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

Term work

on

Operating Systems

(PCS 502)

2022-23

Submitted to:

Ms. Manika Manwal

Asst. Professor
GEHU, D.Dun

Submitted by:

Anant Gupta

University Roll. No.:2018156
Class Roll. No./Section: 15/G

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GRAPHIC ERA HILL UNIVERSITY, DEHRADUN

ACKNOWLEDGMENT

I would like to particularly thank my Operating Systems Lab Faculty **Ms. Manika Manwal** for his patience, support and encouragement throughout the completion of this Term work.

At last, but not the least I greatly indebted to all other persons who directly or indirectly helped me during this course.

Anant Gupta

University. Roll No.- 2018156

B. Tech CSE-G-V Sem

Session: 2021-22

GEHU, Dehradun

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

Table of Contents

	Avoidance to check for the safe and unsafe state.	
12	C program to Implement First In First Out page replacement policy	
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 no-01

Objective: C Program to demonstrate the working of fork () system call

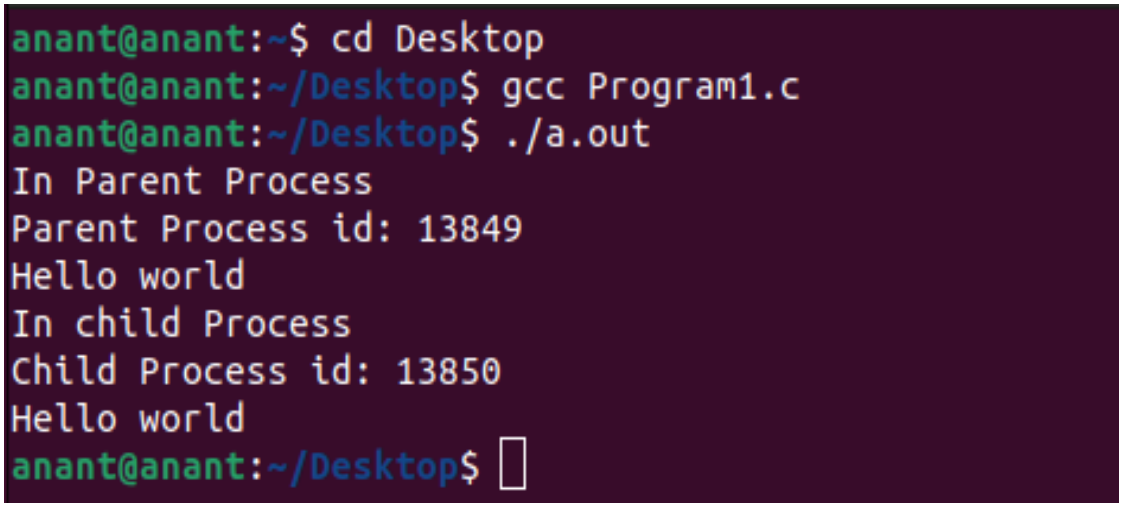
Fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork () system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
It takes no parameters and returns an integer value. Below are different values returned by fork ().

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

Program:

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

{
   int p=fork();
   if(p==0)

   {
      printf("In child Process\n");
      printf("Child Process id: %d\n",getpid());
   }
   else if(p>0)
   {
      printf("In Parent Process\n");
      printf("Parent Process id: %d\n",getpid());
   }
   else
   {
    printf("Error in System call\n");
   }

   printf("Hello world\n");

   return 0;

