FIRST TERM EXAMINATION [FEB. 2016]
SIXTH SEMESTER [B.TECH]
OPERATING SYSTEM [ETCS–304]

Time : 1.30 Hrs. **M.M. : 30**

Note: *Attempt Q no. 1 which is compulsory and any two more questions.*

Q.1(*a*) Why page size is always represented as power of 2?

Ans : It is most efficient to break the address into X **page** bits and Y offset bits, rather than perform arithmetic on the address to calculate the **page** number and offset. Because each bit position **represents** a **power of 2**, splitting an address between bits results in a **page size** that is a **power of 2.**

Q.1(*b*) What is multiprogramming operating system? Explain.

Ans: In a multiprogramming system there are one or more programs loaded in main memory which are ready to execute. Only one program at a time is able to get the CPU for executing its instructions (i.e., there is at most one process running on the system) while all the others are waiting their turn.

The main idea of multiprogramming is to maximize the use of CPU time. Indeed, suppose the currently running process is performing an I/O task (which, by definition, does not need the CPU to be accomplished). Then, the OS may interrupt that process and give the control to one of the other in-main-memory programs that are ready to execute (i.e. *process context switching*). In this way, no CPU time is wasted by the system waiting for the I/O task to be completed, and a running process keeps executing until either it voluntarily releases the CPU or when it blocks for an I/O operation. Therefore, the ultimate goal of multiprogramming is to keep the CPU busy as long as there are processes ready to execute.

Note that in order for such a system to function properly, the OS must be able to load multiple programs into separate areas of the main memory and provide the required protection to avoid the chance of one process being modified by another one. Other problems that need to be addressed when having multiple programs in memory is *fragmentation* as programs enter or leave the main memory. Another issue that needs to be handled as well is that large programs may not fit at once in memory which can be solved by using *paging* and *virtual memory*.

Finally, note that if there are N ready processes and all of those are highly CPU-bound (i.e., they mostly execute CPU tasks and none or very few I/O operations), in the very worst case one program might wait all the other N-1 ones to complete before executing.

Q.1.(*c*) Explain Process Control Block in brief?

Ans: A process in an operating system is represented by a data structure known as a **process control block (PCB) or process descriptor.** The PCB contains important information about the specific process including

- The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- Unique identification of the process in order to track "which is which" information.
- A pointer to parent process.
- Similarly, a pointer to child process (if it exists).

- The priority of process (a part of CPU scheduling information).
- Pointers to locate memory of processes.
- A register save area.
- The processor it is running on.

Q.1.(*d*) What is starvation? What are the possible solutions to solve t problem?

Ans: In computer science, **starvation** is a problem encountered in concurre computing where a process is perpetually denied necessary resources to process its wo Starvation may be caused by errors in a scheduling or mutual exclusion algorithm, i can also be caused by resource leaks, and can be intentionally caused via a denial- service attack such as a fork bomb.

Starvation is similar to deadlock in that it causes a process to freeze. Two or m processes become deadlocked when each of them is doing nothing while waiting f resource occupied by another program in the same set. On the other hand, a process i starvation when it is waiting for a resource that is continuously given to other proces Starvation-freedom is a stronger guarantee than the absence of deadlock: a mu exclusion algorithm that must choose to let one of two processes into a critical section picks one arbitrarily is deadlock-free, but not starvation-free.

Possible solution: A possible solution to starvation is to use a scheduling algor with priority queue that also uses the **aging technique**. Aging is a technique of grad increasing the priority of processes that wait in the system for a long time.

Q.1.(*e*) Explain race condition with suitable example?

Ans: Race conditions are most commonly associated with computer scienc computer memory or storage, a race condition may occur if commands to read and a large amount of data are received at almost the same instant, and the ma attempts to overwrite some or all of the old data while that old data is still being The result may be one or more of the following: a computer crash, an "illegal opera notification and shutdown of the program, errors reading the old data or errors wr the new data. A race condition can also occur if instructions are processed in the inco order.

Example: Suppose for a moment that two processes need to perform a bit flip specific memory location. Under normal circumstances the operation should work this:

Process 1	Process 2	Memory Value
Read value		0
Flip value		1
	Read value	1
	Flip value	0

In this example, Process 1 performs a bit flip, changing the memory value from 1. Process 2 then performs a bit flip and changes the memory value from 1 to 0. If a condition occurred causing these two processes to overlap, the sequence could potenti look more like this:

Process 1	Process 2	Memory Value
Read value		0
	Read value	0
Flip value		1
	Flip value	1

In this example, the bit has an ending value of 1 when its value should be 0. This occurs because Process 2 is unaware that Process 1 is performing a simultaneous bit flip.

Q.2.(a) Explain the layered approach of operating system? What are main task of OS.

Ans: Layered Approach: With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS or UNIX systems. The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under the top-down approach, the overall functionality and features are determined and the separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

A system can be made modular in many ways. One method is the **layered approach,** in which the operating system is broken up into a number of layers (levels). The bottom layer (layer 0) id the hardware; the highest (layer N) is the user interface.

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating – system layer-say, layer M-consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers.

The **main advantage** of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified.

Each layer is implemented with only those operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers. The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.

Other requirement may not be so obvious. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. However, on a larger system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.

A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-

management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified; data may need to be passed, and so on. Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a non-layered system. These limitations have caused a small backlash against layering in recent years. Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of layer definition and interaction.

Main tasks of Operating System:

There are several tasks which are performed by practically all operating systems regardless of the complexity of the computer the operating system is being used on.

These tasks include :

- Managing communications between software and hardware.
- Allocation of computer memory.
- Allocation of CPU time.
- Organising data on backing storage devices.

Managing Communications between Software and Hardware

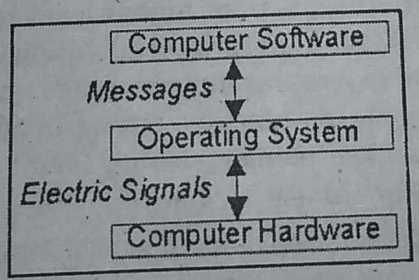
There are many tasks which practically every program written for a computer will need to carry out. For example :

- Loading and saving data and programs.
- Communicating with peripherals such as a mouse or a printer.
- Displaying information on the VDU.

