CMR
INSTITUTE OF
TECHNOLOGY

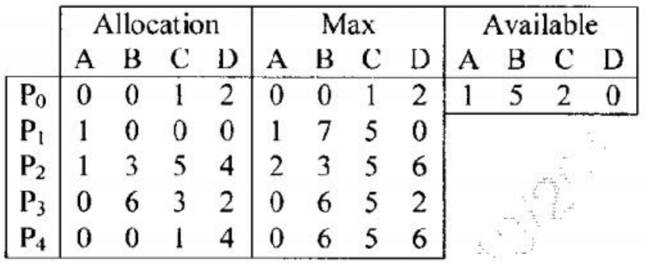
USN | | | | | | | | | |

Internal Assesment Test – II

Sub:	Operating Systems	Code:	16MCA24
Date: 09.05.2017	Duration: 90 mins	Max Marks: 50	Sem: II	Branch: MCA

Answer Any FIVE FULL Questions

		Marks	OBE CO	OBE RBT
1(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
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
3(a)	What is demand paging? Explain how TLB improves the performance of demand paging with neat diagram.	[10]	CO3	L2
4 (a)	Explain the segmentation memory management. Describe the hardware support this is required for its implementation.	[10]	CO3	L2
5 (a)	What are the different file access methods? Explain briefly.	[5]	CO4	L2
(b)	Describe the methods used for implementing directories.	[5]	CO4	L1
6 (a) (b)	Consider the snapshot of a system and answer the following using Banker's algorithm: i) What is the content of matrix 'Need'? ii) Is the system in a safe state? iii) If a request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately?	[10]	CO3	L4



	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

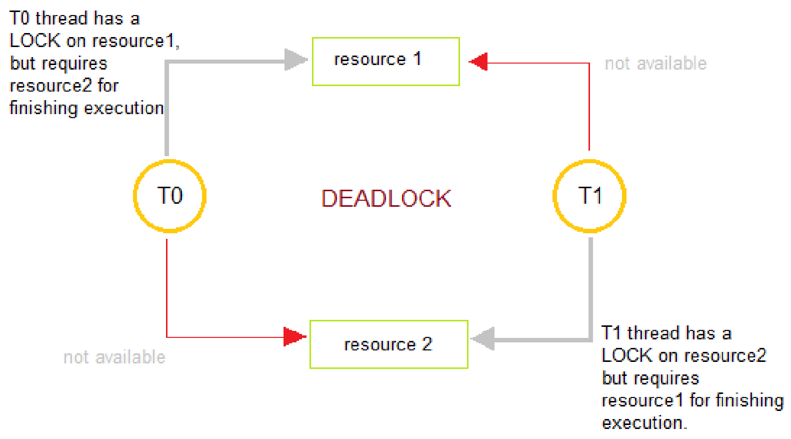
		Marks	CO	RBT
7 (a)	State the dining philosophers' problem and give solution for the same, using semaphores.	[10]	CO3	L3
8 (a)	Consider the following page reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 Assuming 3 frames, find the number of page faults when the following algorithms are used: i) LRU ii) FIFO iii) Optimal. Note that initially all the frames are empty.	[10]	CO4	L4

Internal Assesment Test – II Answer Key

Sub: Operating Systems Code: 16MCA24

1. a) What is deadlock? Explain with a small diagram. What are the necessary conditions for the deadlock to occur in a system

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


T0 thread has a LOCK on resource1, but requires resource2 for finishing execution
resource 1
not available
T0
DEADLOCK
T1
not available
resource 2
T1 thread has a LOCK on resource2 but requires resource1 for finishing execution.

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, …, *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,…… …, *Pn*–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.

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

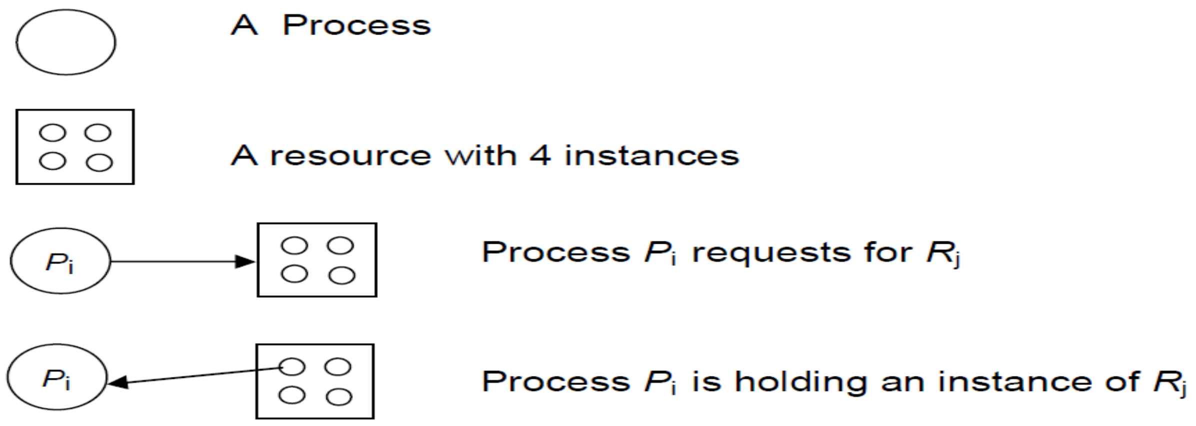
The resource allocation graph is a directed graph that depicts a state of the system of resources and processes with each process and each resource represented by a node. Itis a graph consisting of a set of vertices *V* and a set of edges *E* with following notations:
- V is partitioned into two types:

o *P* = {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system.
o *R* = {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system.

