Sub:	INTERNAL TEST – II Answer Key **Operating Systems**	Code:17MCA24

1 (a) Compare preemptive with non preemptive scheduling — 3

BASIS FOR COMPARISON	PREEMPTIVE SCHEDULING	NON PREEMPTIVE SCHEDULING
Basic	The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Interrupt	Process can be interrupted in between.	Process can not be interrupted till it terminates or switches to waiting state.
Starvation	If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Overhead	Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Flexibility	Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Cost	Preemptive scheduling is cost associated.	Non-preemptive scheduling is not cost associative.

(b) Discuss about the OS control structures — 7

- To manage the processes, the OS should have the information about the current status of each process and resource.
- For this purpose OS constructs and maintains tables for each entity
- The four types of tables maintained by OS are explained here.

Memory Table: Used to keep track of both main and secondary memory. They must include the following information:

- Allocation of main memory to processes
- Allocation of secondary memory to processes
- Protection attributes for access to shared memory regions
- Information needed to manage virtual memory

I/O Table: Used by OS to manage the I/O devices and the channels. At any given moment of time, the OS needs to know

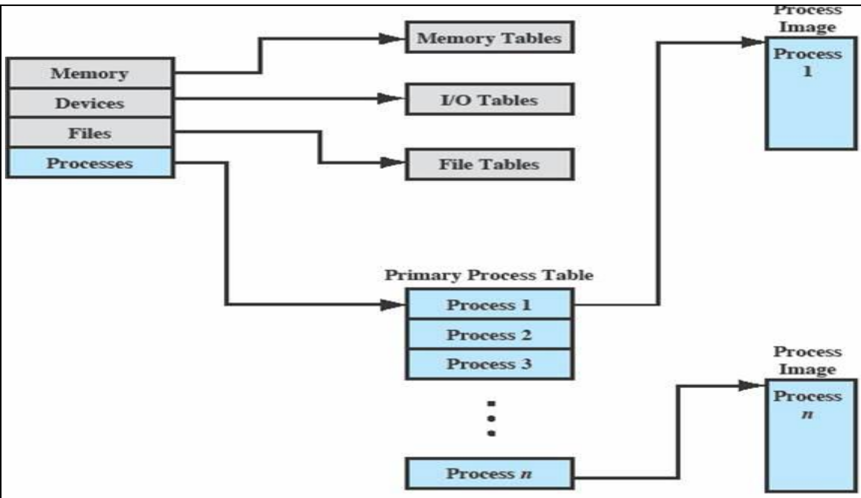
- Whether the I/O device is available or assigned
- The status of I/O operation
- The location in main memory being used as the source or destination of the
- I/O transfer

File Table: Most of the times, these tables are maintained by file management system. These tables provide information about:

- Existence of files
- Location on secondary memory
- Current Status
- Any other relevant attributes

Process Table: To manage processes the OS needs to know following details of the processes

- Current state
- Process ID
- Location in memory



2 Explain the five state process model with transition diagram, showing how to change process model for a suspend process 10

PROCESS

A process can be defined in several ways:

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources.

Two essential elements of a process are:

- **program code:** which may be shared with other processes that are executing the same program
- **Set of data:** associated with that code

The various States of the Process are as Followings:-

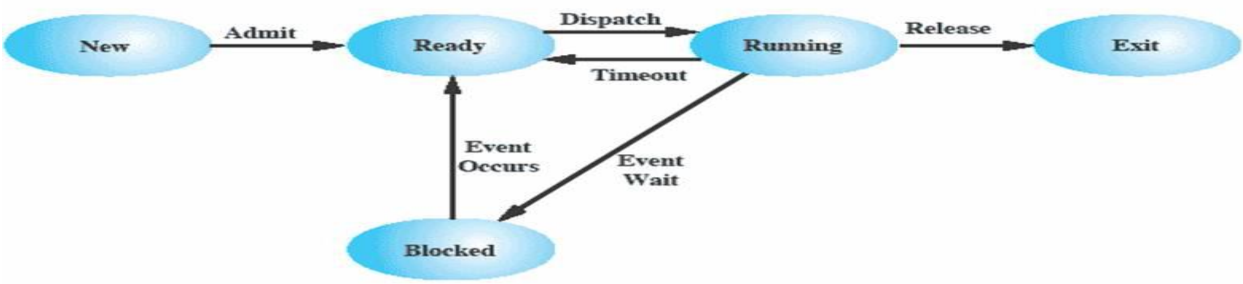
1) **New State:** When a user request for a Service from the System , then the System will first initialize the process or the System will call it an initial Process . So Every new Operation which is Requested to the System is known as the New Born Process.