There has to be some control over these activities to make sure that problems do not occur. Consider what might happen if a computer program was trying to load a file from a floppy disk and, at the same time, another program was trying to save data.

Therefore the operating system manages all of these tasks. If, for example, a program wants to save or print a file it must send a request to the operating system asking it to do so. The operating system will then carry out the task.

Because an application such as a spreadsheet works by communicating with the operating system the application will probably only work with one particular operating system. If you buy Microsoft Works for Windows 2000 it will only operate on computers with the Windows 2000 operating system. If you want to use Microsoft Works with the MS-DOS operating system you will have to buy a different version of the application.

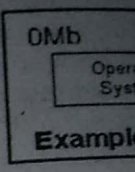


Putting commonly used operations such as saving data into the operating system also reduces the amount of work that a person writing a new application must do.

Allocation of Computer Memory

Computer memory must be allocated to (divided between) the different tasks that the computer is performing. Even if your computer appears to be doing only one task such as word processing there will be several different items to which memory must be allocated. For example memory would have to be allocated to storing the word

processor program, storing y
image that is being displayed
will also use up some memo



Allocation of CPU Time

Most computers have
(CPU). Therefore the CPU
must be divided up betwee
If you were printing a docu
would need to be divided u

- Letting you type on
- Sending the docume
- Updating the screen

Chunks of the CPU's
Time slices are normally
same time.

Organising Data on Ba

Work and programs
kept when a computer is
data on backing storage
organised so that they ca
a **filename**.

Most operating syst
known as **folders**). File

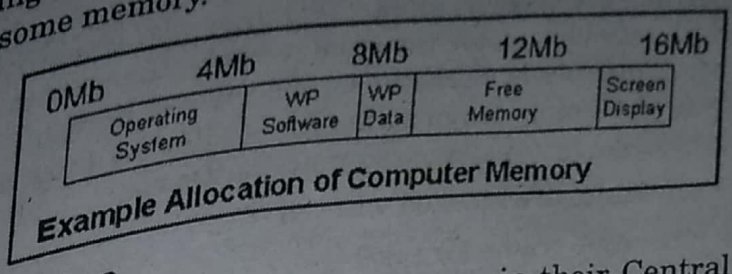
Q.2.(*b*) On a simpl
page entries & full pa
satisfied by the assoc
memory page table t
effective access time

Ans: Effective Acce

Where m_1 represe
reference through main
ratio.

So,

processor program, storing your word processed document and storing a copy of the image that is being displayed on the monitor. And of course the operating system itself will also use up some memory.



Example Allocation of Computer Memory

Allocation of CPU Time

Most computers have only one microprocessor in their Central Processing Unit (CPU). Therefore the CPU can only process one piece of data at a time. The CPU's time must be divided up between all of the tasks that the computer is currently carrying out. If you were printing a document form a word processor application then the CPU's time would need to be divided up between:

- Letting you type on the computer.
- Sending the document that is being printed to the printer.
- Updating the screen display to show you what is going on.

Chunks of the CPU's time known as **time slices** are allocated to each task in turn. Time slices are normally very **short** so that each task can appear to be running at the same time.

Organising Data on Backing Storage Devices

Work and programs must be saved on a backing storage device so that they can be kept when a computer is turned off. The operating system is responsible for organising data on backing storage devices. Work and programs are saved as **files** which must be organised so that they can be found and loaded when required. Each file is identified by a **filename**.

Most operating systems divide a backing storage device up into **directories** (also known as **folders**). Files can be stored within each directory.

Q.2.(b) On a simple paged system, associative registers hold the most active page entries & full page table is stored in the main memory. If the references satisfied by the associative registers take 100ns & reference through the main memory page table take 180ns. What must be the hit ratio to be achieved on effective access time of 125ns?

Ans: Effective Access Time= $[x*(m_1+m_2)] + [(1-x)*(c+m_1+m_2)]$

Where m_1 represents references satisfied by associative registers, m_2 represents reference through main memory, c represents access the page table, x represents the hit ratio.

So,

$$EAT = [x*(100+180)]+[(1-x)*(180+100+180)]$$
$$125 = (280x)+(460-460x)$$
$$125 = 460-180x$$
$$180x = 460-125$$
$$180x = 335$$
$$x = 1.86\%(\text{Hit Ratio})$$

Q.2.(c) What is Belady's Anomaly? Explain with the help of suitable examples.

Ans: Belady's Anomaly is also called FIFO anomaly. Usually, on increasing the number of frames allocated to a process virtual memory, the process execution is faster because fewer page faults occur. Sometimes, the reverse happens, i.e., the execution time increases even when more frames are allocated to the process. This is Belady's **Anomaly**. This is true for certain page reference patterns.

Example : Page faults are marked with an "f".

1: In this case Assume frame size is 3

Page Requests	3	2	1	0	3	2	4	3	2	1	0
Newest Page	3f	2f	1f	0f	3f	2f	4f	4	4	1f	0f
		3	2	1	0	3	2	2	2	4	1
Oldest Page			3	2	1	0	3	3	3	2	4

2: In this case Assume frame size is 4

Page Requests	3	2	1	0	3	2	4	3	2	1	0
Newest Page	3f	2f	1f	0f	0	0	4f	3f	2f	1f	0f
		3	2	1	1	1	0	4	3	2	1
			3	2	2	2	1	0	4	3	2
Oldest Page				3	3	3	2	1	0	4	3

In the first example (with fewer pages), there are 9 page faults.

In the second example (with more pages), there are 10 page faults.

Q.3.(a) What are various scheduling criteria?

Ans: There are many different criteria's to check when considering the "best" scheduling algorithm :

1. CPU utilization: To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded).

2. Throughput: It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

3. Turnaround time: It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

4. Waiting time: The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

5. Load average: It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

6. Response time: Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

I.P. U

Q.3.(b) Consider the fol time in milliseconds?

Process
A
B
C
D
E

Draw the Gantt cha
(i) Average waiting Time First, Round Robi
(ii) Average turnarou Time First, Round Robi
Ans: Gantt Chart f

For FCFS:

Process	AT
A	0
B	1
C	2
D	3
E	4

(i) Average Waitin
(ii) Average Turna
Gantt Chart for SRT

For SRTF:

Process	AT
A	0
B	1
C	2
D	3
E	4

(i) Average Wai
(ii) Average Tu

Q.3.(b) Consider the following set of processes with the CPU burst & arrival me in milliseconds?

