a) Explain file operations.

File is an abstract data type. To define it properly, we need to define certain operations on it:

- Creating a file: This includes two steps: find the space in file system and make an entry in the directory.
- Writing a file: To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Using the name of the file, the system searches the directory to find the location of the file. The system must keep a *write* pointer to the location in the file where the next write is to take place. The *write* pointer must be updated whenever a write occurs.
- Reading a file: To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.
- Repositioning within a file: The directory is searched for the appropriate entry, and the current-file-position is set to a given value. This file operation is also known as *file seek*.
- Deleting a file: To delete a file, search the directory. Then, release all file space and erase the directory entry.
- Truncating a file: The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, the truncation allows all attributes to remain unchanged-except for file length. The file – length is reset to zero and its file space released.

Other common operations include *appending* new information to the end of an existing file and *renaming* an existing file. Most of the file operations involve searching the directory for the entry associated with the named file. To avoid this constant searching, the OS keeps small table (known as *open – file table*) containing information about all open files. When a file operation is requested, this table is checked. When the file is closed, the OS removes its entry in the open-file table.

Every file which is open has certain information associated with it:

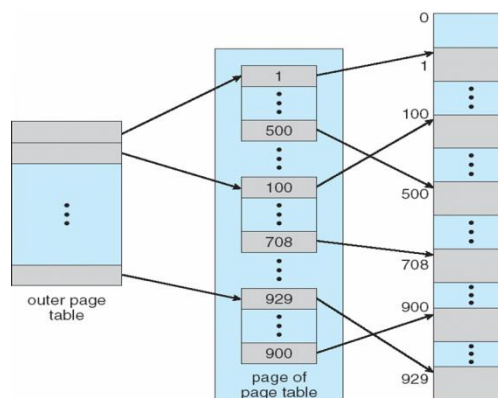
- File pointer: Used to track the last read-write location. This pointer is unique to each process.
- File open count: When a file is closed, its entry position (the space) in the open-file table must be reused. Hence, we need to track the number of opens and closes using the file open count.
- Disk location of the file: Most file operations require the system to modify data within the file. So, location of the file on disk is essential.
- Access rights: Each process opens a file in an access mode (read, write, append etc).

This information is by the OS to allow or deny subsequent I/O requests.

9(b) Mention the different page table structures. Explain in brief

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.



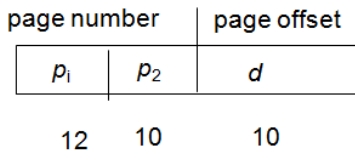
A logical address (on 32-bit machine with 1K page size) is divided into:

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

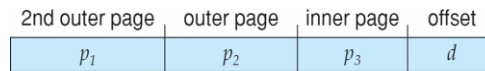
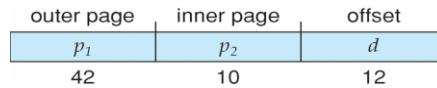
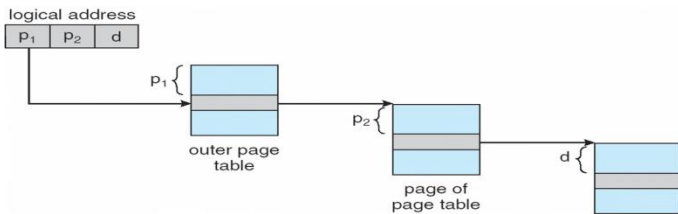
Since the page table is paged, the page number is further divided into:

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

Thus, a logical address is as follows:



where p_1 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 –

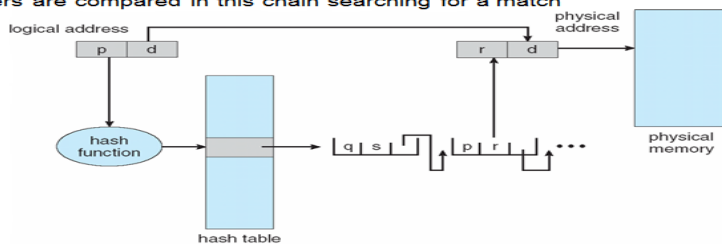
Common in address spaces > 32 bits

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 –

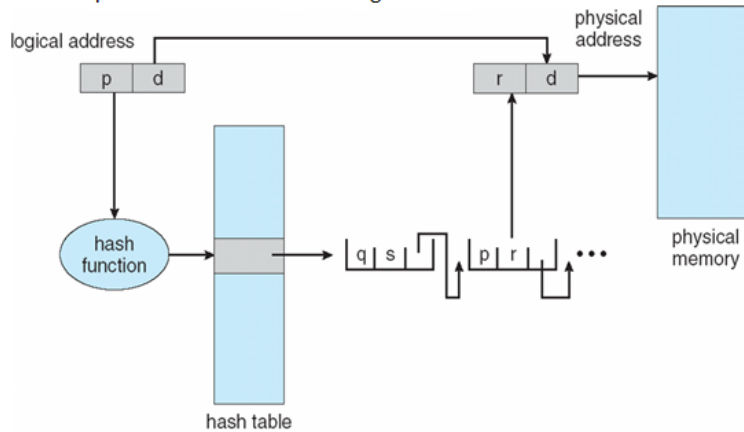
Common in address spaces > 32 bits

The virtual page number is hashed into a page table

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

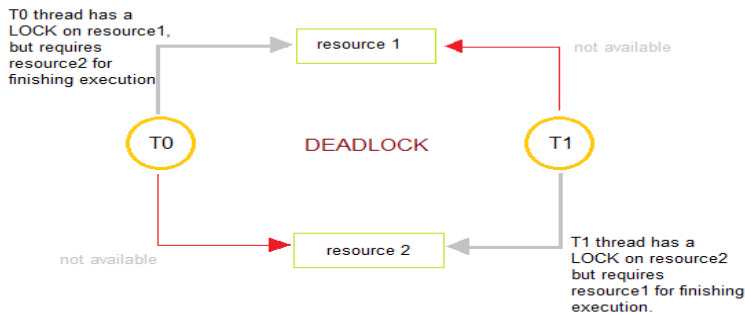
Virtual page numbers are compared in this chain searching for a match

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



10.a What is deadlock? Explain with a small diagram. What are the conditions occurring when deadlock arises?

