Graphic Era
HILL UNIVERSITY
Established by an Act of the State Legislature of Uttarakhand (Adhiniyam Sankhya 12 of 2011)

Table of Contents

13	C program to Implement Least recently used page replacement policy	
14	C program to Implement FCFS Disk Scheduling Algorithm	
15.	C program to Implement SCAN Disk Scheduling Algorithm	

Program 1

Q. Write a C program to demonstrate the use of fork() System call

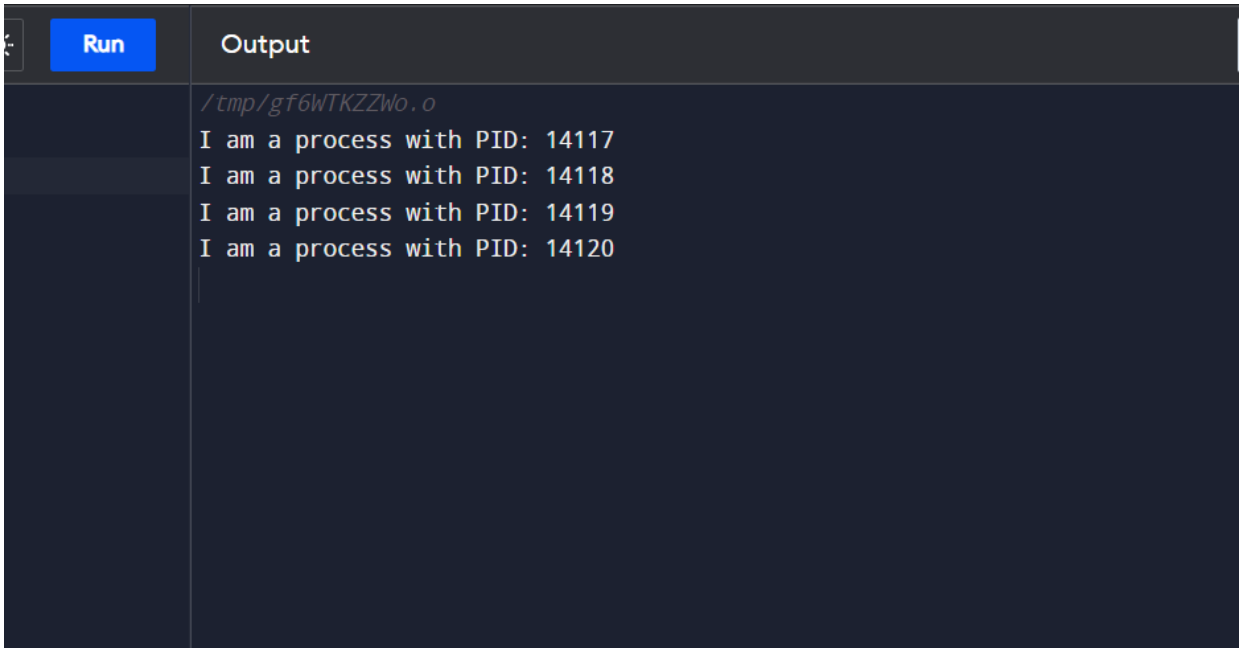
Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call.

Source Code:

```
#include<stdio.h>
#include <unistd.h>

int main() {

    fork();
    fork();

    printf("I am a process with PID: %d\n", getpid());

    return 0;
}
```

Madhav Singh 1918441 Section: I

Output



Program 2

Q. C Program in which Parent Process Computes the SUM OF EVEN NUMBERS and Child Process Computes the sum of ODD NUMBERS stored in array using fork () . First the child process should print its answer i.e sum of odd numbers, then parent should print its answer, i.e sum of even numbers.

A parent process is one that creates a child process using a fork() system call. A parent process may have multiple child processes, but a child process only one parent process. On the success of a fork() system call: The Process ID (PID) of the child process is returned to the parent process.

Source Code:

```
#include <stdio.h>
#include <unistd.h>

int main() {

    int arr[10] = {10, 14, 2, 5, 1, 3, 7, 19, 18};
    int pid = fork();

    if (pid != 0) {
        int evenSum = 0, oddSum = 0;

        for (int i=0; i<10; i++){
            if (arr[i] % 2 == 1) {
                oddSum += arr[i];
            }
        }
        printf("Odd elements sum by child process: %d\n", oddSum);

        for (int i=0; i<10; i++){
            if (arr[i] % 2 == 0) {
                evenSum += arr[i];
```

```
            }
        }
        printf("Even elements sum by parent process: %d\n", evenSum);
    }

    return 0;
}
```

Madhav Singh 1918441 Section: I

Output

Program 3

Q. C program to Implement the Orphan Process and Zombie Process.

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

Source Code:

```
#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main() {

    int pid = fork();

    if (pid > 0) {
        printf("I am parent!\n");
        sleep(5);
    } else if (pid == 0) {
        printf("I am an orphan process!\n");
    }
    else
        printf("I am a zombie process!\n");
        exit(0);
    return 0;
}
```

Output


Output
Clear
/tmp/gf6WTKZZWo.o
I am parent!
I am an orphan process!