Process	Arrival Time	Burst Time
A	0	10
B	1	5
C	2	8
D	3	4
E	4	11

Draw the Gantt chart & find :

(i) Average waiting time for these processes with the shortest Remaining Time First, Round Robin (Time quantum=3ms) & FCFS scheduling algorithms.

(ii) Average turnaround time for these processes with the shortest Remaining Time First, Round Robin (Time quantum=3ms) & FCFS scheduling algorithms.

Ans: Gantt Chart for FCFS

A	B	C	D	E

0 10 15 23 27 38

For FCFS:

Process	AT	BT	CT	TAT=(CT-AT)	WT=(TAT-BT)
A	0	10	10	10	0
B	1	5	15	14	9
C	2	8	23	21	13
D	3	4	27	24	20
E	4	11	38	34	23

(i) Average Waiting Time=(0+9+13+20+23)/5=65/5=**13ms**

(ii) Average Turnaround Time=(10+14+21+24+34)/5=103/5=**20.6ms**

Gantt Chart for SRTF

A	B	D	C	A	E

0 1 6 10 18 27 38

For SRTF:

Process	AT	BT	CT	TAT=(CT-AT)	WT=(TAT-BT)
A	0	10	27	27	17
B	1	5	6	5	0
C	2	8	18	16	8
D	3	4	10	7	3
E	4	11	38	34	23

(i) Average Waiting Time=(17+0+8+3+23)/5=51/5=**10.2ms**

(ii) Average Turnaround Time=(27+5+16+7+34)/5=89/5=**17.8ms**

Gantt Chart for RR(Time quantum 3ms)

A	B	C	D	A	E	B	C	D	A	E	A	C	E

0 3 6 9 12 15 18 20 23 24 27 30 31 33 38

For RR(Time quantum 3ms)

Process	AT	BT	CT	TAT=(CT-AT)	WT=(TAT-BT)
A	0	10	31	31	21
B	1	5	20	19	14
C	2	8	33	31	23
D	3	4	24	21	17
E	4	11	38	34	23

(i) Average Waiting Time=(21+14+23+17+23)/5=98/5=**19.6ms**

(ii) Average Turnaround Time=(31+19+31+21+34)/5=136/5=**27.2ms**

Q.4. (*a*) Explain bakery algorithm. Prove that it satisfies all the three requirements for critical section problem.

Ans: Bakery algorithm: The Bakery algorithm is one of the simplest known solutions to the mutual exclusion problem for the general case of N process. The basic idea is that each non-thinking process has a variable that indicates the position of the process in a hypothetical queue of all the non-thinking processes. Each process in the queue scans the variables of the other processes, and enters the critical section only upon determining that it is at the head of the queue.

But the resulting algorithm is still not easy to understand. So in this note we first look at a simplified version of the bakery algorithm, based a coarser grain of atomicity than is allowed by the mutual exclusion problem.

Solution to the Critical Section Problem must meet three conditions...

1. Mutual exclusion: If process is executing in its critical section, no other process is executing in its critical section

2. Progress: If no process is executing in its critical section and there exists some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this decision cannot be postponed indefinitely

If no process is in critical section, can decide quickly who enters

Only one process can enter the critical section so in practice, others are put on the queue

3. Bounded waiting: There must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

The wait is the time from when a process makes a request to enter its critical section until that request is granted

In practice, once a process enters its critical section, it does not get another turn until a waiting process gets a turn (managed as a queue).

The three requirements for critical section problem

Suppose that Pi is in its critical section, and Pk(k!=i) has already chosen its number[k], there are two cases:

1. Pk has already chosen its n
critical section.
In this case, if (number[i], i
equal, so (number[i],i) > (number
critical section before Pk does, a
condition does not hold, which is
Note that if Pk get a new numb
2. Pk did not chose its numb
section.
In this case, since Pk has
must chose its number later tha
number than Pi, so (number[i]

Q.4.(*b*) Define a monito

Ans: In concurrent progr
allows threads to have both m
condition to become true. Mo
that their condition has b
and **condition variables**.
that are waiting for a certai
temporarily give up exclusiv
regaining exclusive access

Another definition of m
wrapped mutual exclusion
more than one thread. The
executed with mutual exc
executing any of its meth
ability for threads to wai
"monitor"). For the rest
"thread-safe object/class

Q.4.(c) Explain the

Ans: The ***Dining P***
(E. W. Dijkstra. Co-oper

Dining Philosoph
circular table with five
between each plate is
table is a bowl of spagh
who spend most of thei
hungry and need to ea

In order to eat, a
the two chopsticks to
eat the spaghetti on

Thus, each philo
process P[i]
while true
{ THINK
PICKU
EAT;
PUTD
}

1. Pk has already chosen its number when Pi does the last test before entering its critical section.

In this case, if (number([i], i) < (number[k],k) does not hold, since they cannot be equal, so (number[i],i) > (number[k],k). Suppose this is true, then Pi cannot get into the cirtical section before Pk does, and Pi will looping at the last while statement until the condition does not hold, which is modified by Pk when it exits from its critical section. Note that if Pk get a new number again, it must be larger than that of Pi's.

2. Pk did not chose its number when Pi does the last test before entering its critical section.

In this case, since Pk has chosen its number when Pi is in its critical section, Pk must chose its number later than Pi. According to the algorithm, Pk can only get a bigger number than Pi, so (number[i],i) < number([k],k) holds.

Q.4.(b) Define a monitor?

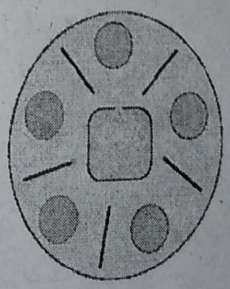
Ans: In concurrent programming, a **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signalling other threads that their condition has been met. A monitor consists of a mutex (lock) object and **condition variables**. A **condition variable** is basically a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

Another definition of **monitor** is a **thread-safe** class, object, or module that uses wrapped mutual exclusion in order to safely allow access to a method or variable by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: At each point in time, at most one thread may be executing any of its methods. Using a condition variable(s), it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor"). For the rest of this article, this sense of "monitor" will be referred to as a "thread-safe object/class/module".