}
```

Output:


anant@anant:~$ cd Desktop
anant@anant:~/Desktop$ gcc Program1.c
anant@anant:~/Desktop$ ./a.out
In Parent Process
Parent Process id: 13849
Hello world
In child Process
Child Process id: 13850
Hello world
anant@anant:~/Desktop$

Program no-02

Objective: 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.

Fork System Call:

Parent Process: All the processes are created when a process executes the fork() system call except the startup process. The process that executes the fork() system call is the parent process. 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.
- 0 is returned to the child process.

On the failure of a fork() system call,

- -1 is returned to the parent process.
- A child process is not created.

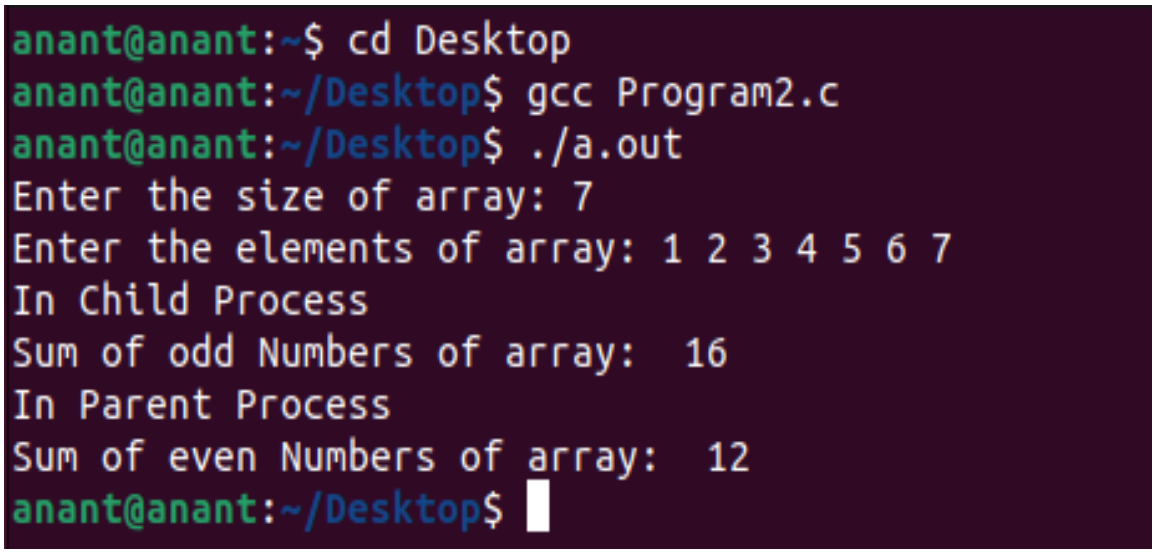
Child Process: A child process is created by a parent process in an operating system using a fork() system call. A child process may also be known as subprocess or a subtask.

Program:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
   int n,sumodd=0,sumeven=0;
   printf("Enter the size of array: ");
   scanf("%d",&n);
   int arr[n];
   printf("Enter the elements of array: ");
   for(int i=0;i<n;i++)
   {
    scanf("%d",&arr[i]);
   }
   int p=fork();
   if(p==0)
   {
      printf("In Child Process\n");
```

```
        for(int i=0;i<n;i++)
        {
            if(arr[i]%2!=0)
            sumodd+=arr[i];
}
           printf("Sum of odd Numbers of array:  %d\n",sumodd);
    }
    else if(p>0)
    {
        sleep(2);
        printf("In Parent Process\n");
        for(int i=0;i<n;i++)
        {
            if(arr[i]%2==0)
            sumeven+=arr[i];
        }
         printf("Sum of even Numbers of array:  %d\n",sumeven);
    }
    else
    {
        printf("Error in System call\n");
    }

    return 0;
}
```

Output:


anant@anant:~$ cd Desktop
anant@anant:~/Desktop$ gcc Program2.c
anant@anant:~/Desktop$ ./a.out
Enter the size of array: 7
Enter the elements of array: 1 2 3 4 5 6 7
In Child Process
Sum of odd Numbers of array: 16
In Parent Process
Sum of even Numbers of array: 12
anant@anant:~/Desktop$