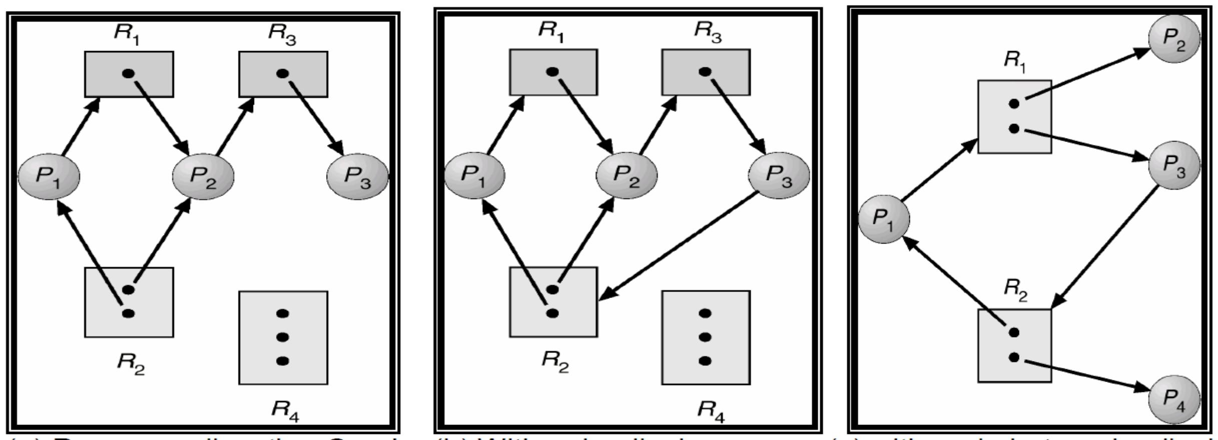
 Request edge: A directed edge *P*i *Rj* indicates that the process *P*i has requested for an instance of the resource *Rj* and is currently waiting for that resource.

 Assignment edge: A directed edge *Rj* *Pi* indicates that an instance of the resource *Rj* has been allocated to the process *Pi*

The following symbols are used while creating resource allocation graph:


A Process
A resource with 4 instances
Pi
Process Pi requests for Rj
Pi
Process Pi is holding an instance of Rj

Examples of resource allocation graph are shown in Figure 5.1. Note that, in Figure 5.1(c), the processes P2 and P4 are not depending on any other resources. And, they will give up the resources R1 and R2 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

2. a) 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.

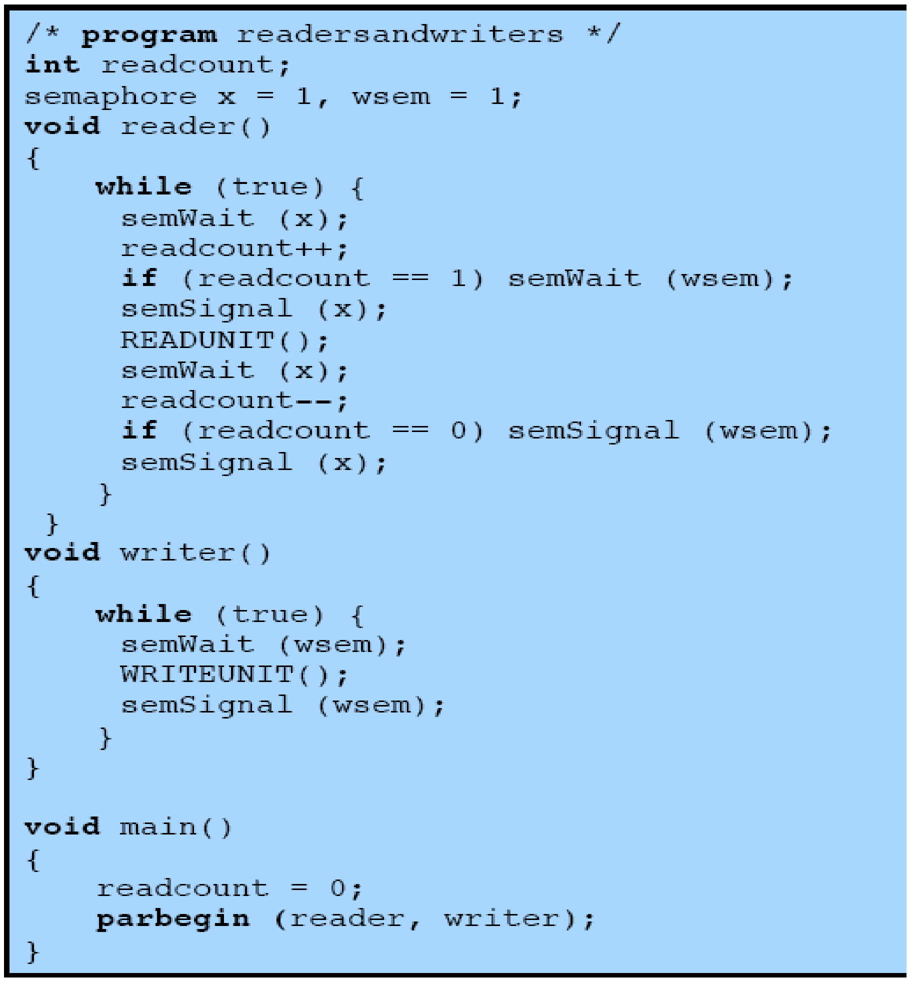
2.b) Explain reader's writer's problem and write the solution using semaphore.