Q.4.(c) Explain the Dining Philosophers Problem?

Ans: The ***Dining Philosophers*** problems are a classic synchronization problem (E. W. Dijkstra. Co-operating Sequential Processes).

Dining Philosophers: There is a dining room containing a circular table with five chairs. At each chair is a plate, and between each plate is a single chopstick. In the middle of the table is a bowl of spaghetti. Near the room are five philosophers who spend most of their time thinking, but who occasionally get hungry and need to eat so they can think some more.



In order to eat, a philosopher must sit at the table, pick up the two chopsticks to the left and right of a plate, then serve and eat the spaghetti on the plate.

Thus, each philosopher is represented by the following pseudocode:

```
process P[i]
while true do
{THINK;
 PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
 EAT;
 PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
}
```

A philosopher may THINK indefinately. Every philosopher who EATs will ev
finish. Philosophers may PICKUP and PUTDOWN their chopsticks in either
non deterministically, but these are atomic actions, and, of course, two philo
cannot use a single CHOPSTICK at the same time.

The problem is to design a protocol to satisfy the liveness condition: *any phil*
who tries to EAT, eventually does.

Discussion. Of course, the best thing is to go off and try to solve this prob
your own by exploring various protocols philosophers might use to acquire cho
You are likely to quickly encounter the same *deadlock* and *livelock* scenarios we
the *mutual exclusion* problem, but you will quickly see in this case that mutual ex
is too primitive a synchronization mechanism for solving this problem.

Similar solutions are found in most operating systems textbooks. All of the
renditions of the original treatment by Dijkstra, which motivated the *sema*
mechanism he was introducing in his original article. A\emph{semaphore} is an i
or boolean value, S, with two associated atomic operations:

DOWN(S) *wait* until $S > 0$, then decrement S;

UP(S) increment S

In time-sharing systems, "waiting" is implemented by the operating system,
may put processes on a wait-list for later execution. In hardware, "waiting" m
accomplished by busy-waiting or by some form of explicit signaling, such as
passing.

FIRST TERM EXAMINATION [FEB. 2017]
SIXTH SEMESTER [B.TECH]
OPERATING SYSTEMS [ETCS–304]

Time : 1.5 Hrs. M.M. : 30

Note: Attempt any three questions including Q. No. 1 which is compulsory.

Q.1. (a) What is Convoy effect. Explain with the help of suitable examples. **(2.5)**

Ans. Convoy Effect is phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to few slow processes.

FCFS algorithm is non-preemptive in nature, that is, once CPU time has been allocated to a process, other processes can get CPU time only after the current process has finished. This property of FCFS scheduling leads to the situation called Convoy Effect.

Suppose there is one CPU intensive (large burst time) process in the ready queue, and several other processes with relatively less burst times but are Input/Output (I/O) bound (Need I/O operations frequently).

The following then takes place –

- The I/O bound processes are first allocated CPU time. As they are less CPU intensive, the quickly get executed and then goto I/O queues.
- Now, the CPU intensive process is allocated CPU time. As its burst time is high, it takes time to complete.
- While the CPU intensive process is being executed, the I/O bound processes complete their I/O operations and are moved back to ready queue.
- However, the I/O bound processes are made to wait as the CPU intensive process still hasn't finished. This leads to I/O devices being idle.
- When the CPU intensive process gets over, it is sent to the I/O queue so that it can access and I/O device.
- Meanwhile, the I/O bound processes get their required CPU time and move back to I/O queue.
- However, they are made to wait because the CPU intensive process is still accessing an I/O device. As a result, the CPU is sitting idle now.

Hence in Convoy Effect, one slow process slows down the performance of the entire set of processes, and leads to wastage of CPU time and other devices.

To avoid Convoy Effect, preemptive scheduling algorithms like Round Robin Scheduling can be used – as the smaller processes don't have to wait much for CPU time – making their execution faster and leading to less resources sitting idle.

Q.1.(b)What is batch operating system. Explain **(2.5)**

Ans. The main function of a batch processing system is to automatically keep executing the jobs in a batch. This is the important task of a batch processing system i.e. performed by the 'Batch Monitor' resided in the low end of main memory.

This technique was possible due to the invention of hard-disk drives and card readers. Now the jobs could be stored on the disk to create the pool of jobs for its execution as a batch. First the pooled jobs are read and executed by the batch monitor, and then these jobs are grouped; placing the identical jobs (jobs with the similar needs) in the same batch, So, in the batch processing system, the batched jobs were executed automatically one after another saving its time by performing the activities (like loading

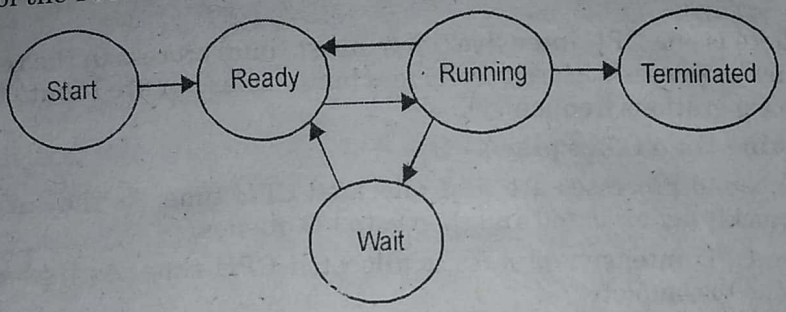
of compiler) only for once. It resulted in improved system utilization due to reduced turn around time.

In the early job processing systems, the jobs were placed in a job queue and when space was available in the main memory, a job was selected from the job queue and was loaded into memory. Once the job loaded into primary memory, it competes for the processor. When the processor became available, the processor scheduler selects job that was loaded in the memory and execute it.

In batch , strategy is implemented to provide a batch file processing. So in this approach files of the similar batch are processed to speed up the task.

Q.1. (c) Explain various states of a process. **(2.5)**

Ans. A process which is executed by the process have various states, the state of the process is also called as the **states of the process**, The states includes whether the process has executed or whether the process is waiting for some input and output from the user and whether the Process is Waiting for the CPU to Run the Program after the Completion of the Process. 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.

