Q7. IMPLEMENTATION OF ROUND ROBIN SCHEDULING ALGO.

A round-robin is a CPU scheduling algorithm that shares equal portions of resources in circular orders to each process and handles all processes without prioritization. In the round-robin, each process gets a fixed time interval of the slice to utilize the resources or execute its task called time quantum or time slice. Some of the round-robin processes are pre-empted if it executed in a given time slot, while the rest of the processes go back to the ready queue and wait to run in a circular order with the scheduled time slot until they complete their task. It removes the starvation for each process to achieve CPU scheduling by proper partitioning of the CPU.

PROGRAM :

```
#include<stdio.h>
#include<stdbool.h>
#include <stdlib.h>
struct process_struct{
  int pid;
  int at;
  int bt;
  int ct,wt,tat,rt,start_time;
  int bt_remaining;
}ps[100];
int findmax(int a, int b){
    return a>b?a:b;
}
int comparatorAT(const void * a, const void *b){
   int x =((struct process_struct *)a) -> at;
   int y =((struct process_struct *)b) -> at;
   if(x<y)
     return -1;
   else if( x>=y)
    return 1;
}
int comparatorPID(const void * a, const void *b){
   int x =((struct process_struct *)a) -> pid;
   int y =((struct process_struct *)b) -> pid;
   if(x<y)
     return -1;
   else if( x>=y)
    return 1;
}
int main(){
    int n,index;
    int cpu_utilization;
    bool visited[100]={false},is_first_process=true;
    int current_time = 0,max_completion_time;
    printf("Enter total number of processes: ");
    scanf("%d",&n);
    int queue[100],front=-1,rear=-1;
    float sum_tat=0,sum_wt=0,sum_rt=0;
    for(int i=0;i<n;i++)
        printf("\nEnter Process %d Arrival Time: ",i);
        scanf("%d",&ps[i].at);
        ps[i].pid=i;
    for(int i=0;i<n;i++){
       printf("\nEnter Process %d Burst Time: ",i);
       scanf("%d",&ps[i].bt);
       ps[i].bt_remaining= ps[i].bt;}
   printf("\nEnter time quanta: ");
   scanf("%d",&tq);
   qsort((void *)ps,n, sizeof(struct process_struct),comparatorAT);
   queue[rear]=0;
   while(completed != n){
     index = queue[front];
     front++;
     if(ps[index].bt_remaining == ps[index].bt){
           ps[index].start_time = findmax(current_time,ps[index].at);
           total_idle_time += (is_first_process == true) ? 0 : ps[index].start_time - current_time;
           current_time =  ps[index].start_time;
           is_first_process = false;
     }
     if(ps[index].bt_remaining-tq > 0){
           ps[index].bt_remaining -= tq;
           current_time += tq;
     }
     else {
           current_time += ps[index].bt_remaining;
           ps[index].bt_remaining = 0;
           completed++;
           ps[index].ct = current_time;
           ps[index].tat = ps[index].ct - ps[index].at;
           ps[index].wt = ps[index].tat - ps[index].bt;
           ps[index].rt = ps[index].start_time - ps[index].at;
           sum_tat += ps[index].tat;
           sum_wt += ps[index].wt;
           sum_rt += ps[index].rt;     }
     for(int i = 1; i < n; i++)
         if(ps[i].bt_remaining > 0 && ps[i].at <= current_time && visited[i] == false) {
           visited[i] = true;}
     if( ps[index].bt_remaining> 0)
         queue[++rear]=index;
     if(front>rear){
         for(int i = 1; i < n; i++){
           if(ps[i].bt_remaining > 0){
             queue[rear++]=i;
             visited[i] = true;}    }}}
  max_completion_time = INT_MIN;
  for(int i=0;i<n;i++)
        max_completion_time = findmax(max_completion_time,ps[i].ct);
  length_cycle = max_completion_time - ps[0].at;
  cpu_utilization = (float)(length_cycle - total_idle_time)/ length_cycle;
  qsort((void *)ps,n, sizeof(struct process_struct),comparatorPID);
  printf("\nProcess No.\tAT\tCPU Burst Time\tStart Time\tCT\tTAT\tWT\tRT\n");
  for(int i=0;i<n;i++)
    printf("%d\t\t%d\t%d\t\t%d\t\t%d\t%d\t%d\t%d\n",i,ps[i].at,ps[i].bt,ps[i].start_time,ps[i].ct,ps[i].tat,ps[i].wt,ps[i].rt);
  printf("\n");
  printf("\nAverage Turn Around time= %.2f",(float)sum_tat/n);
  printf("\nAverage Waiting Time= %.2f",(float)sum_wt/n);
  printf("\nAverage Response Time= %.2f",(float)sum_rt/n);
  printf("\nThroughput= %.2f",n/(float)length_cycle);
  printf("\nCPU Utilization(Percentage)= %.2f",cpu_utilization*100);
  return 0;
}
```