- 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);
}
```

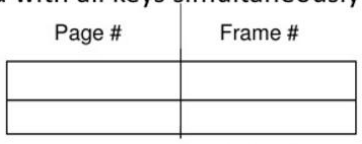
3.a) What is demand paging? Explain how TLB improves the performance of demand paging with neat diagram.

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.

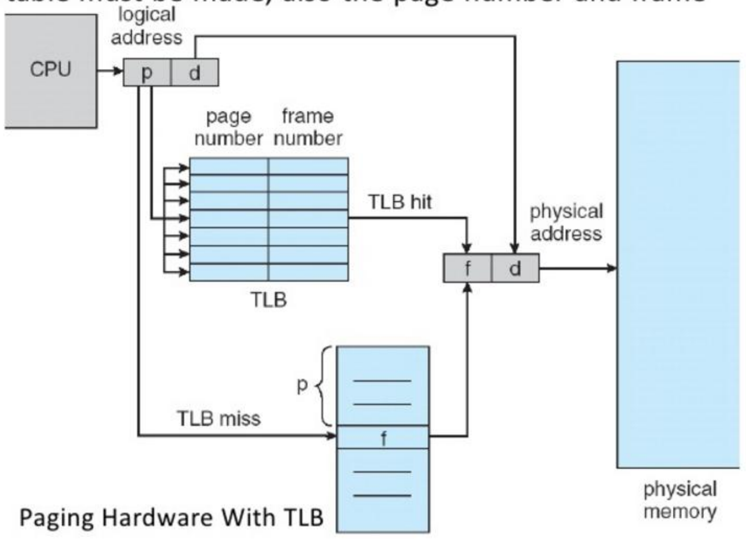
Demand Paging: Bring a page into memory only when it is needed.

-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



- 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.



Paging Hardware With TLB

4. a) 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.

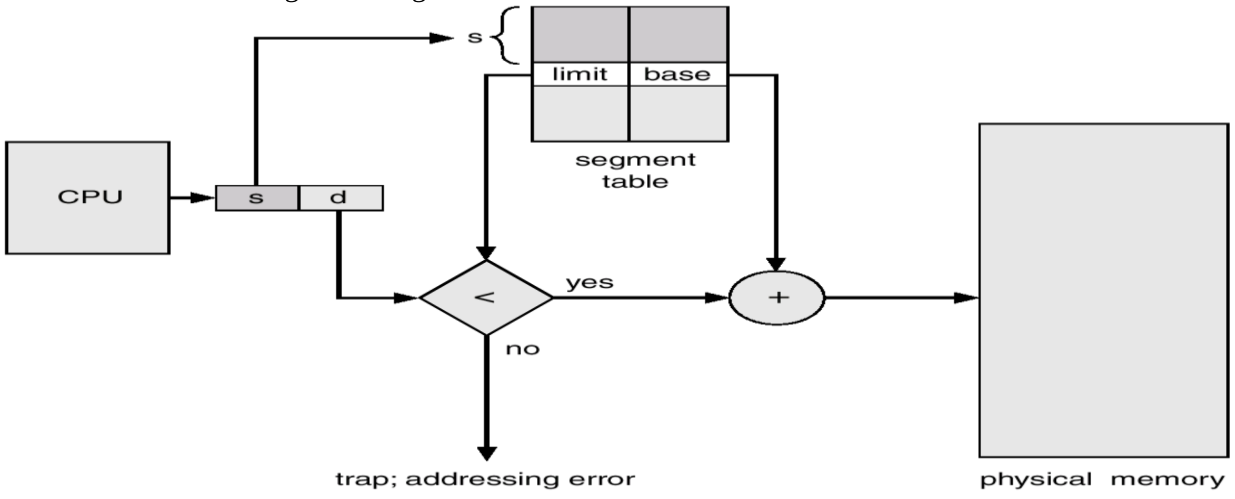


Fig 1 Segmentation Hardware

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.

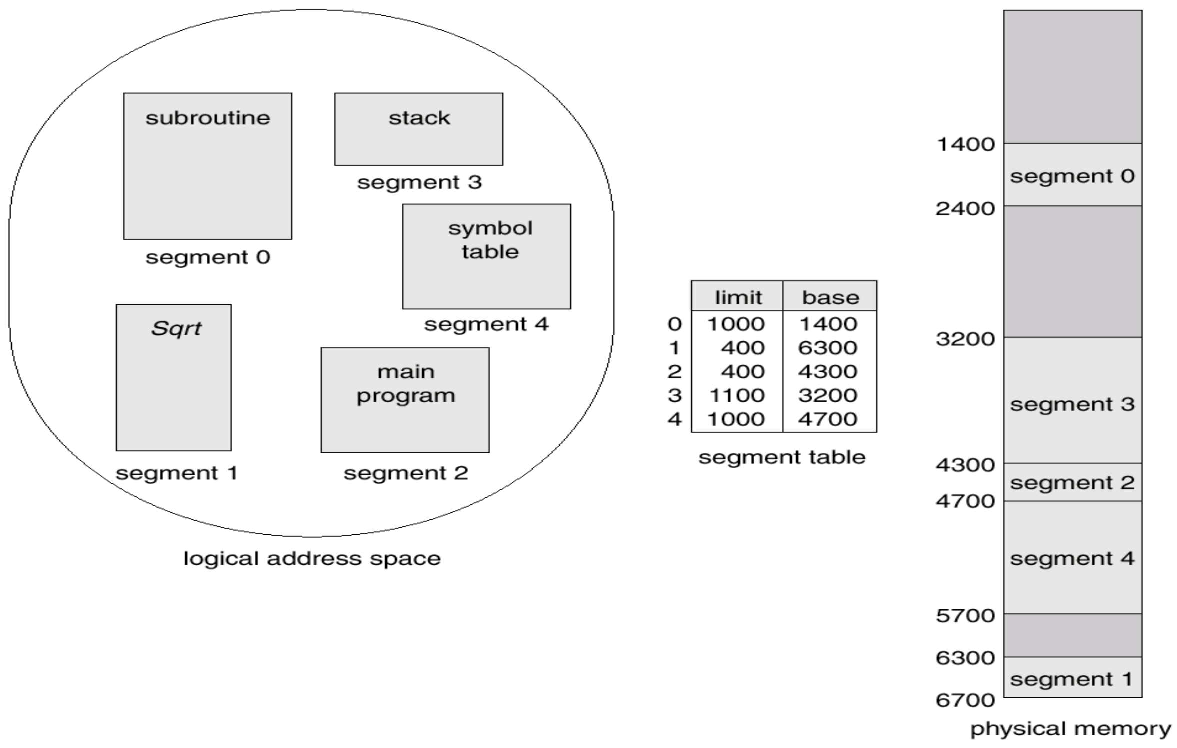


Fig : 2 Example of Segmentation

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).

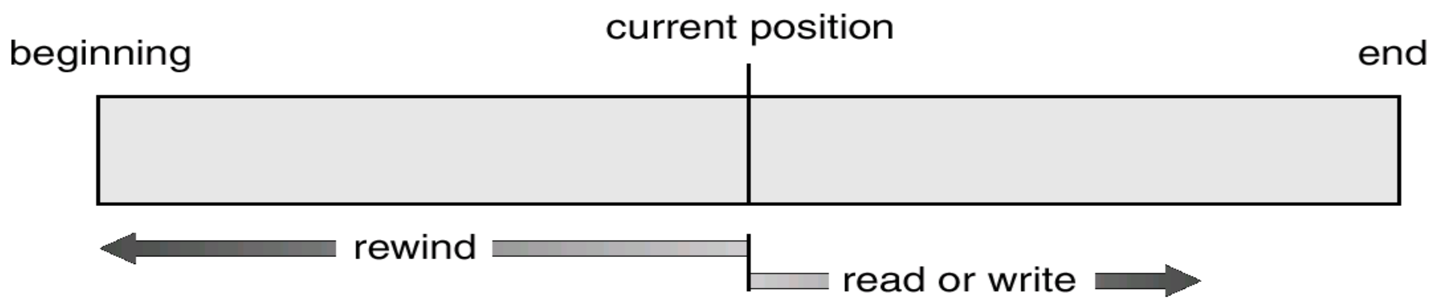


Figure 7.1 Sequential Access

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