Program 4

Q. C program to Implement FCFS CPU Scheduling Algorithm

First Come First Serve (FCFS) is **an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival .....** In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue.

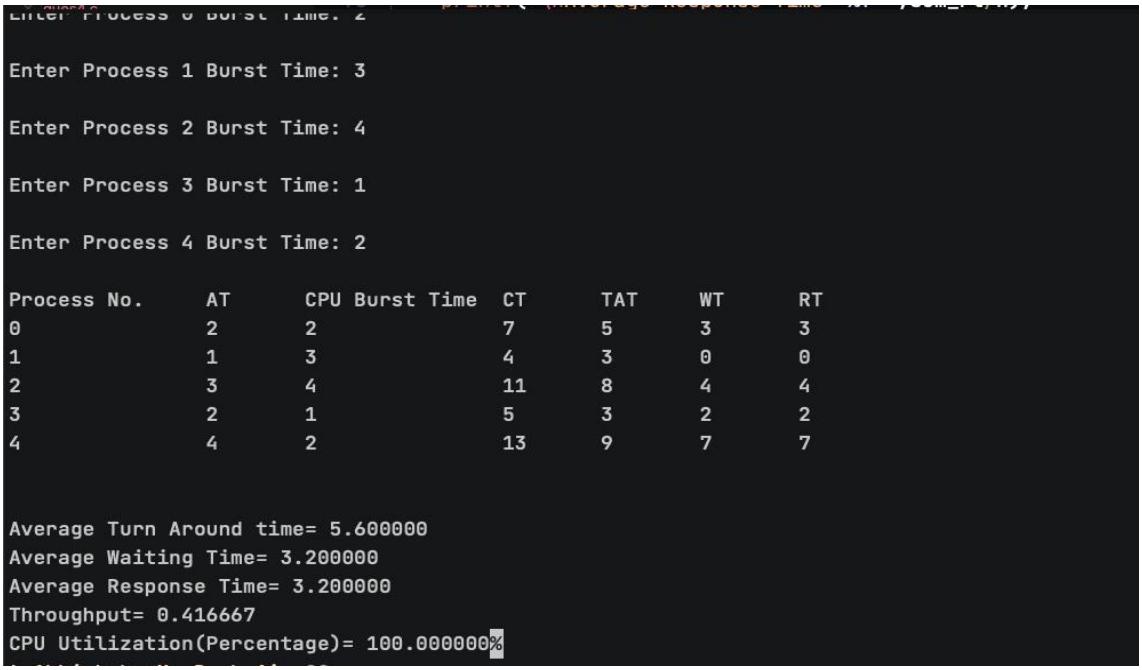
Source Code:

```
#include<stdio.h>
#include <stdlib.h>

struct process_struct
{
  int pid;
  int at;       //Arrival Time
  int bt;       //CPU Burst time
  int ct,wt,tat,rt,start_time;   // Completion, waiting, turnaround, response time
}ps[100];       //Array of structure to store the info of each process.


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;  // No sorting
    else if( x>=y) // = is for stable sort
        return 1; // Sort
```

```
    return 0;
}
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; // No sorting
    else if( x>=y)
       return 1;    // Sort
    return 0;
}
int main()
{
    int n;
    printf("Enter total number of processes: ");
    scanf("%d",&n);
    float sum_tat=0,sum_wt=0,sum_rt=0;
    int length_cycle,total_idle_time=0;
    float cpu_utilization;

    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);
    }


    //sort
    qsort((void *)ps,n, sizeof(struct process_struct),comparatorAT);

   //calculations
    for(int i=0;i<n;i++)
```

```
    {
      ps[i].start_time = (i==0) ? ps[i].at : findmax(ps[i].at, ps[i-1].ct);
      ps[i].ct =  ps[i].start_time + ps[i].bt;
      ps[i].tat = ps[i].ct-ps[i].at;
      ps[i].wt = ps[i].tat-ps[i].bt;
      ps[i].rt=ps[i].wt;

      sum_tat += ps[i].tat;
      sum_wt  +=  ps[i].wt;
      sum_rt += ps[i].rt;
      total_idle_time += (i==0) ? 0 : (ps[i].start_time -  ps[i-1].ct);
    }

    length_cycle = ps[n-1].ct - ps[0].start_time;

    qsort((void *)ps,n, sizeof(struct process_struct),comparatorPID);

    //Output
    printf("\nProcess No.\tAT\tCPU Burst Time\tCT\tTAT\tWT\tRT\n");
    for(int i=0;i<n;i++)

printf("%d\t\t%d\t%d\t\t%d\t%d\t%d\t%d\n",ps[i].pid,ps[i].at,ps[i].bt,ps[i].ct,ps[i].t
at,ps[i].wt,ps[i].rt);

    printf("\n");

    cpu_utilization = (float)(length_cycle - total_idle_time)/ length_cycle;

    printf("\nAverage  Turn Around  time=  %f  ",sum_tat/n);
    printf("\nAverage Waiting Time= %f ",sum_wt/n);
    printf("\nAverage Response Time= %f ",sum_rt/n);
    printf("\nThroughput= %f",n/(float)length_cycle);
    printf("\nCPU Utilization(Percentage)= %f",cpu_utilization*100);

    return 0;
}
```