Q11.IMPLEMENTATION OF BANKER'S AVOIDANCE ALGO FOR DEADLOCK AVOIDANCE.
Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether them allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise, the process must wait until some other process release the resources.

PROGRAM :
```
#include<stdio.h>
int main(){
        int n,m;
        printf("ENTER THE NUMBER OF PROCESSES        :");
        scanf("%d",&n);
        printf("ENTER THE NUMBER OF RESOURCES        :");
        scanf("%d",&m);
        int allocation[n][m];
        printf("ENTER THE ALLOCATION MATRIX\n");
        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        scanf("%d",&allocation[i][j]);
                }
        }
        int process[n][m];
        printf("ENTER THE PROCESS MATRIX \n");
        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        scanf("%d",&process[i][j]);
                }
        }
        int available[m];
        printf("ENTER THE AVAILABLE MATRIX\n");
        for(int i=0;i<m;i++){
                scanf("%d",&available[i]);
        }
        int fun[n],answer[n];
        int index=0;
        for(int i=0;i<n;i++){
                fun[i]=0;
        }
        int need[n][m];
        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        need[i][j]=process[i][j]-allocation[i][j];
                }
        }
        printf("NEED MATRIX IS \n");
        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        printf("  %d  ",need[i][j]);
                }
                printf("\n");
        }
        int y=0;
        for(int k=0;k<n;k++){
                for(int i=0;i<n;i++){
```

```
                if(fun[i]==0){
                    int flag=0;
                    for(int j=0;j<m;j++){
                        if(need[i][j]>available[j]){
                            flag=1;
                            break;
                        }
                    }
                    if(flag==0){
                        answer[index++]=i;
                        for(y=0;y<m;y++){
                            available[y]+=allocation[i][y];
                        }
                        fun[i]=1;
                    }
                }
            }
        }
        int flag=1;
        for(int i=0;i<n;i++){
            if(fun[i]==0){
                flag=0;
                printf("THIS PROCESS WILL NOT COMPLETE ");
            }
        }
        if(flag==1){
            printf("PROCESS THAT WILL COMPLETE     :");
            for(int i=0;i<n-1;i++){
                printf("  P-%d   ",answer[i]);
            }
            printf("   P-%d  ",answer[n-1]);
        }
        return 0;
}
```