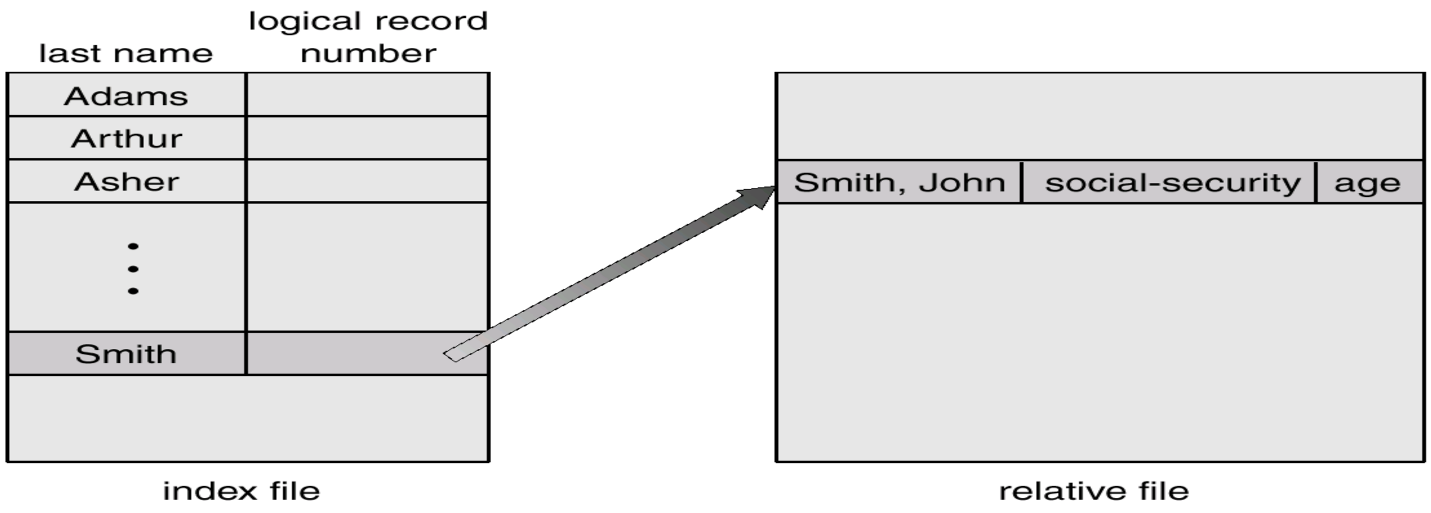
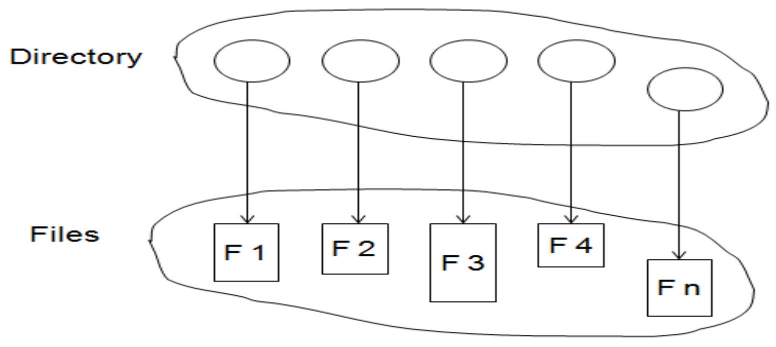


Figure 7.2 Example of index and relative files

b) Describe the methods used for implementing directories.

Directory is:

A collection of nodes containing information about all files.



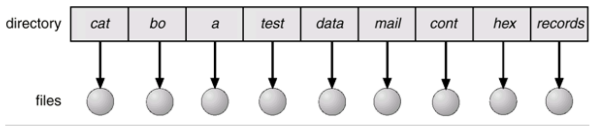
Both the directory structure and the files reside on disk.

Backups of these two structures are kept on tapes.

Directory Structures:

Single-Level Directory

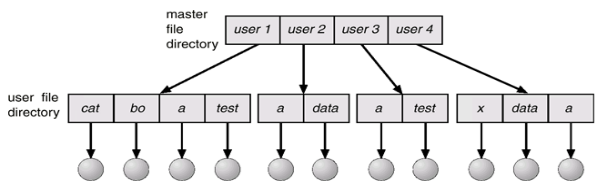
- A single directory for all users.



- Naming problem
- Grouping problem

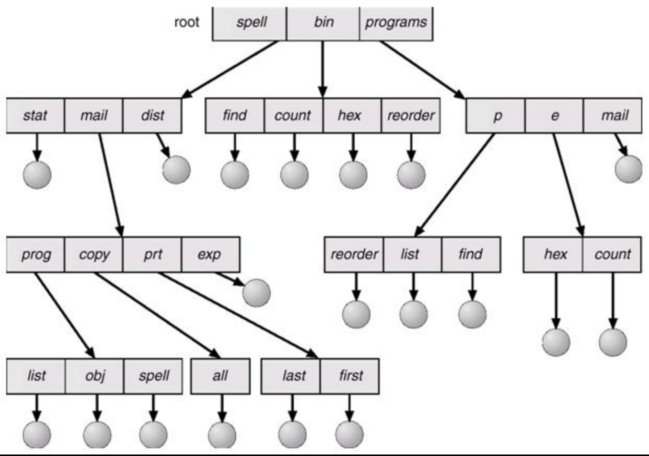
Two-Level Directory

- Separate directory for each user.



- Path name
- Can have the saem file name for different user
- Efficient searching
- No grouping capability

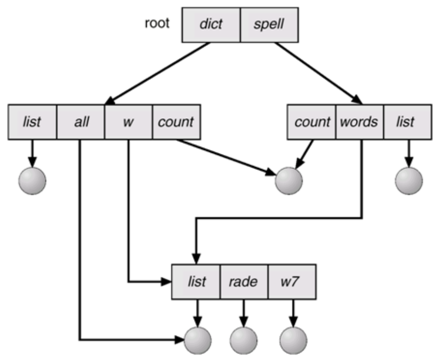
Tree-Structured Directories



- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - **cd** /spell/mail/prog
  - **type** list