(5) 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 deallocate the Memory which is allocated to the Process. So this is called as the Terminated Process.

As we know that there are many processes those are running at a Time, this is not true. **A processor can execute only one Process at a Time**. There are the various States of the Processes those determined which Process will be executed. The Processor will Execute all the processes by using the States of the Processes, the Processes those are on the Waiting State will not be executed and CPU will Also divides his time for Execution if there are Many Processes those are Ready to Execute.

When a Process Change his State from one State to Another, then this is also called as the **Process State Transition**. In this, a running process may goes on Wait and a ready Process may goes on the Wait State and the Wait State can be goes on the Running State.

Q.1. (d) What is Belady's Anomaly. **(2.5)**

Ans. In computer storage, **Belady's anomaly** proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. Laszlo Belady demonstrated this in 1969.

In common computer memory management, information is loaded in specific sized chunks. Each chunk is referred to as *a page*. The central processor can only load a limited number of pages at a time. It requires a *frame* for each page it can *load*. *A page fault* occurs when a page is not found, and might need to be loaded from disk into memory.

When a page fault occurs and all frames are in use, one must be cleared to make room for the new page. A simple algorithm is FIFO. Whichever page has been in the frames the longest is the one that is cleared. Until Belady's anomaly was demonstrated, it was believed that an increase in the number of page frames would always provide the same number or fewer page faults.

It is also called FIFO anomaly. Usually, on increasing the number of frames allocated to a process virtual memory, the process execution is faster, because fewer page faults occur. Sometimes, the reverse happens, i.e., the execution time increases even when more frames are allocated to the process. This is Belady's Anomaly. This is true for certain page reference patterns.

Q.2. (a) Differentiate between Multi Programming,Multi Processing and Multi Tasking OS. **(5)**

Ans. Multiprogramming: In a multiprogramming system there are one or more programs loaded in main memory which are ready to execute. Only one program at a time is able to get the CPU for executing its instructions (i.e., there is at most one process running on the system) while all the others are waiting their turn.

The main idea of multiprogramming is to maximize the use of CPU time. Indeed, suppose the currently running process is performing an I/O task (which, by definition, does not need the CPU to be accomplished). Then, the OS may interrupt that process and give the control to one of the other in-main-memory programs that are ready to execute (i.e. *process context switching*). In this way, no CPU time is wasted by the system waiting for the I/O task to be completed, and a running process keeps executing until either it voluntarily releases the CPU or when it blocks for an I/O operation. Therefore, the ultimate goal of multiprogramming is to keep the CPU busy as long as there are processes ready to execute.

Note that in order for such a system to function properly, the OS must be able to load multiple programs into separate areas of the main memory and provide the required protection to avoid the chance of one process being modified by another one. Other problems that need to be addressed when having multiple programs in memory is *fragmentation* as programs enter or leave the main memory. Another issue that needs to be handled as well is that large programs may not fit at once in memory which can be solved by using *paging* and *virtual memory*.

Multiprocessing: Multiprocessing sometimes refers to executing multiple processes (programs) at the same time. In fact, multiprocessing refers to the *hardware* (i.e., the CPU units) rather than the *software* (i.e., running processes). If the underlying hardware provides more than one processor then that is multiprocessing. Several

variations on the basic scheme exist, e.g., multiple cores on one die or multiple dies in one package or multiple packages in one system.

Anyway, a system can be both multiprogrammed by having multiple programs running at the same time and multiprocessing by having more than one physical processor.

Multitasking: Multitasking has the same meaning of multiprogramming but in a more general sense, as it refers to having multiple (programs, processes, tasks, threads) running at the same time. This term is used in modern operating systems when multiple tasks share a common processing resource (e.g., CPU and Memory). At any time the CPU is executing one task only while other tasks waiting their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task (i.e. *processor thread context switching*).

A *task* in a multitasking operating system is not a whole application program but it can also refer to a "thread of execution" when one process is divided into sub-tasks. Each smaller task does not hijack the CPU until it finishes like in the older multiprogramming but rather a fair share amount of the CPU time called quantum.

Just to make it easy to remember, both multiprogramming and multitasking operating systems are **(CPU) time sharing** systems. However, while in multiprogramming (older OSs) one program as a whole keeps running until it blocks, in multitasking (modern OSs) time sharing is best manifested because each running process takes only a fair quantum of the CPU time.

Q.2. (b) Consider there are 3 frame allocated to a process and the reference string is:1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6. Calculate the number of page faults for the following page replacement algorithms, assuming all frames are initially empty.

(i) LRU **(ii) FIFO** **(iii) Optimal**

Ans. LRU replacement:

(i)

	P	P	P	P	P	P	P	P	P	P
Initially Empty	1	2	3	4	2	1	5	6	2	1
	1	2	3	4	2	1	5	6	2	1

	P	P	P	P	P	P	P	P	P	P
	2	3	7	6	3	2	1	2	3	6
	2	3	7	6	3	2	1	2	3	6

Assuming one frames by LRU

(ii)

	P	P	P	P	P	P	P	P	P	P
Initially Empty	1	1	3	3	2	2	5	5	2	2
		2	2	4	4	1	1	6	6	1
	1	2	3	4	2	1	5	6	2	