Q12.IMPLEMENTATION OF FIFO PAGE REPLACEMENT ALGORITHM

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

PROGRAM :

```
#include<stdio.h>
#include<stdbool.h>
#include<string.h>
struct PageTable {
    int frame_no;
    bool valid;
};
bool isPagePresent(struct PageTable PT[],int page,int n) {
    if(PT[page].valid == 1)
            return true;
    return false;
}
void updatePageTable(struct PageTable PT[],int page,int fr_no,int status) {
    PT[page].valid=status;
    PT[page].frame_no=fr_no;
}
void printFrameContents(int frame[],int no_of_frames) {
    for(int i=0; i<no_of_frames; i++)
            printf("%d ",frame[i]);
    printf("\n");
}
int main() {
    int i,n,no_of_frames,page_fault=0,current=0;
    bool flag=false;
    printf("\n Enter the no. of pages:\n");
    scanf("%d",&n);
    int reference_string[n];
    printf("\n Enter the reference string(different page numbers) :\n");
    for(int i=0; i<n; i++)
            scanf("%d",&reference_string[i]);
    printf("\n Enter the no. of frames you want to give to the process :");
    scanf("%d",&no_of_frames);
    int frame[no_of_frames];
    memset(frame,-1,no_of_frames*sizeof(int));
    struct PageTable PT[50] ;
    for(int i=0; i<50; i++)
            PT[i].valid=0;
    printf("\n****The Contents inside the Frame array at different time:****\n");
    for(int i=0; i<n; i++) {
            if( ! (isPagePresent(PT,reference_string[i],n))) {
                    page_fault++;
                    if(flag==false && current < no_of_frames) {
                            frame[current]=reference_string[i];
                            printFrameContents(frame,no_of_frames);
                            updatePageTable(PT,reference_string[i],current,1);
                            current = current + 1;
                            if(current == no_of_frames) {
                                    current=0;
                                    flag=true;
```

```
                                }
                        }
                        else {
                                updatePageTable(PT,frame[current], -1,0);
                                frame[current]=reference_string[i];
                                printFrameContents(frame,no_of_frames);
                                updatePageTable(PT,reference_string[i],current,1);
                                current = ( current + 1)% no_of_frames;
                        }
                }
        }
        printf("\nTotal No. of Page Faults = %d\n",page_fault);
        printf("\nPage Fault ratio = %.2f\n",(float)page_fault/n);
        printf("\nPage Hit Ratio = %.2f\n",(float)(n- page_fault)/n);
        return 0;
}
```

Q13.IMPLEMENTATION OF LRU PAGE REPLACEMENT ALGORITHM

Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page in not found in the memory frames.

PROGRAM :

```
#include<stdio.h>
#include<stdbool.h>
#include<string.h>
#include<limits.h>
struct PageTable {
    int frame_no;
    int last_time_of_access;
    bool valid;
};
bool isPagePresent(struct PageTable PT[],int page) {
    if(PT[page].valid == 1)
            return true;
    return false;
}
void updatePageTable(struct PageTable PT[],int page,int fr_no,int status,int access_time) {
    PT[page].valid=status;
    if(status == 1 ) {
            PT[page].last_time_of_access =  access_time;
            PT[page].frame_no=fr_no;
    }
}
void printFrameContents(int frame[],int no_of_frames) {
    for(int i=0; i<no_of_frames; i++)
            printf("%d ",frame[i]);
    printf("\n");
}
void searchLRUPage(struct PageTable PT[], int frame[], int no_of_frames, int *LRU_page_index) {
    int min = INT_MAX;
    for(int i=0; i<no_of_frames; i++) {
            if(PT[frame[i]].last_time_of_access < min) {
                    min = PT[frame[i]].last_time_of_access;
                    *LRU_page_index = i;
            }
    }
}
int main() {
    int i,n,no_of_frames,page_fault=0,current=0;
    bool flag=false;
    printf("\n Enter the no. of pages:\n");
    scanf("%d",&n);
    int reference_string[n];
    printf("\n Enter the reference string(different page numbers) :\n");
    for(int i=0; i<n; i++)
            scanf("%d",&reference_string[i]);
    printf("\n Enter the no. of frames you want to give to the process :");
    scanf("%d",&no_of_frames);
    int frame[no_of_frames];
    memset(frame,-1,no_of_frames*sizeof(int));
    struct PageTable PT[50] ;
    for(int i=0; i<50; i++)
            PT[i].valid=0;
    printf("\n****The Contents inside the Frame array at different time:****\n");
    for(int i=0; i<n; i++) {
            if( ! (isPagePresent(PT,reference_string[i]))) {
                    page_fault++;
                    if(flag==false && current < no_of_frames) {
                            frame[current]=reference_string[i];
                            printFrameContents(frame,no_of_frames);
                            updatePageTable(PT,reference_string[i],current,1,i);
                            current = current + 1;
                            if(current == no_of_frames) {
                                    flag=true;
                            }
                    }
                    else {
                            int LRU_page_index;
                            searchLRUPage(PT,frame,no_of_frames,&LRU_page_index);
                            updatePageTable(PT,frame[LRU_page_index], -1,0,i);
                            frame[LRU_page_index]=reference_string[i];
                            printFrameContents(frame,no_of_frames);
                            updatePageTable(PT,reference_string[i],LRU_page_index,1,i);
                    }
            }
            PT[reference_string[i]].last_time_of_access =  i;
    }
    printf("\nTotal No. of Page Faults = %d\n",page_fault);
    printf("\nPage Fault ratio = %.2f\n",(float)page_fault/n);
    printf("\nPage Hit Ratio = %.2f\n",(float)(n- page_fault)/n);
    return 0;
}
```

Q14. IMPLEMENTATION OF FCFS DISK SCHEDULING ALGORITHM.
FCFS is the simplest disk scheduling algorithm. As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue. The algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service.