Acyclic-Graph Directories

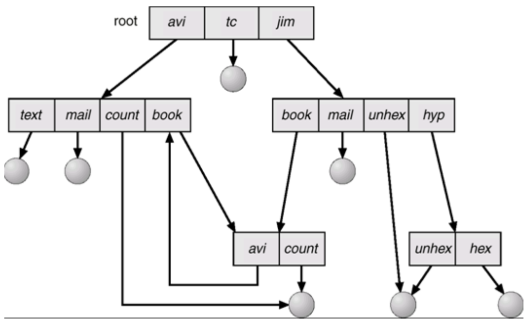
- Have shared subdirectories and files.



- Two different names (aliasing)
- If *dict* deletes *list* ⇒ dangling pointer.

  Solutions:
  - Backpointers, so we can delete all pointers. Variable size records a problem.
  - Backpointers using a daisy chain organization.
  - Entry-hold-count solution.

General Graph Directory



- How do we guarantee no cycles?
  - Allow only links to file not subdirectories.
  - Garbage collection.
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK.

6.a) Consider the snapshot of a system and answer the following using Banker's algorithm:

i) What is the content of matrix 'Need'?

ii) Is the system in a safe state?

iii) If a request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately?

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Solution (a)

What is the content of the matrix *Need*?

	Need A B C D
P_0	0 0 0 0
P_1	0 7 5 0