Output


Enter Process 1 Burst Time: 3
Enter Process 2 Burst Time: 4
Enter Process 3 Burst Time: 1
Enter Process 4 Burst Time: 2
Process No. AT CPU Burst Time CT TAT WT RT
0 2 2 7 5 3 3
1 1 3 4 3 0 0
2 3 4 11 8 4 4
3 2 1 5 3 2 2
4 4 2 13 9 7 7
Average Turn Around time= 5.600000
Average Waiting Time= 3.200000
Average Response Time= 3.200000
Throughput= 0.416667
CPU Utilization(Percentage)= 100.000000%

Program 5

Q. C program to implement SRTF algorithm.

In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, **the short term scheduler schedules the process** with the least remaining burst time among the list of available processes and the running process.

Source Code:

```
#include<stdio.h>
#include<stdbool.h>
#include<limits.h>

struct process_struct
{
  int pid;
  int at;
  int bt;
  int ct,wt,tat,rt,start_time;
}ps[100];

int findmax(int a, int b)
{
   return a>b?a:b;
}

int findmin(int a, int b)
{
   return a<b?a:b;
}

int main()
{

   int n;
   float bt_remaining[100];
   bool is_completed[100]={false},is_first_process=true;
   int current_time = 0;
   int completed = 0;;
```

```
float sum_tat=0,sum_wt=0,sum_rt=0,total_idle_time=0,length_cycle,prev=0;
float cpu_utilization;

int max_completion_time,min_arrival_time;

printf("Enter total number of processes: ");
scanf("%d",&n);
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);
    bt_remaining[i]= ps[i].bt;
}

while(completed!=n)
{
    //find process with min. burst time in ready queue at current time
    int min_index = -1;
    int minimum = INT_MAX;
    for(int i = 0; i < n; i++) {
        if(ps[i].at <= current_time && is_completed[i] == false) {
            if(bt_remaining[i] < minimum) {
                minimum = bt_remaining[i];
                min_index = i;
            }
            if(bt_remaining[i]== minimum) {
                if(ps[i].at < ps[min_index].at) {
                    minimum= bt_remaining[i];
                    min_index = i;
                }
            }
        }
    }


    if(min_index==-1)
    {
        current_time++;
    }
```

```
        else
        {
            if(bt_remaining[min_index] == ps[min_index].bt)
            {
                    ps[min_index].start_time = current_time;
                    total_idle_time += (is_first_process==true) ? 0 :
(ps[min_index].start_time -  prev);
                    is_first_process=false;
            }
            bt_remaining[min_index] -= 1;
            current_time++;
            prev=current_time;
            if(bt_remaining[min_index] == 0)
            {
                ps[min_index].ct = current_time;
                ps[min_index].tat = ps[min_index].ct - ps[min_index].at;
                ps[min_index].wt= ps[min_index].tat - ps[min_index].bt;
                ps[min_index].rt = ps[min_index].start_time - ps[min_index].at;


                sum_tat +=ps[min_index].tat;
                sum_wt += ps[min_index].wt;
                sum_rt += ps[min_index].rt;
                completed++;
                is_completed[min_index]=true;
                //total_idle_time += (is_first_process==true) ? 0 :
(ps[min_index].start_time -  prev);
               // prev= ps[min_index].ct; // or current_time;
            }
        }
    }
    //Calculate Length of Process completion cycle
    max_completion_time = INT_MIN;
    min_arrival_time = INT_MAX;
    for(int i=0;i<n;i++)
    {
        max_completion_time = findmax(max_completion_time,ps[i].ct);
        min_arrival_time = findmin(min_arrival_time,ps[i].at);
    }
    length_cycle = max_completion_time - min_arrival_time;

    //Output
    printf("\nProcess No.\tAT\tCPU Burst Time\tCT\tTAT\tWT\tRT\n");
    for(int i=0;i<n;i++)
```

```
printf("%d\t\t%d\t%d\t\t%d\t%d\t%d\n",ps[i].pid,ps[i].at,ps[i].bt,ps[i].ct,ps[i].t
at,ps[i].wt,ps[i].rt);

    printf("\n");

    cpu_utilization = (float)(length_cycle - total_idle_time)/ length_cycle;

    printf("\nAverage Turn Around time= %f ",(float)sum_tat/n);
    printf("\nAverage Waiting Time= %f ",(float)sum_wt/n);
    printf("\nAverage Response Time= %f ",(float)sum_rt/n);
    printf("\nThroughput= %f",n/(float)length_cycle);
    printf("\nCPU Utilization(Percentage)= %f",cpu_utilization*100);
    return 0;
}
```