2) **Running State:** When the Process is Running under the CPU, or When the Program is Executed by the CPU , then this is called as the Running process and when a process is Running then this will also provides us Some Outputs on the Screen.

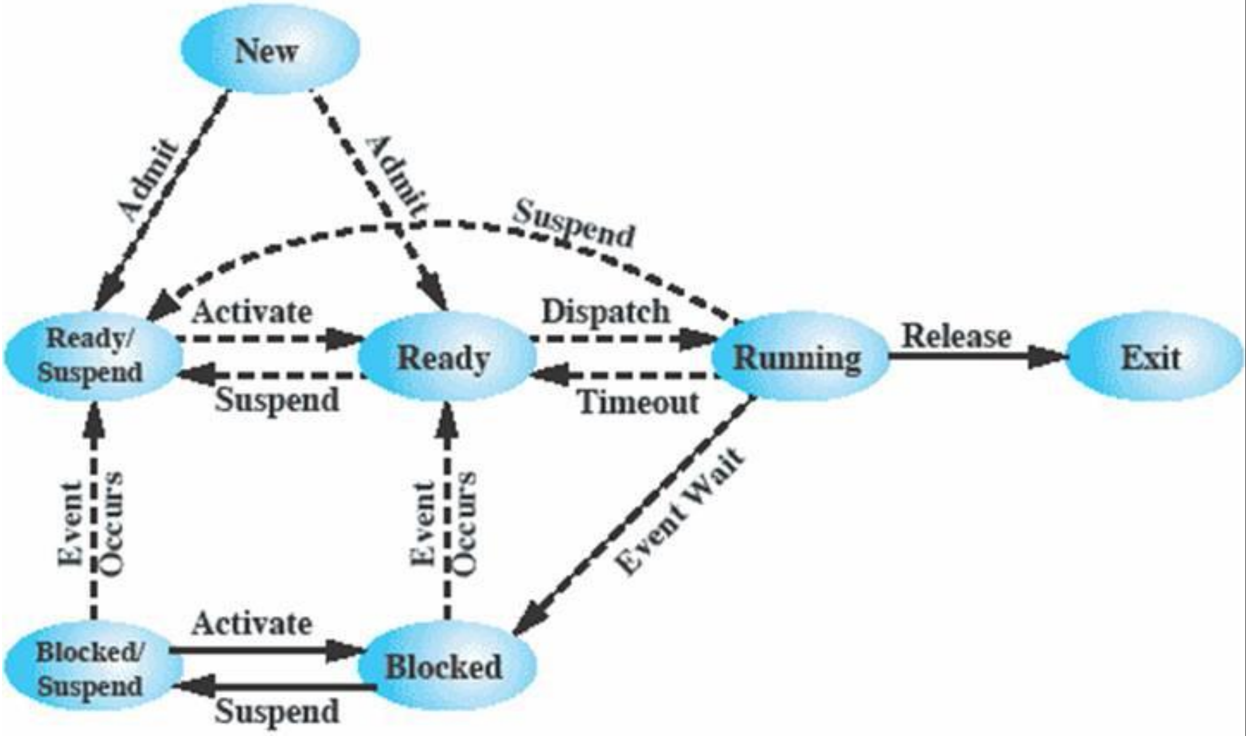
3) **Waiting:** When a Process is Waiting for Some Input and Output Operations then this is called as the Waiting State. And in this process is not under the Execution instead the Process is Stored out of Memory and when the user will provide the input then this will Again be on ready State.

4) **Ready State:** When the Process is Ready to Execute but he is waiting for the CPU to Execute then this is called as the Ready State. After the Completion of the Input and outputs the Process will be on Ready State means the Process will Wait for the Processor to Execute.