P_2	1 0 0 2
P_3	0 0 2 0
P_4	0 6 4 2

Solution (b)

Is the system in a safe state?

Yes, there exist several sequences that satisfy safety requirements (e.g. P_0, P_2, P_1, P_3, P_4).

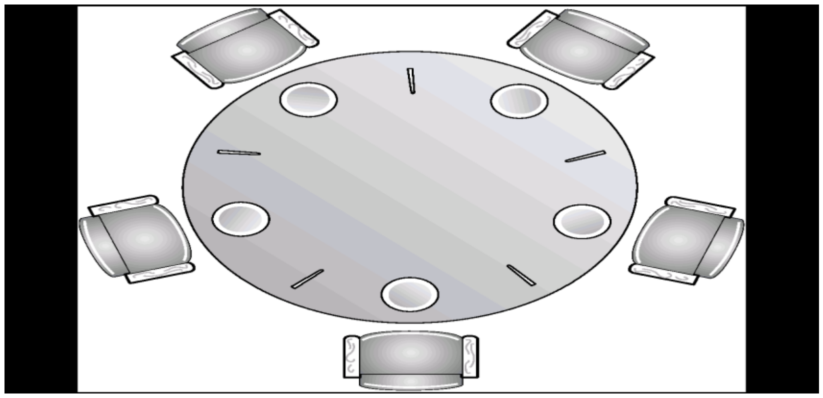
Solution (c)

If a request from process P_1 arrives for (0, 4, 2, 0), can the request be granted immediately?

Pretend that the allocation can be made since the Available matrix is (1, 5, 2, 0), and it will now change to (1, 1, 0, 0). The next step is to find the safe sequence of processes. Alloc for P_1 becomes (1,4,2,0) and Need for P_1 becomes (0, 3, 3, 0).
One possible safe sequence is: P_0, P_2, P_3, P_1, P_4.

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).

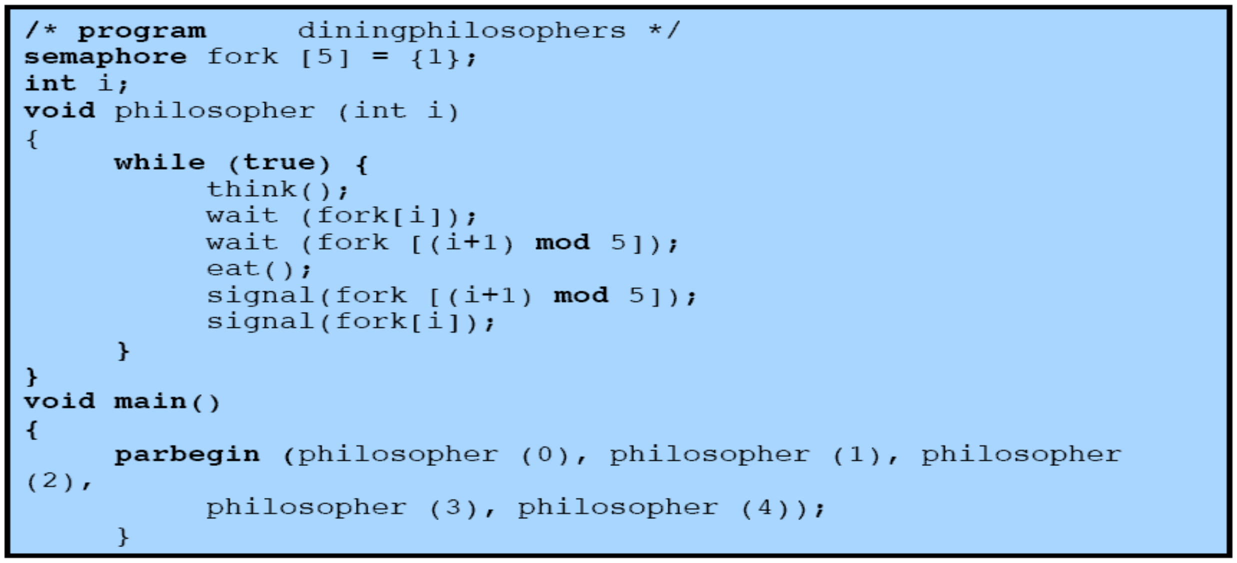


A philosopher may pick up only one chopstick at a time.
 She cannot pick up a chopstick that is already in hand of a neighbor.
 When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