Output

```
Enter Process 3 Arrival Time: 2

Enter Process 0 Burst Time: 2

Enter Process 1 Burst Time: 2

Enter Process 2 Burst Time: 3

Enter Process 3 Burst Time: 4

Process No.     AT      CPU Burst Time  CT      TAT     WT      RT
0               1       2               3       2       0       0
1               2       2               5       3       1       1
2               3       3               8       5       2       2
3               2       4               12      10      6       6


Average Turn Around time= 5.000000
Average Waiting Time= 2.250000
Average Response Time= 2.250000
Throughput= 0.363636
CPU Utilization(Percentage)= 100.000000%
```

Program 6

Q. C program to Implement Round Robin CPU scheduling algo.

Round Robin is a **CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way**. It is simple, easy to implement, and starvation-free as all processes get fair share of CPU ..... It is preemptive as processes are assigned CPU only for a fixed slice of time at most.

Source Code:

```
#include<stdio.h>

int main()
{
   int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
   float avg_wt, avg_tat;
   printf("Total number of process in the system: ");
   scanf("%d", &NOP);
   y = NOP;

   for(i=0; i<NOP; i++)
   {
   printf("\nEnter the Arrival and Burst time of the Process[%d]", i+1);
   printf("\n\nArrival time is: ");
   scanf("%d", &at[i]);
   printf("Burst time is: ");
   scanf("%d", &bt[i]);
   temp[i] = bt[i]; // store the burst time in temp array
   }
   printf("\nEnter the Time Quantum for the process: ");
   scanf("%d", &quant);
   printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
   for(sum=0, i = 0; y!=0; )
   {
   if(temp[i] <= quant && temp[i] > 0) // define the conditions
      {
      sum = sum + temp[i];
      temp[i] = 0;
      count=1;
      }
```

```
        else if(temp[i] > 0)
        {
            temp[i] = temp[i] - quant;
            sum = sum + quant;
        }
        if(temp[i]==0 && count==1)
        {
            y--; //decrement the process no.
            printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i],
sum-at[i]-bt[i]);
            wt = wt+sum-at[i]-bt[i];
            tat = tat+sum-at[i];
            count =0;
        }
        if(i==NOP-1)
        {
            i=0;
        }
        else if(at[i+1]<=sum)
        {
            i++;
        }
        else
        {
            i=0;
        }
    }
    avg_wt = wt * 1.0/NOP;
    avg_tat = tat * 1.0/NOP;
    printf("\n Average Turn Around Time: %.3f", avg_wt);
    printf("\n Average Waiting Time: %.3f", avg_tat);

    return 0;
}
```

Output

```
Enter the Arrival and Burst time of the Process[1]

Arrival time is: 0
Burst time is: 8

Enter the Arrival and Burst time of the Process[2]

Arrival time is: 1
Burst time is: 4

Enter the Arrival and Burst time of the Process[3]

Arrival time is: 3
Burst time is: 4

Enter the Time Quantum for the process: 6

 Process No             Burst Time             TAT            Waiting Time
Process No[2]           4                              9                       5
Process No[3]           4                              11                      7
Process No[1]           8                              16                      8
 Average Turn Around Time: 6.667
 Average Waiting Time: 12.000%
```

Program 7

Q. C program to Implement Preemptive Priority CPU scheduling algo.

SRTF, Which Stands for **Shortest Remaining Time First** is a scheduling algorithm used in Operating Systems, which can also be called as the preemptive version of the SJF scheduling algorithm. The process which has the least processing time remaining is executed first.