Terminated State: After the Completion of the Process , the Process will be Automatically terminated by the CPU . So this is also called as the Terminated State of the Process. After executing the whole Process the Processor will also de-allocate the Memory which is allocated to the Process. So this is called as the Terminated Process.



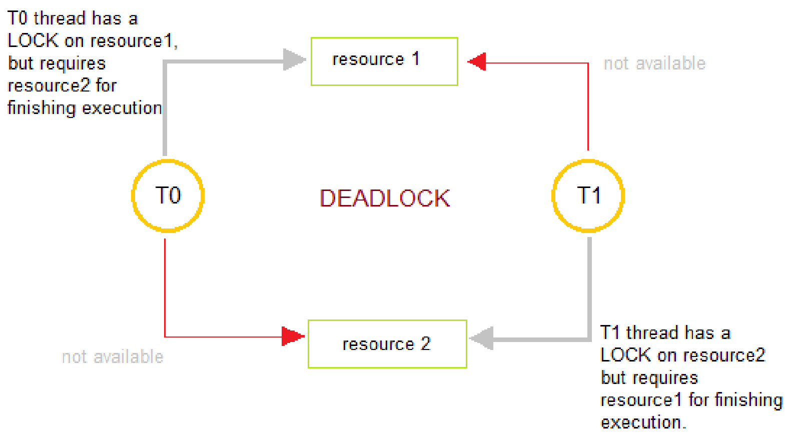
For Suspend process



With two suspend states - Process State Transition Diagram with Suspend States

3 (a) What is deadlock? Explain with a small diagram. What are the necessary conditions for the deadlock to occur in a system — 6

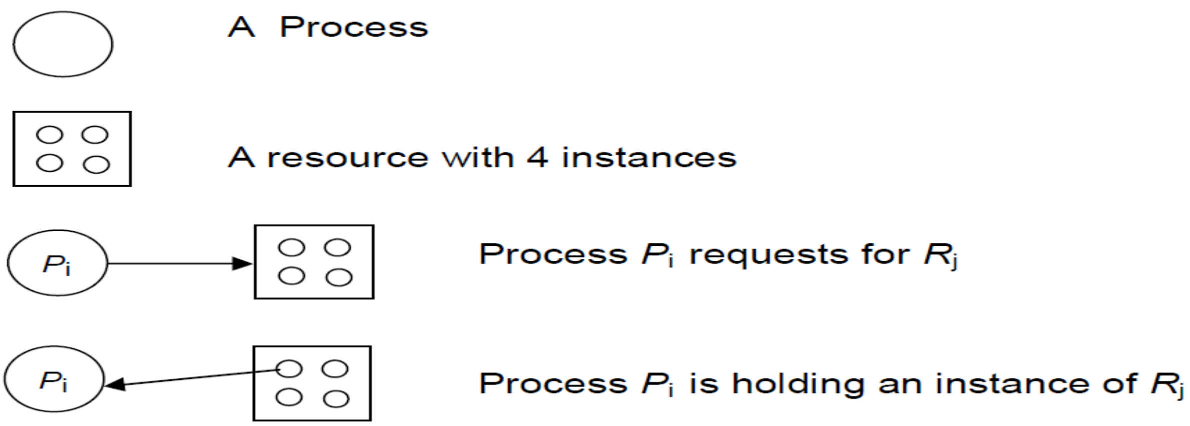
Deadlock can be defined as the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can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