 When she finishes eating, she puts down both of her chopsticks 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.

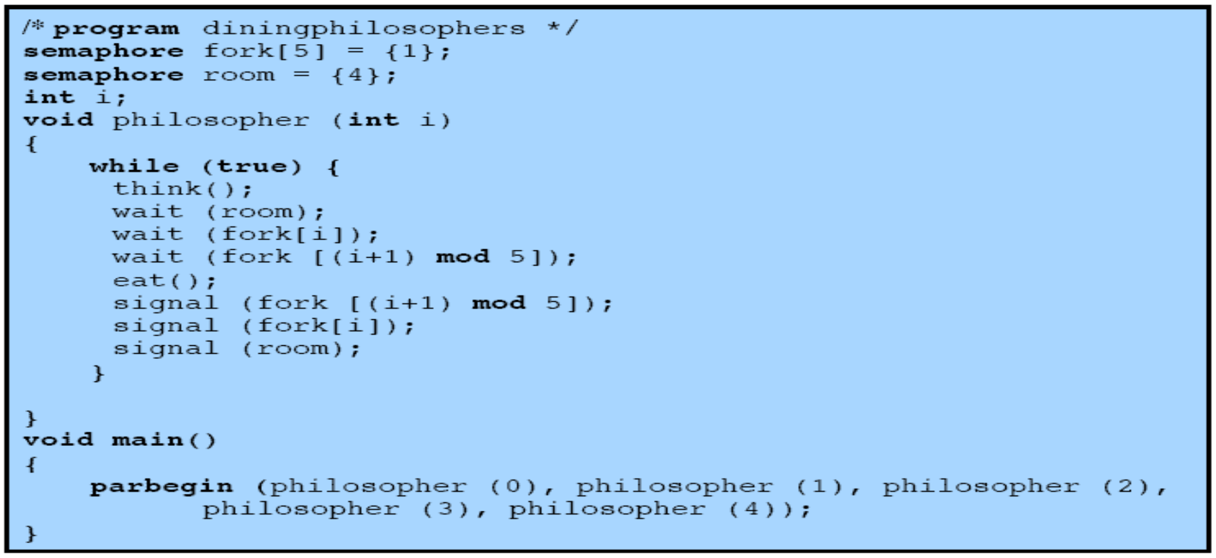
The dinning philosopher problem is considered a classic problem because it is an example of a large class of concurrency-control problems.

- 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) Consider the following page reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

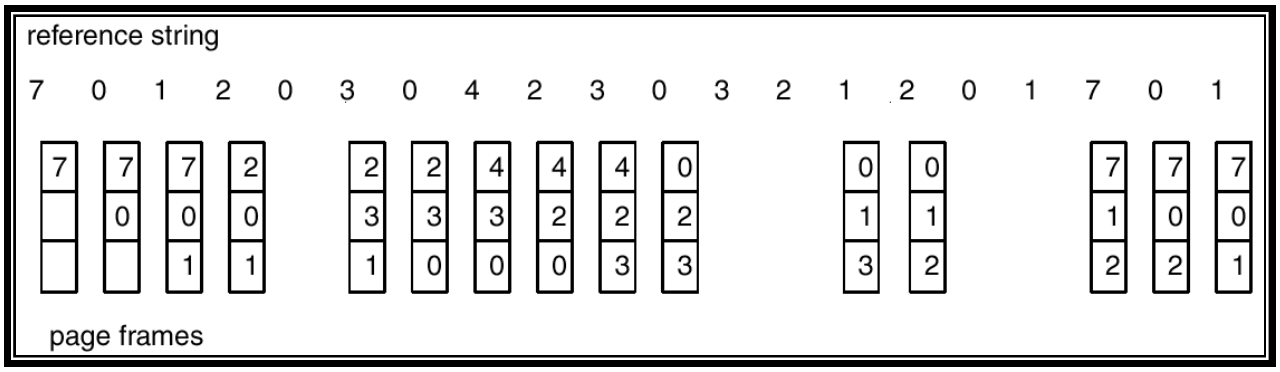
Assuming 3 frames, find the number of page faults when the following algorithms are used: i) LRU ii) FIFO iii) Optimal. Note that initially all the frames are empty.

FIFO Page Replacement

It is the simplest page – replacement algorithm. As the name suggests, the first page which has been brought into memory will be replaced first when there no space for new page to arrive. Initially, we assume that no page is brought into memory. Hence, there will be few (that is equal to number of frames) page faults, initially. Then, whenever