Source Code:

```
#include<stdio.h>
#include<stdbool.h>
#include<limits.h>

struct process_struct
{
  int pid;
  int at;
  int bt;
  int ct,wt,tat,rt,start_time;
}ps[100];

int findmax(int a, int b)
{
    return a>b?a:b;
}

int findmin(int a, int b)
{
    return a<b?a:b;
}

int main()
{

    int n;
    float bt_remaining[100];
    bool is_completed[100]={false},is_first_process=true;
    int current_time = 0;
    int completed = 0;;
```

```
float sum_tat=0,sum_wt=0,sum_rt=0,total_idle_time=0,length_cycle,prev=0;
float cpu_utilization;

int max_completion_time,min_arrival_time;

printf("Enter total number of processes: ");
scanf("%d",&n);
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);
    bt_remaining[i]= ps[i].bt;
}

while(completed!=n)
{
    int min_index = -1;
    int minimum = INT_MAX;
    for(int i = 0; i < n; i++) {
        if(ps[i].at <= current_time && is_completed[i] == false) {
            if(bt_remaining[i] < minimum) {
                minimum = bt_remaining[i];
                min_index = i;
            }
            if(bt_remaining[i]== minimum) {
                if(ps[i].at < ps[min_index].at) {
                    minimum= bt_remaining[i];
                    min_index = i;
                }
            }
        }
    }


    if(min_index==-1)
    {
        current_time++;
    }
    else
```

```
        {
            if(bt_remaining[min_index] == ps[min_index].bt)
            {
                    ps[min_index].start_time = current_time;
                    total_idle_time += (is_first_process==true) ? 0 :
(ps[min_index].start_time -  prev);
                    is_first_process=false;
            }
            bt_remaining[min_index] -= 1;
            current_time++;
            prev=current_time;
            if(bt_remaining[min_index] == 0)
            {
                ps[min_index].ct = current_time;
                ps[min_index].tat = ps[min_index].ct - ps[min_index].at;
                ps[min_index].wt= ps[min_index].tat - ps[min_index].bt;
                ps[min_index].rt = ps[min_index].start_time - ps[min_index].at;

                sum_tat +=ps[min_index].tat;
                sum_wt += ps[min_index].wt;
                sum_rt += ps[min_index].rt;
                completed++;
                is_completed[min_index]=true;
            }
        }
    }

    max_completion_time = INT_MIN;
    min_arrival_time = INT_MAX;
    for(int i=0;i<n;i++)
    {
        max_completion_time = findmax(max_completion_time,ps[i].ct);
        min_arrival_time = findmin(min_arrival_time,ps[i].at);
    }
    length_cycle = max_completion_time - min_arrival_time;

    //Output
    printf("\nProcess No.\tAT\tCPU Burst Time\tCT\tTAT\tWT\tRT\n");
    for(int i=0;i<n;i++)

printf("%d\t\t%d\t%d\t\t%d\t%d\t%d\t%d\n",ps[i].pid,ps[i].at,ps[i].bt,ps[i].ct,ps[i].t
at,ps[i].wt,ps[i].rt);

    printf("\n");
```

```
    cpu_utilization = (float)(length_cycle - total_idle_time)/ length_cycle;

    printf("\nAverage Turn Around time= %f ",(float)sum_tat/n);
    printf("\nAverage Waiting Time= %f ",(float)sum_wt/n);
    printf("\nAverage Response Time= %f ",(float)sum_rt/n);
    printf("\nThroughput= %f",n/(float)length_cycle);
    printf("\nCPU Utilization(Percentage)= %f",cpu_utilization*100);
    return 0;
}
```

Output

```
Enter Process 3 Arrival Time: 4

Enter Process 0 Burst Time: 2

Enter Process 1 Burst Time: 2

Enter Process 2 Burst Time: 4

Enter Process 3 Burst Time: 4

Process No.     AT      CPU Burst Time  CT      TAT     WT      RT
0               0       2               2       2       0       0
1               2       2               4       2       0       0
2               3       4               8       5       1       1
3               4       4               12      8       4       4


Average Turn Around time= 4.250000
Average Waiting Time= 1.250000
Average Response Time= 1.250000
Throughput= 0.333333
CPU Utilization(Percentage)= 100.000000%
```

Program 8

Q. C program to Implement Interprocess Communication using PIPE.
A Pipe is a technique used for inter process communication. A pipe is **a mechanism by which the output of one process is directed into the input of another process**. Thus it provides one way flow of data between two related processes. .... One can write into a pipe from input end and read from the output end.

Source Code:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
   int fd[2],n;
   char buffer[100];
   pid_t p;
   pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]
   p=fork();
   if(p>0) //parent
   {   printf("Parent Passing value to child\n");
       write(fd[1],"hello\n",6);
       //fd[1] is the write end of the pipewait(NULL);
   }
   else // child
   {
      printf("Child printing received value\n");
      n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe
      write(1,buffer,n);
   }
   return 0;
}
```

Output


Parent Passing value to child
Child printing received value
hello

Program 9

Q. C program to Implement Interprocess Communication using shared memory.

Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes. Inter Process Communication method helps to speedup modularity. A semaphore is a signaling mechanism technique.

Source Code:

```
// Writer Process Code

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/ipc.h>

int main() {

    // Writer Process
    key_t key = ftok("MSG", 5);
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT);
    char *ptr = (char*)shmat(shmid, NULL, 0);
    printf("Enter data to put in shared memory: ");
    fgets(ptr, 512, stdin);
    shmdt(ptr);

    return 0;
}
// Reader Process Code

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/shm.h>
#include <sys/ipc.h>

int main() {

    key_t key = ftok("MSG", 5);
```

```
    int shmid = shmget(key, 1024, 0666);
    char *ptr = (char*)shmat(shmid, NULL, 0);
    printf("Data passed by writer process in shared memory is: %s", ptr);
    shmdt(ptr);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
    }
```

Program 10

Q. C program to demonstrate working of execl() where parent process executes "ls" command and child process executes "date" command.

The execl() function **replaces the current process image with a new process image specified by path**. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

Source Code:

```
#include<stdio.h>
#include <unistd.h>
#include<stdlib.h>
#include <sys/types.h>
#include<sys/wait.h>

int main() {

    pid_t pid;
    pid = fork();

    if (pid < 0) {
        printf("Fork Failed\n");
        exit(1);
    } else if (pid == 0) {
        execl("/bin/date", "date", NULL);
    } else {
        execl("/bin/ls", "ls", "-l", NULL);
    }

    return 0;
}
```

Output

```
Sun Nov  7 17:17:02 IST 2021
total 184
-rwxr-xr-x  1 crytek  staff  49552 Nov  7 17:17 a.out
-rw-r--r--  1 crytek  staff    151 Nov  7 12:01 ques1.c
-rw-r--r--  1 crytek  staff    369 Nov  7 17:16 ques10.c
-rw-r--r--  1 crytek  staff    603 Nov  7 13:05 ques2.c
-rw-r--r--  1 crytek  staff    332 Nov  7 13:23 ques3.c
-rw-r--r--  1 crytek  staff   2603 Nov  7 13:35 ques4.c
-rw-r--r--  1 crytek  staff   3916 Nov  7 13:41 ques5.c
-rw-r--r--  1 crytek  staff   1697 Nov  7 15:46 ques6.c
-rw-r--r--  1 crytek  staff   3567 Nov  7 15:51 ques7.c
-rw-r--r--  1 crytek  staff    590 Nov  7 16:08 ques8.c
-rw-r--r--  1 crytek  staff    402 Nov  7 16:31 ques9.c
```

Program 11

Q. C program to Implement Banker"s Algo for Deadlock Avoidance to check for the safe and unsafe state

The Banker algorithm, sometimes referred to as the detection algorithm, is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources.

Source Code:

```
#include <stdio.h>
#include <stdbool.h>

struct process_info
{
   int max[10];
   int allocated[10];
   int need[10];
};

int no_of_process,no_of_resources;

void input(struct process_info process[no_of_process],int
available[no_of_resources])
{
   //Fill array of Structure
   for(int i=0;i<no_of_process;i++)
   {
     printf("Enter process[%d] info\n",i);
     printf("Enter Maximum Need: ");
     for(int j=0;j<no_of_resources;j++)
     scanf("%d",&process[i].max[j]);
     printf("Enter No. of Allocated Resources for this process: ");
     for(int j=0;j<no_of_resources;j++)
```

```
      {
        scanf("%d",&process[i].allocated[j]);
        process[i].need[j]= process[i].max[j] - process[i].allocated[j];
      }
    }
    printf("Enter Available Resources: ");
    for(int i=0;i<no_of_resources;i++)
    {
      scanf("%d",&available[i]);
    }
}

//Print the Info in Tabular Form
void showTheInfo(struct process_info process[no_of_process])
{
    printf("\nPID\tMaximum\t\tAllocated\tNeed\n");
    for(int i=0;i<no_of_process;i++)
    {
        printf("P[%d]\t",i);
        for(int j=0;j<no_of_resources;j++)
          printf("%d ",process[i].max[j]);
        printf("\t\t");
        for(int j=0;j<no_of_resources;j++)
          printf("%d ",process[i].allocated[j]);
        printf("\t\t");
        for(int j=0;j<no_of_resources;j++)
          printf("%d ",process[i].need[j]);
        printf("\n");

    }
}

bool applySafetyAlgo(struct process_info process[no_of_process],int
available[no_of_resources],int safeSequence[no_of_process])
{
   bool finish[no_of_process];
   int work[no_of_resources];
   for(int i=0;i<no_of_resources;i++)
   {
     work[i]=available[i];
```

```
}
for(int i=0;i<no_of_process;i++)
  finish[i]=false;
bool proceed=true;
int k=0;
while(proceed)
{
  proceed=false;
  for(int i=0;i<no_of_process;i++)
  {
      bool flag=true;

      if(finish[i]==false)
      {

        for(int j=0;j<no_of_resources;j++)
        {
          if(process[i].need[j] <= work[j])
          {
            continue;
          }
          else
          {
            flag=false;
            break;
          }
        }
        if(flag==false)
          continue;

       //If we get Index i(or process i), update work
       for(int j=0;j<no_of_resources;j++)
       work[j]=work[j]+ process[i].allocated[j];
       finish[i]=true;
       safeSequence[k++]=i;
       proceed=true;

      }
    }//end of outer for loop
```

```
    } // end of while

    int i;
    for( i=0;i<no_of_process&&finish[i]==true;i++)
    {
      continue;
    }

    if(i==no_of_process)
     return true;
    else
     return false;
}

//Checks if we State is safe or not
bool isSafeState(struct process_info process[no_of_process],int
available[no_of_resources],int safeSequence[no_of_process])
{

   if(applySafetyAlgo(process,available,safeSequence)==true)
     return true;
   return false;