	P	P	P	P	P	P	P	P	P
2	2	7	7	3	3	1	1	3	3
1	3	3	6	6	2	2	2	2	6
2	3	7	6	3	2	1	2	3	6

Assuming two frames by LRU

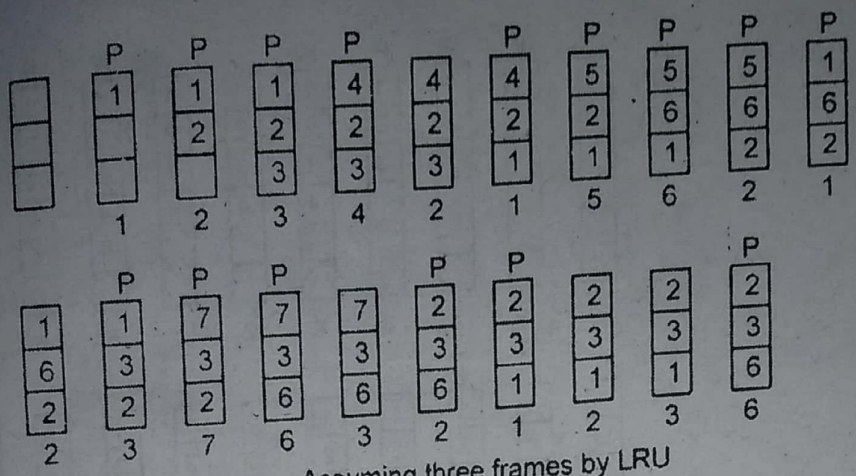
(iii) Initiall Empty

(iv) Initially Empty

1
2
5
6
2

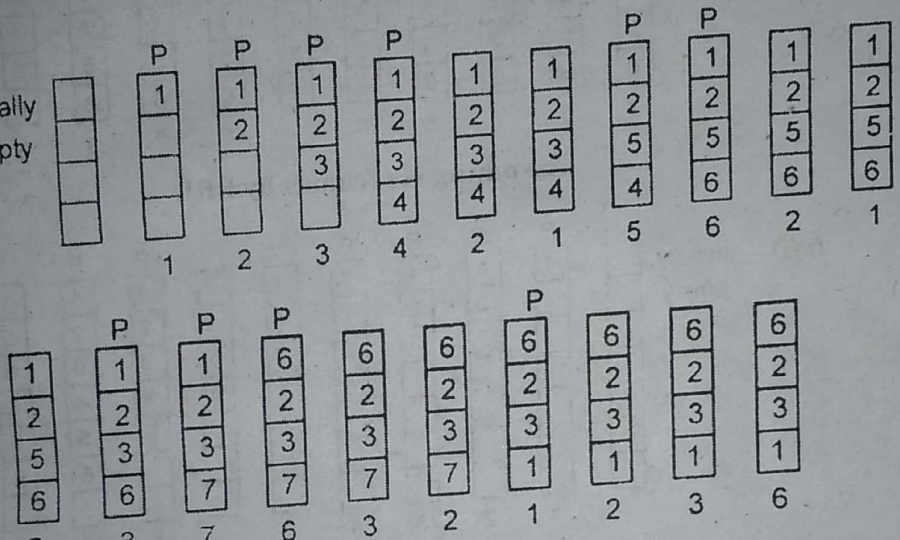
(v) Initially Empty

(iii) Initially Empty



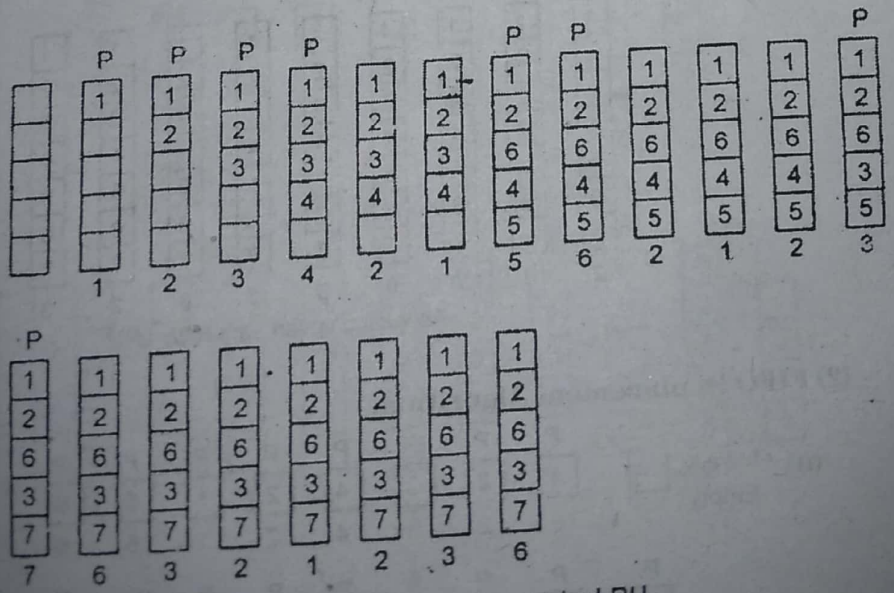
Assuming three frames by LRU

(iv) Initially Empty



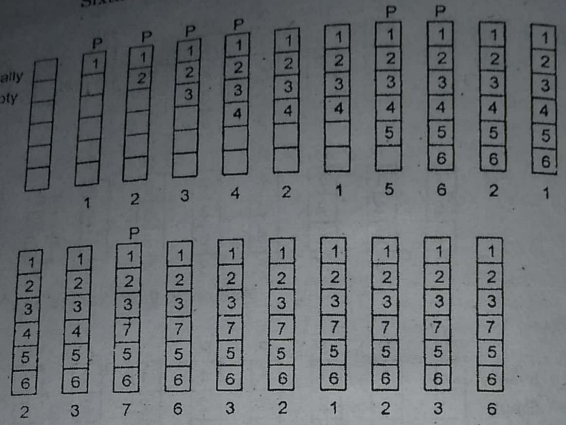
Assuming four frames by LRU

(v) Initially Empty



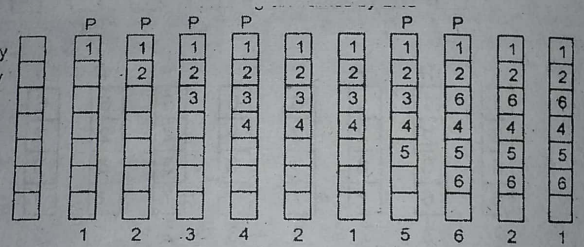
Assuming five frames by LRU

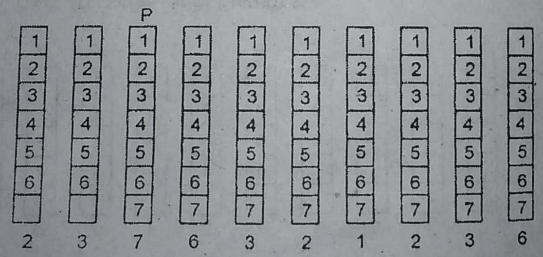
(vi) Initially Empty



Assuming six frames by LRU

(vii) Initially Empty

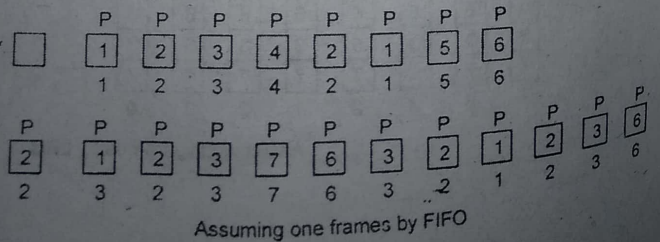




Assuming sevan frames by LRU

(2) FIFO Replacement Algorithm:

(i) Initially Empty



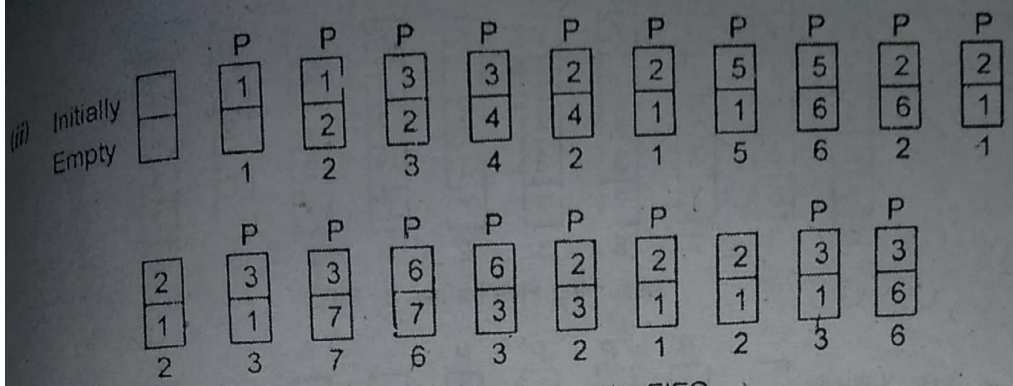
Assuming one frames by FIFO

(ii) Initially Empty

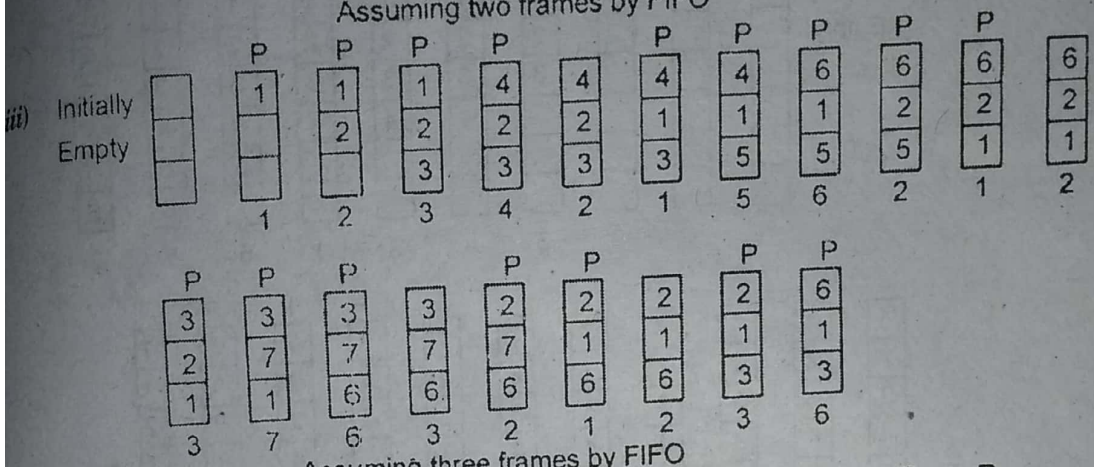
(iii) Initially Empty

(iv) Initially Empty

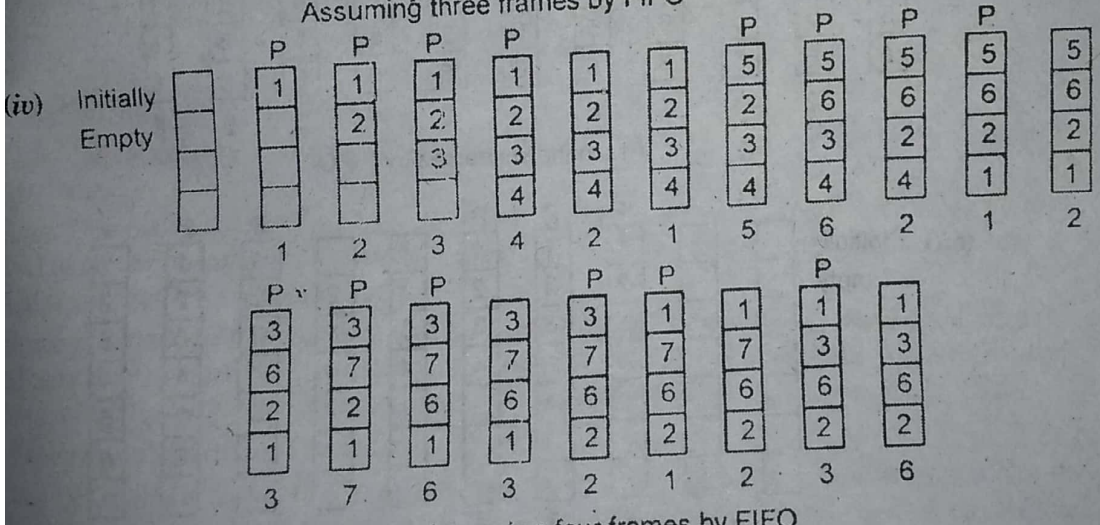
(v) Initially Empty



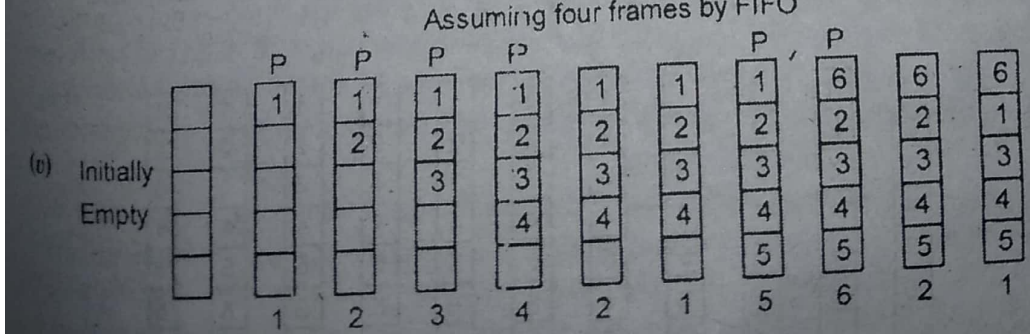
Assuming two frames by FIFO



Assuming three frames by FIFO

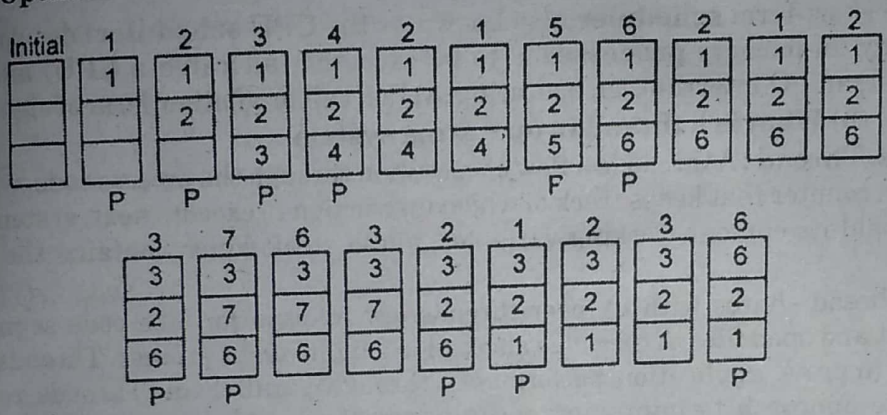


Assuming four frames by FIFO



No. of Frames	Number of page Faults	
	LRU	FIFO
One	20	20
Two	18	18
Three	15	17
Four	10	14
Five	08	10
Six	07	10
Seven	07	07

(iii) Optimal



Total no. of page faults = 11.

Q.3. (a) Explain various scheduling queues and scheduler in process scheduling. (3)

Ans. The process scheduling is the activity of the process manager that handles the moval of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Multiprogramming operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

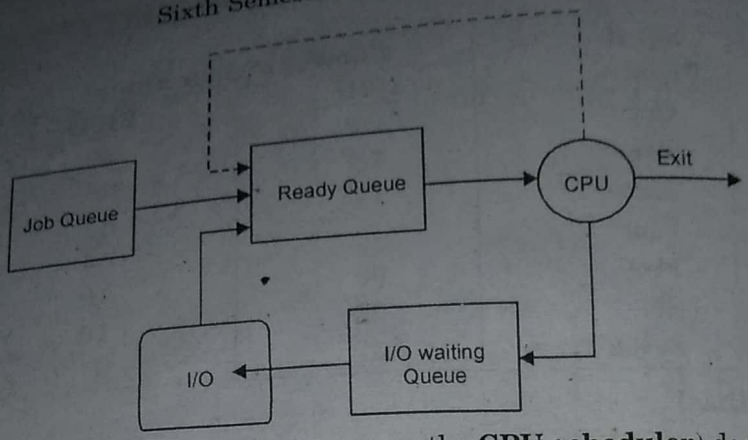
Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues.

- **Job queue–** This queue keeps all the processes in the system.
- **Ready queue–** This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues–** The processes which are blocked due to unavailability of an I/O device constitute this queue.

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.



The short-term **scheduler**(also known as the **CPU scheduler**) decides which of the ready, in-memory **processes** is to be executed (allocated a **CPU**) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal.

Q.3. (b) What is a thread in operating system. (2)

Ans. Thread : A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. A thread is also called a lightweight 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.

Q.3. (c) Consider the following set of processes with CPU burst and arrival time in milliseconds. (5)