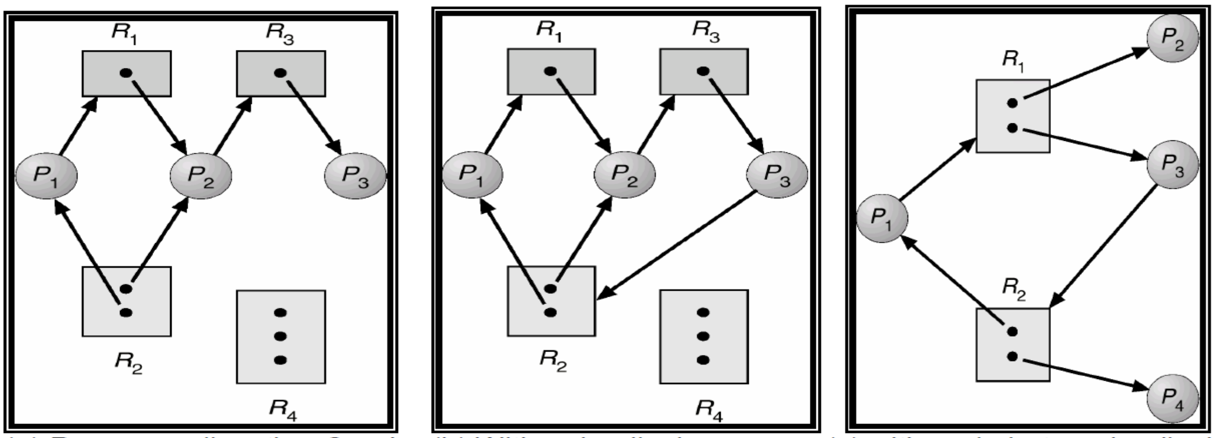


Deadlock can arise if four conditions hold simultaneously:

- 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, …, *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.	
(b)	Write short notes about resource allocation graph 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:  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.	4

<table>
<tr><td></td><td>


(a) Resource allocation Graph (b) With a deadlock (c) with cycle but no deadlock
Figure 5.1 Resource allocation graphs
</td><td></td></tr>
<tr><td>4</td><td>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.</td><td>10</td></tr>
</table>

```
/* 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);
}
```

5 (a) Define is response time and turnaround time? 3

Response Time: The time duration from the submission of a request till the first response received is known as response time.

 Turnaround time: The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

(b) Consider the following set of processes with given length of CPU burst: 7

Processes	P1	P2	P3	P4	P5
Burst Time	6	2	8	3	4
Arrival Time	2	5	1	0	4

Draw Gantt chart for SJF (preemptive) and SFJ (non-preemptive.
Find the average waiting time, average turn around time, throughput for each scheduling algorithm.

SJF (Non - Preemptive)

P4	P1	P2	P5	P3

0 3 9 11 15 23

AWT = [(0-0)+(3-2)+(9-5)+(11-4)+(15-1)] / 5
= (0+1+4+7+14)/5
=26/5 = 5.2ms

Average Turnaround Time = [(9-2)+(11-5)+(23-1)+(3-0)+(15-4)]/5
= (7+6+22+3+11)/5
= 49/5=9.8 ms

Through put : 5/23 = 0.21 ms

SJF (Preemptive)

P4	P1	P5	P2	P5	P1	P3

0 3 4 5 7 10 15 23

Waiting Time for P1 = 3-2+6=7
Waiting Time for P2 = 5-5 = 0
Waiting Time for P3 = 15-1=14
Waiting Time for P4 = 0-0=0
Waiting Time for P5 = 4-4+2=2
AWT = (7+0+14+0+2)//5 = 23/5 = 4.6 ms

Average Turnaround Time : (15-2) + (7-5) + (23-1) + (3-0) + (10-4) / 5
= (13+2+22+3+6) /5
= 46/2 = 9.32 ms

Through put : 5/23 = 0.21 ms

6 (a) Define is waiting time, throughput? 3

Throughput: The number of processes completed per time unit is called as throughput.

 Waiting Time: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

(b) Consider the following set of processes with given length of CPU burst: 7

Processes	P1	P2	P3	P4	P5
Burst Time	10	1	2	1	5
Priority	3	1	3	4	2

All processes arrived at time 0 in the given order.
Draw Gantt chart using SJF (non-preemptive), Priority (Non-preemptive) [Smallest number implies highest priority], and Round Robin [Quantum-2 ms] scheduling policies. Find the average waiting time for each scheduling policy.