there is a request for a page, it is checked inside the frames. If that page is not available, page – replacement should take place.

Example: Consider a reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Let the number of frames be 3.

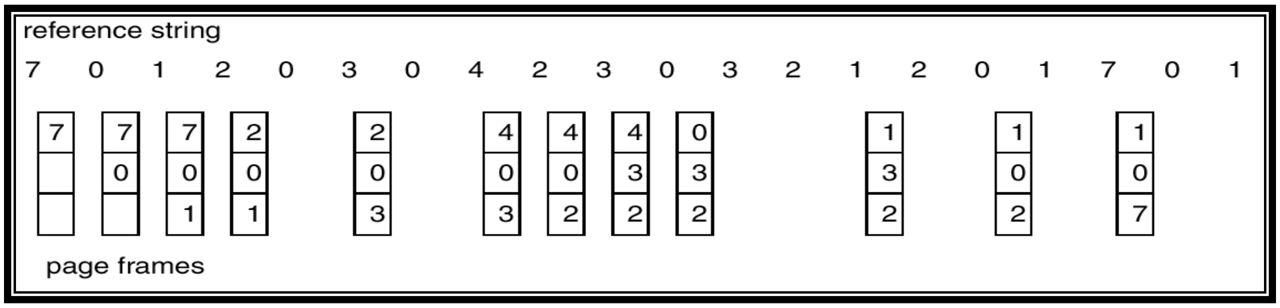


In the above example, there are 15 page faults.

LRU Page Replacement

Least Recently Used page replacement algorithm states that: **Replace the page that has not been used for the longest period of time.** This algorithm is better than FIFO.

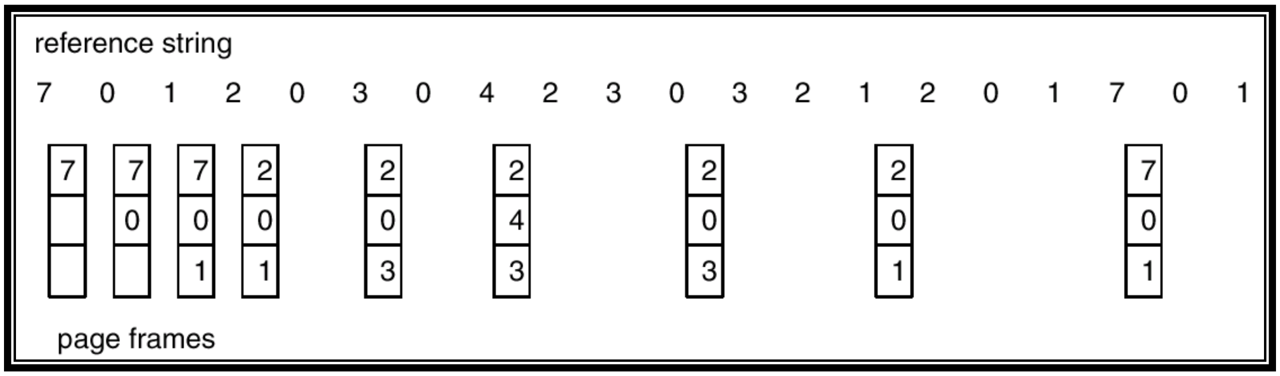
Example:



Here, number of page faults = 12

Optimal Page Replacement Algorithm

An Optimal Page Replacement algorithm (also known as ***OPT*** or ***MIN*** algorithm) do not suffer from Belady's anomaly. It is stated as: **Replace the page that will not be used for the longest period of time.**



Here, number of page faults = 9

This algorithm results in lowest page – faults.