Process	Arrival Time	Burst Time
A	3	10
B	1	1
C	4	2
D	3	1
E	0	5

Draw the Gantt chart & find:

(*i*) Average waiting time for these processes with the Shortest Remaining Time First, Round Robin (Time quantum = 3 ms) & FCFS scheduling algorithm.

(*ii*) Average turnaround time for these processes with the SRTF, Round Robin & FCFS Algorithms.

Ans. $TAT = CT - AT$

$WT = TAT - BT$

FCFS:

E	B	A	D	C

0 5 6 16 17 19

SRTF:

PR: (TQ = 3ms)

Q.4. (a) Explain
requirements for cri
Ans. Refer Q4(a)
Q.4. (b) Explain
Ans. Refer Q1(e)
Q.4. (c) What adv
different levels of a
Ans. Processes wh
such as editors, can b
for frequent servicing
switches to complete

Process	AT	BT	CT	TAT	wT
A	3	10	16	13	3
B	1	01	6	5	4
C	4	02	19	15	13
D	3	01	17	14	13
E	0	05	5	5	0

RTF:

E	B	E	D	C	E	A

0 1 2 3 4 6 9 19

Process	AT	BT	CT	TAT	wT
A	3	10	19	16	6
B	1	1	2	1	0
C	4	2	6	2	0
D	3	1	4	1	0
E	0	5	9	9	4

PR: (TQ = 3ms)

E	B	E	A	D	C	A

0 3 6 6 9 10 12 19

Process	AT	BT	CT	TAT	wT
A	3	10	19	16	6
B	1	1	4	3	2
C	4	2	12	8	6
D	3	1	10	7	6
E	0	5	6	6	1

Q.4. (a) Explain bakery algorithm.Prove that it satisfies all the three requirements for critical section problems. (4)

Ans. Refer Q4(a) First term Examination 2016

Q.4. (b) Explain race condition with suitable example. (3)

Ans. Refer Q1(e) First Term Examination 2016

Q.4. (c) What advantage is there in having different time quantum sizes on different levels of a multilevel queueing system. (3)

Ans. Processes which need more frequent servicing, for instance interactive processes such as editors, can be in a queue with a small time quantum. Processes with no need for frequent servicing can be in a queue with a larger quantum, requiring fewer context switches to complete the processing, making more efficient use of the computer.