Deadlock can be defined as the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set. Deadlock is permanent because none of the events is ever triggered



Deadlock can arise if four conditions hold simultaneously:

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Mutual exclusion

At least one resource must be non-sharable mode i.e. only one process can use a resource at a time. The

requesting process must be delayed until the resource has been released. But mutual exclusion is required to ensure consistency and integrity of a database.

Hold and wait

A process must be holding at least one resource and waiting to acquire additional resources held by other processes.

No preemption A resource can be released only voluntarily by the process holding it after that process has completed its task i.e. no resource can be forcibly removed from a process holding it.

Circular wait

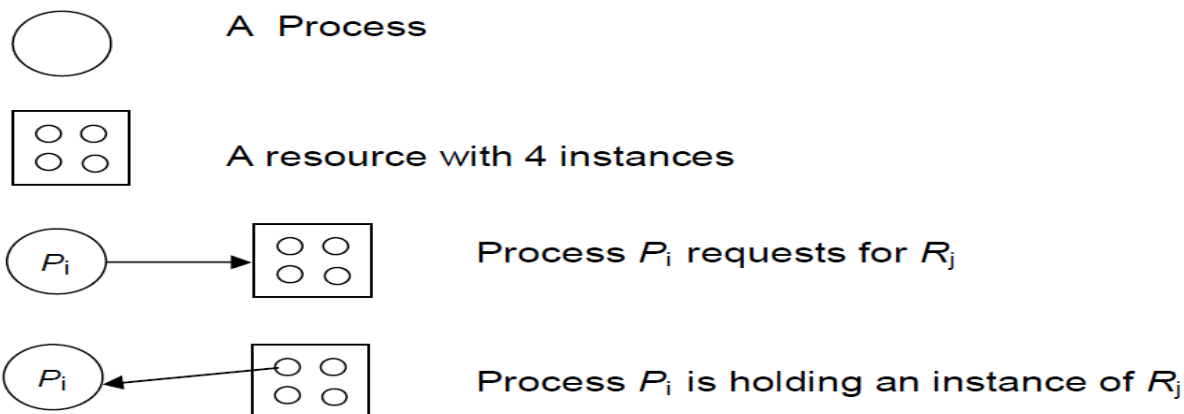
There exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2, \dots, P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

10b) Explain how resource allocation graph is used to describe deadlocks

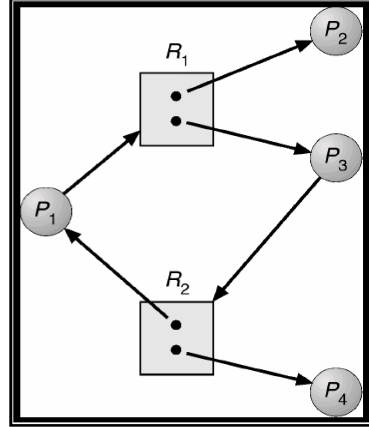
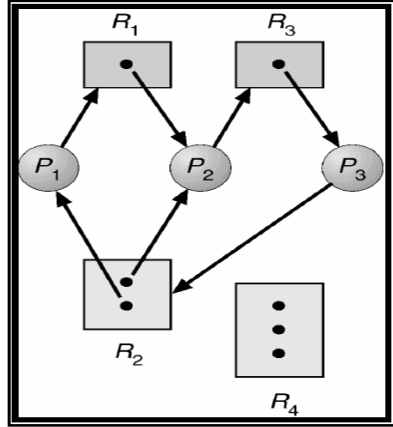
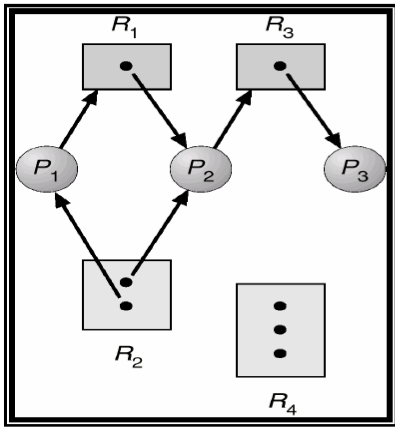
The resource allocation graph is a directed graph that depicts a state of the system of resources and processes with each process and each resource represented by a node. It is a graph consisting of a set of vertices V and a set of edges E with following notations:

- V is partitioned into two types:
 - o $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - o $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
- **Request edge:** A directed edge $P_i \rightarrow R_j$ indicates that the process P_i has requested for an instance of the resource R_j and is currently waiting for that resource.
- **Assignment edge:** A directed edge $R_j \rightarrow P_i$ indicates that an instance of the resource R_j has been allocated to the process P_i

The following symbols are used while creating resource allocation graph:



Examples of resource allocation graph are shown in Figure 5.1. Note that, in Figure 5.1(c), the processes P_2 and P_4 are not depending on any other resources. And, they will give up the resources R_1 and R_2 once they complete the execution. Hence, there will not be any deadlock.



(a) Resource allocation Graph (b) With a deadlock (c) with cycle but no deadlock
 Figure 5.1 Resource allocation graphs