SJF (Non preemptive)

P2	P4	P3	P5	P1

0 1 2 4 9 19

AWT = 9+0+2+1+4 = 16/5 = 3.2 ms

Priority (Non-preemptive)

P2	P5	P1	P3	P4

0 1 6 16 18 19

AWT = 6+0+16+18+1 = 41/5 = 8.2 ms

Round Robin

P1	P2	P3	P4	P5	P1	P5	P1	P5	P1	P1

0 2 3 5 6 8 10 12 14 15 17 19

Waiting Time for P1 = 0+(8-2)+(12-10)+(15-14) = 0+6+2+1=9
Waiting Time for P2 = 2
Waiting Time for P3 = 3
Waiting Time for P4 = 5
Waiting Time for P5 = 6+(10-8)+(14-12) = 6+2+2=10
AWT = 9+2+3+5+10 = 29/5 = 5.8 ms

7 (a) What is a thread? Explain the benefits of a multithreaded programming — 4

A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. Following figure shows the working of the single and multithreaded processes.

The benefits of threads are:

- Thread takes less time to create compared to a process
- It takes less time to terminate compared to a process
- Switching between two threads takes less time than switching processes
- Threads can communicate with each other without invoking the kernel

(b) Explain with diagram user level threads and kernel level threads — 6

User – level and Kernel – level Threads

A thread can be implemented as either a user – level thread (ULT) or kernel – level thread (KLT). The KLT is also known as *kernel – supported threads* or *lightweight processes.*

User – level Threads: In ULT, all work of thread management is done by the application and the kernel is not aware of the existence of threads. It is shown in Figure 3.12 (a).

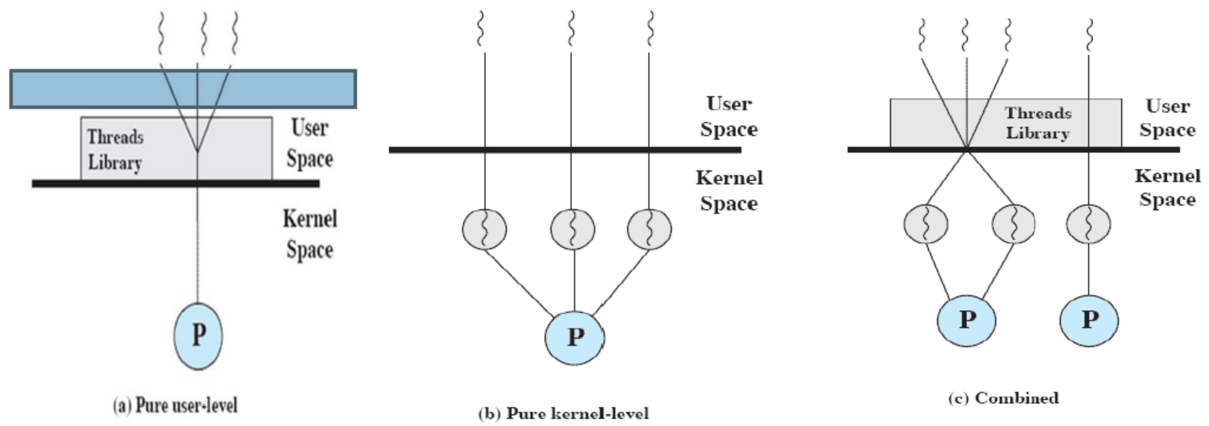
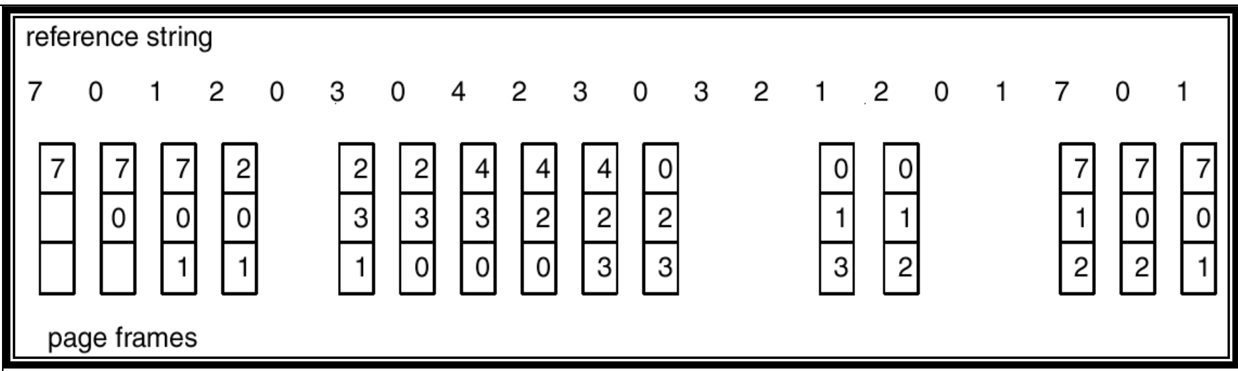