PROGRAM :

```
#include<stdio.h>
#include<math.h>
int main(){
    int n,initial,ans=0;
    printf("ENTER THE NUMBER OF DISK PLATTERS   :             ");
    scanf("%d",&n);
    int RQ[n];
    printf("ENTER THE NUMBER OF REQUESTS : ");
    for(int i=0;i<n;i++){
            scanf("%d",&RQ[i]);
    }
    printf("ENTER THE INTIALS : ");
    scanf("%d",&initial);
    ans=ans+abs(initial-RQ[0]);
    for(int i=1;i<n;i++){
            ans=ans + abs (RQ[i]-RQ[i-1]);
            initial=RQ[i];
    }
    printf("TOTAL HEAD COUNT IS %d",ans);
    return 0;
}
```

Q15. IMPLEMENTATION OF SCAN DISK SCHEDULING ALGORITHM.
In SCAN disk scheduling algorithm, head starts from one end of the disk and moves towards the other end, servicing requests in between one by one and reach the other end. Then the direction of the head is reversed and the process continues as head continuously scan back and forth to access the disk. So, this algorithm works as an elevator and hence also known as the elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

PROGRAM :
```
#include<stdio.h>
#include<stdlib.h>
#include<limits.h>
int comparator(const void * a, const void *b) {
    int x =*(int *)a;
    int y =*(int *)b;
    if(x<y)
            return -1;
    else if( x>=y)
            return 1;
}
int min_element(int request_queue[],int n) {
    int min = INT_MAX;
    for(int i=0; i<n; i++) {
            if(request_queue[i] < min)
                    min = request_queue[i];
    }
    return min;
}
int max_element(int request_queue[],int n) {
    int max = INT_MIN;
    for(int i=0; i<n; i++) {
            if(request_queue[i] > max)
                    max = request_queue[i];
    }
    return max;
}
int applySCANAlgo(int total_cylinders,int request_queue[], int initial_pos, int seek_sequence[], int direction,int
*sequence_size,int n) {
    int total_head_movement=0,j=0,k=0;
    int left[n+1], right[n+1];
    if(direction == 0) {
            if(initial_pos > min_element(request_queue,n))
                    right[j++]=total_cylinders-1;
            right[j++]=initial_pos;
    } else if(direction == 1) {
            if(initial_pos < max_element(request_queue,n))
                    left[k++]=0;
            left[k++]=initial_pos;
    }
    for (int i = 0; i<n; i++) {
            if (request_queue[i] < initial_pos)
                    left[k++]=request_queue[i];
            if (request_queue[i] > initial_pos)
                    right[j++]=request_queue[i];
    }
    qsort((void *)left,k, sizeof(int),comparator);
    qsort((void *)right,j, sizeof(int),comparator);
    int completed = 2;
    while (completed--) {
            if (direction == 0) {
                    for (int i = 0; i < j; i++) {
                            total_head_movement += abs(initial_pos - right[i]);
                            initial_pos = right[i];
                            seek_sequence[*sequence_size]=right[i];
                            (*sequence_size)++;
                    }
                    direction = 1;
            }
            else if (direction == 1) {
                    for (int i = k - 1; i >= 0; i--) {
                            total_head_movement +=  abs(initial_pos - left[i]);
                            initial_pos = left[i];
                            seek_sequence[*sequence_size]=left[i];
                            (*sequence_size)++;
                    }
                    direction = 0;
            }
    }
    return total_head_movement;
}
int main() {
    int total_cylinders,total_head_movement=0,initial_pos,n,direction,pos;
    printf("\nEnter the total no. of cylinders in HDD:\n");
    scanf("%d",&total_cylinders);
    printf("\nEnter the no. of cylinders to enter in Request queue:\n");
    scanf("%d",&n);
    int request_queue[n];
    int seek_sequence[n+10];
    int sequence_size=0;
    printf("\nEnter the cylinders no. in Request queue :\n");
    for(int i=0; i<n; i++)
            scanf("%d",&request_queue[i]);
    printf("\nEnter the initial Position of RW head: ");
    scanf("%d",&initial_pos);
    printf("\nEnter the direction in which Read Write head is moving:\n ");
    printf("\nEnter 0 if moving to higher cylinder else Enter 1: ");
    scanf("%d",&direction);
    if(initial_pos < 0 || initial_pos > total_cylinders - 1) {
            printf("Wrong Initial Position Enetered !!");
            exit(0);
    }
    total_head_movement =
applySCANAlgo(total_cylinders,request_queue,initial_pos,seek_sequence,direction,&sequence_size,n);
    printf("\n\n*********** OUTPUT **********");
    printf("\nSeek Sequence: ");
    for(int i=0; i<sequence_size; i++)
            printf("%d ",seek_sequence[i]);
    printf("\nTotal No. of Head Movements = %d\n",total_head_movement);
    printf("\nAverage head movements = %.2f\n\n",(float)total_head_movement/n);
    return 0;
}
```