}


int main()
{
   printf("Enter No of Process\n");
   scanf("%d",&no_of_process);
   printf("Enter No of Resource Instances in system\n");
   scanf("%d",&no_of_resources);
   int available[no_of_resources];
   int safeSequence[no_of_process];
   //Create Array of Structure to store Processes's Informations
   struct process_info process[no_of_process];

   printf("***********Enter details of processes*************\n");

   input(process,available);
```

```
    showTheInfo(process);
    if(isSafeState(process,available,safeSequence))
    {

      printf("\nSystem is in SAFE State\n");
      printf("Safe Sequence is: ");
      for(int i=0;i<no_of_process;i++)
        printf("P[%d] ",safeSequence[i]);
    }
    else
      printf("System is in UNSAFE State\n");
    return 0;
}
```

Output

```
Enter Maximum Need: 3
3
2
Enter No. of Allocated Resources for this process: 3
3
2
Enter Available Resources: 6
5
4

PID     Maximum         Allocated       Need
P[0]    5 3 4           2 3 2           3 0 2
P[1]    3 2 1           1 2 3           2 0 -2
P[2]    5 2 4           3 4 1           2 -2 3
P[3]    3 3 2           3 3 2           0 0 0

System is in SAFE State
Safe Sequence is: P[0] P[1] P[2] P[3] %
```

Program 12

Q. C program to Implement First comes first serve page replacement policy.

This is the simplest page replacement algorithm. In this algorithm, **the operating system keeps track of all pages in the memory in a** queue, the 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.

Source Code:

```
#include<stdio.h>

int main()
{
   int i,j,n,a[50],frame[10],no,k,avail,count=0;
   printf("\n ENTER THE NUMBER OF PAGES:\n");
   scanf("%d",&n);
   printf("\n ENTER THE PAGE NUMBER :\n");
   for(i=1;i<=n;i++)
   scanf("%d",&a[i]);

   printf("\n ENTER THE NUMBER OF FRAMES :");
   scanf("%d",&no);
   for(i=0;i<no;i++)
   frame[i]= -1;
   j=0;
   printf("\tref string\t page frames\n");
   for(i=1;i<=n;i++)
   {
      printf("%d\t\t",a[i]);
      avail=0;
      for(k=0;k<no;k++)
      if(frame[k]==a[i])
         avail=1;
```

```
        if (avail==0)
        {
            frame[j]=a[i];
            j=(j+1)%no;
            count++;
            for(k=0;k<no;k++)
            printf("%d\t",frame[k]);
        }
        printf("\n");
    }
        printf("Page Fault Is %d",count);
        return 0;
}
```

Output

```
 ENTER THE NUMBER OF PAGES:
3

 ENTER THE PAGE NUMBER :
1
2
3

 ENTER THE NUMBER OF FRAMES :4
        ref string       page frames
1               1       -1      -1      -1
2               1       2       -1      -1
3               1       2       3       -1
Page Fault Is 3%
```

Program 13

Q. C program to Implement Least recently used page replacement policy

This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called **LRU** (**Least Recently Used**) paging. Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear.

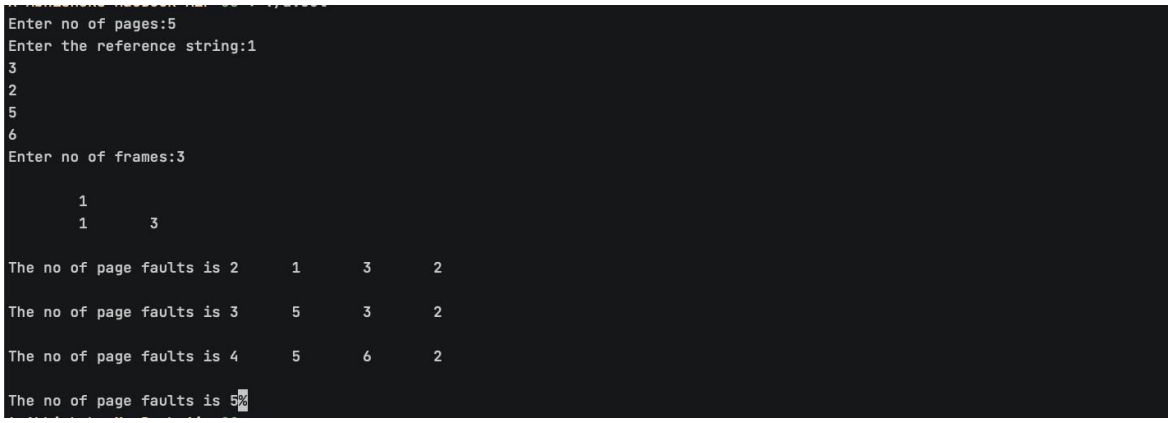
Source Code:

```
#include<stdio.h>
int main()
{
int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];

printf("Enter no of pages:");
scanf("%d",&n);

printf("Enter the reference string:");
for(i=0;i<n;i++)
   scanf("%d",&p[i]);

printf("Enter no of frames:");
   scanf("%d",&f);

q[k]=p[k];
printf("\n\t%d\n",q[k]);
c++;
k++;
for(i=1;i<n;i++)
{
   c1=0;
```

```
for(j=0;j<f;j++)
{
        if(p[i]!=q[j])
        c1++;
}
if(c1==f)
{
   c++;
   if(k<f)
   {
        q[k]=p[i];
        k++;
        for(j=0;j<k;j++)
        printf("\t%d",q[j]);
        printf("\n");
   }
   else
   {
        for(r=0;r<f;r++)
        {
             c2[r]=0;
             for(j=i-1;j<n;j--)
             {
             if(q[r]!=p[j])
             c2[r]++;
             else
             break;
        }
   }
   for(r=0;r<f;r++)
   b[r]=c2[r];
   for(r=0;r<f;r++)
   {
        for(j=r;j<f;j++)
        {
             if(b[r]<b[j])
             {
                  t=b[r];
                  b[r]=b[j];
                  b[j]=t;
```

```
                        }
                }
        }
        for(r=0;r<f;r++)
        {
                if(c2[r]==b[0])
                q[r]=p[i];
                printf("\t%d",q[r]);
        }
        printf("\n");
    }
    }
    printf("\nThe no of page faults is %d",c);
}
    return 0;
}
```

Output


Enter no of pages:5
Enter the reference string:1
3
2
5
6
Enter no of frames:3
1
1 3
The no of page faults is 2 1 3 2
The no of page faults is 3 5 3 2
The no of page faults is 4 5 6 2
The no of page faults is 5%

Program 14

Q. C program to Implement FCFS Disk Scheduling Algorithm.

FCFS (First-Come-First-Serve) is **the easiest disk scheduling algorithm among all the scheduling algorithms**. In the FCFS disk scheduling algorithm, each input/output request is served in the order in which the requests arrive.

Source Code:

```
#include<stdio.h>

int main()
{
   int t[20], n, i, j, tohm[20], tot=0;
   float avhm;

   printf("Enter the no.of tracks: ");
   scanf("%d", &n);

   printf("\nEnter the tracks to be traversed: “);

   for(i=2;i<n+2;i++)
      scanf("%d", &t[i]);

   for(i=1;i<n+1;i++)
   {
      tohm[i]=t[i+1]-t[i]; if(tohm[i]<0)

      tohm[i]=tohm[i]*(-1);
   }

   for(i=1;i<n+1;i++)
      tot+=tohm[i];

   avhm=(float)tot/n;
```

```
    printf("Tracks traversed\tDifference between tracks\n");
    for(i=1;i<n+1;i++)
        printf("%d\t\t\t%d\n",t[i], tohm[i]);
    printf("\nAverage header movements: %f",avhm);
    return 0;
}
```

Output

```
Enter the no.of tracks: 9

Enter the tracks to be traversed: 55 58 60 70 18 90 150 160 184
Tracks traversed        Difference between tracks
1                       54
55                      3
58                      2
60                      10
70                      52
18                      72
90                      60
150                     10
160                     24

Average header movements: 31.888889%
```

Program 15

Q. C program to Implement 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.

Source Code:

```
#include<stdio.h>

int main()
{
   int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum=0;
   printf("\nEnter the no of tracks to be traveresed: ");
   scanf("%d"',&n);
   printf("\nEnter the position of head: ");
   scanf("%d",&h);
   t[0]=0;t[1]=h;
   printf("\nEnter the tracks: ");
   for(i=2;i<n+2;i++)
      scanf("%d",&t[i]);

   for(i=0;i<n+2;i++)
   {
      for(j=0;j<(n+2)-i-1;j++)
      {
         if(t[j]>t[j+1])
         {
            temp=t[j];
            t[j]=t[j+1];
```

```
                t[j+1]=temp;
            }
        }
    }

    for(i=0;i<n+2;i++)
        if(t[i]==h) {
            j=i;
            k=i;
        }

    p=0;
    while(t[j]!=0)
    {
        atr[p]=t[j]; j--;
        p++;
    }
    atr[p]=t[j];
    for(p=k+1;p<n+2;p++,k++)
        atr[p]=t[k+1];

    for(j=0;j<n+1;j++)
    {
        if(atr[j]>atr[j+1])
            d[j]=atr[j]-atr[j+1];
        else
            d[j]=atr[j+1]-atr[j]; sum+=d[j];
    }

    printf("\nAverage header movements:%f",(float)sum/n);

    return 0;
}
```

Output

```
Enter the no of tracks to be traveresed: 9

Enter the position of head: 55

Enter the tracks: 55 58 60 70 18 90 150 160 184

Average header movements:26.555555%
```