Figure 3.12 ULT and KLT

Any application can be programmed to be multithreaded by using a threads library, which a package

	of routines for ULT management. Usually, an application begins with a single thread and begins running in that thread. This application and its thread are allocated to a single process managed by the kernel. The application may spawn a new thread within the same process during its execution. But, kernel is not aware of this activity. The advantages of ULT compared to KLT are given below: • Thread switching doesn't require kernel mode privileges. Hence, the overhead of two switches (user to kernel and kernel back to user) is saved. • Scheduling can be application specific. So, OS scheduling need not be disturbed. • ULTs can run on any OS. So, no change in kernel design is required to support ULTs. There are certain disadvantages of ULTs compared to KLTs: • Usually, in OS many system calls are blocking. So, when a ULT executes a system call, all the threads within the process are blocked. • In a pure ULT, a multithreaded application cannot take advantage of multiprocessing. Kernel – level Threads: In pure KLT model, all work of thread management is done by the kernel. Thread management code will not be in the application level. This model is shown in Figure 3.12(b). The kernel maintains context information for the process as a whole and for individual threads within the process. So, there are certain advantages of KLT : • The kernel can simultaneously schedule multiple threads from the same process on multiple processors. • If one thread in a process is blocked, the kernel can schedule another thread of the same process. • Kernel routines themselves can be multithreaded. But, there is a disadvantage as well: The transfer of control from one thread to another within the same process requires a mode switch to the kernel. Combined Approach: Some OS provide a combination of ULT and KLT as shown in Figure 3.12 (c). In this model, thread creation is done completely in user space. The multiple ULTs from a single application are mapped onto number of KLTs. The programmer may adjust the number of KLTs for a particular application and processor to achieve the best results.	
8	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. 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.	10

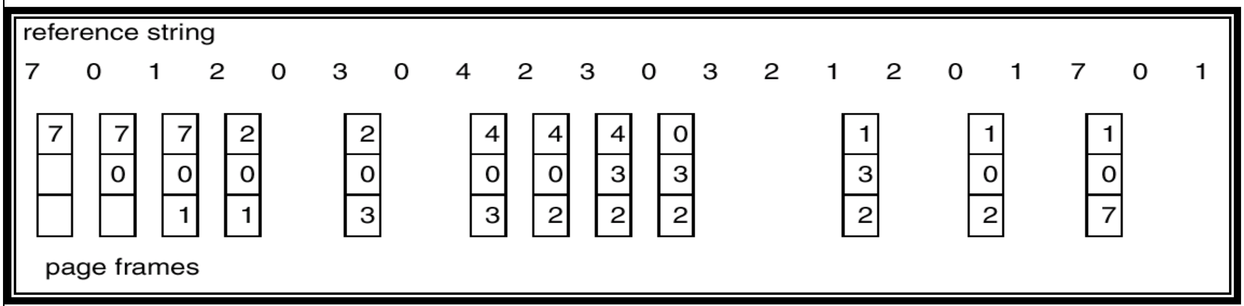


In the above example, there are 15 page faults.

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.