Q10. DEMONSTRATING of execl() WHERE Parent PROCESS EXECUTES "ls" COMMAND AND Child PROCESS EXECUTES "date" COMMAND

execl command in Linux is used to execute a command from the bash itself. This command does not create a new process it just replaces the bash with the command to be executed. If the exec command is successful, it does not return to the calling process.

PROGRAM :

```
#include<stdio.h>   //HEADER FILES #include<sys/wait.h> #include<sys/types.h>
#include<stdlib.h>
#include<unistd.h>
int main() {           //MAIN FUNCTION
     pid_t pid;
     int status;
     pid=fork();       //CALLING FORK TO CREATE A CHILD PROCESS
     if(pid==0) {
             printf("\nChild is executing Date command:\n");
             execl("/bin/date","date",NULL);
             exit(0);
     } else {
             wait(&status);
             printf("\nParent executing ls -l command:\n");
             execl("/bin/ls","ls","-l",NULL);
     }
     return 0;
 }  //END OF MAIN
```

Q9. Implementation of Interprocess communication using shared memory.

SHARED MEMORY:

The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

Server reads from the input file.
The server writes this data in a message using either a pipe, fifo or message queue.
The client reads the data from the IPC channel,again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
Finally the data is copied from the client's buffer.

PROGRAM :

```
#include<stdio.h>   //HEADER FILES #include<string.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
int main() {            //MAIN FUNCTION
    key_t key;
    int shmid;
    void *ptr;
    key=ftok("srfile",'A');
    shmid=shmget(key,1024,0666|IPC_CREAT);
    ptr=shmat(shmid,(void*)0,0);
    printf("\nInput Data :");
    gets(ptr);
    printf("\nThe Data stored :%s\n",ptr);
    shmdt(ptr);
    shmctl(shmid,IPC_RMID,NULL);
    return(0);
} //END OF MAIN
```

Q8. Implementation of Interprocess Communication using PIPE.

INTER-PROCESS COMMUNICATION

Inter-process communication is a “process to process “communication in a single system. 5 types of IPC’s are:

PIPE
FIFO
MESSAGE QUEUE
SHARED MEMORY
SEMAPHORE

PROGRAM :

```
#include<stdio.h>     //HEADER FILES #include<stdlib.h>
#include<unistd.h>
#include<string.h>
int main() {   //MAIN FUNCTION
        pid_t pid;
        char arr[100],str[100];
        int fd[2],nbr,nbw;
        pipe(fd);           //CREATING A PIPE
        pid=fork();         //CALLING FORK TO CREATE A CHILD PROCESS
        if(pid==0) {
                printf("\nEnter a string: ");
                gets(str);
                nbw=write(fd[1],str,strlen(str));
                printf("Child wrote %d bytes\n",nbw);
                exit(0);
        } else {
                nbr=read(fd[0],arr,sizeof(arr));
                arr[nbr]='\0';
                printf("Parent read %d bytes : %s\n",nbr,arr);
        }
        return 0;
} //END OF MAIN
```