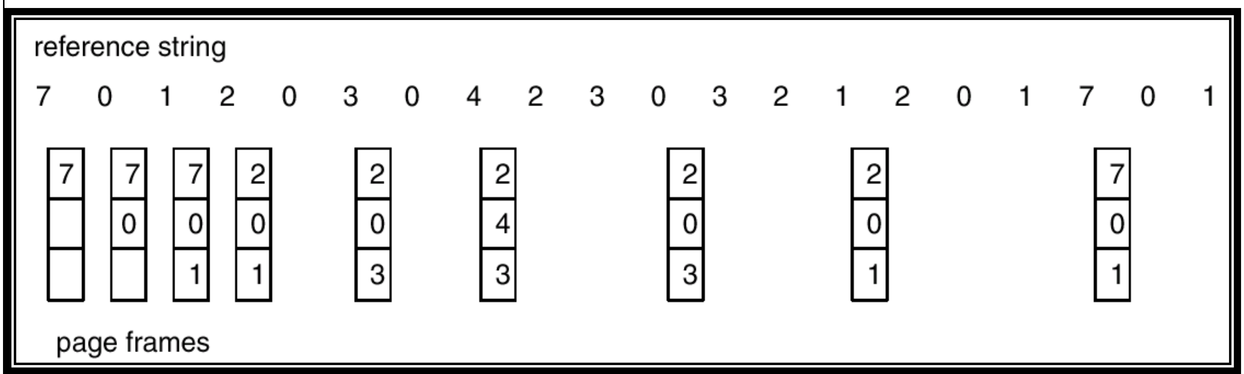
Example:



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.

9 What is demand paging? Explain how TLB improves the performance of demand paging with neat diagram 10

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.

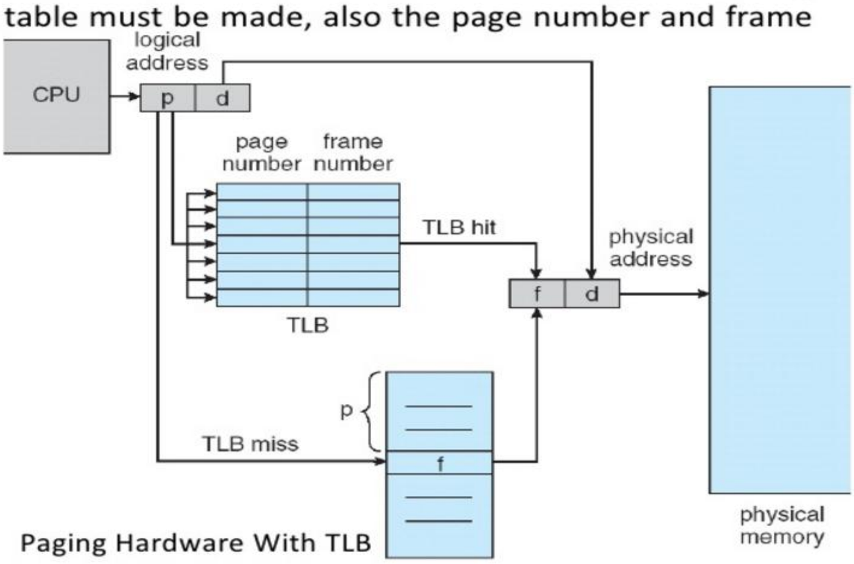
<u>Demand Paging</u>: Bring a page into memory only when it is needed.

-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.

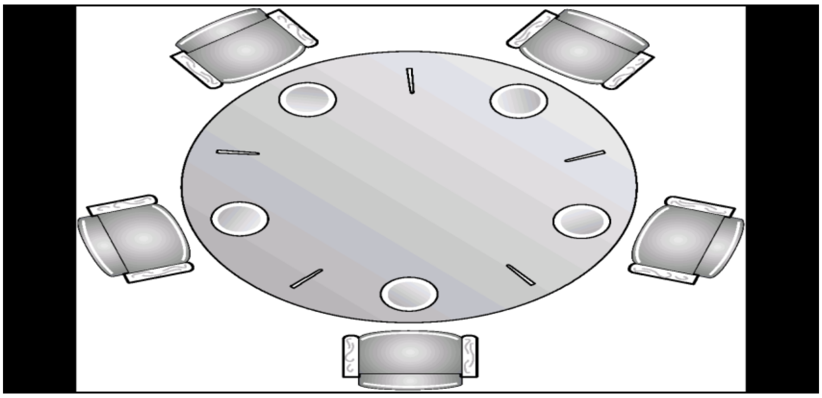


Paging Hardware With TLB

10. State the dining philosophers problem and give solution for the same, using semaphores — 10

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

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.

```
/* 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