Program no-03

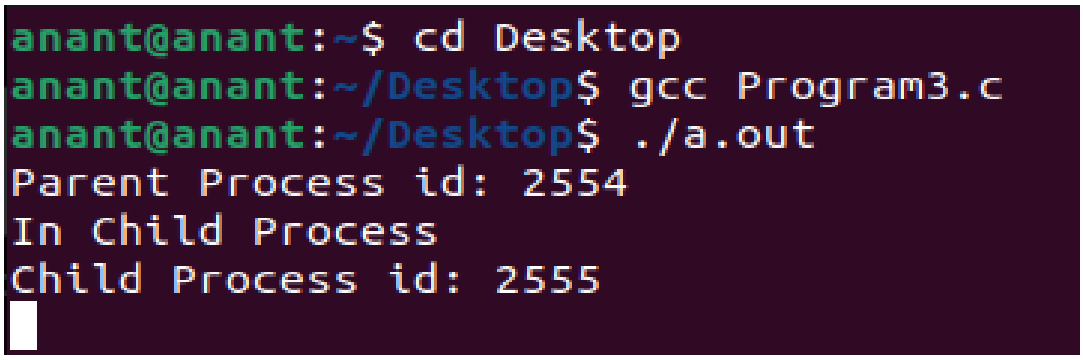
Objective: C program to Implement the Orphan Process and Zombie Process.

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.

Program: Zombie Process

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
int main()
{
   int p=fork();
   if(p==0)
   {
      printf("In Child Process\n");
      printf("Child Process id: %d\n",getpid());
      exit(0);
   }
   else if(p>0)
   {
      printf("Parent Process id: %d\n",getpid());
      sleep(80);
      printf("In Parent Process\n");
   }
   else
   {
      printf("Error in System call\n");
   }
   return 0;
}
```

Output:



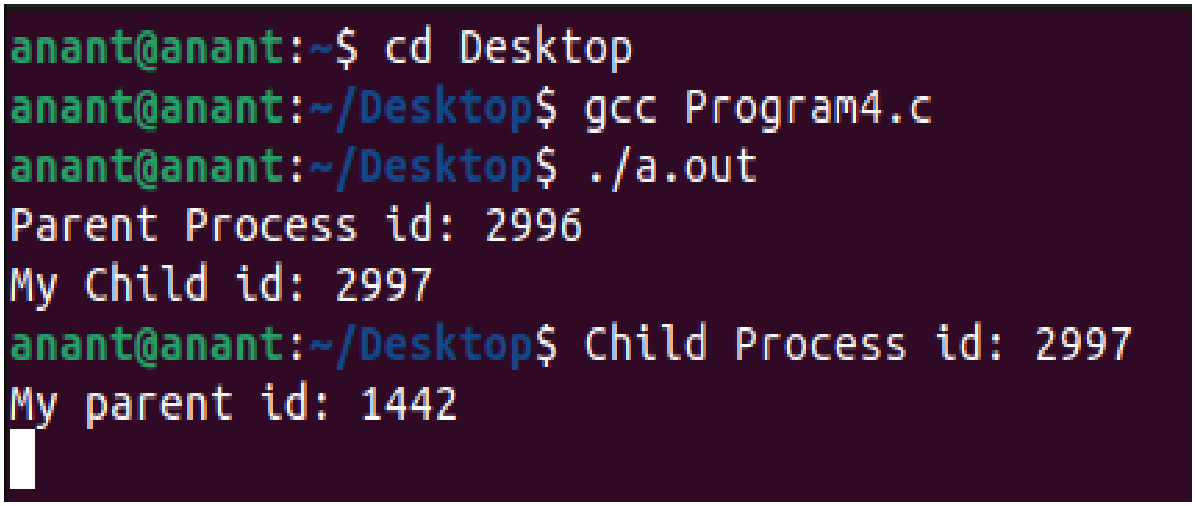
Process 2554 is a Zombie Process

Orphan Process: A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process. In the following code, parent finishes execution and exits while the child process is still executing and is called an orphan process now.

Program: Orphan Process

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
int main()
{
   int p=fork();
   if(p==0)
   {
      sleep(10);
      printf("Child Process id: %d\n",getpid());
      printf("My parent id: %d\n",getppid());
   }
   else if(p>0)
   {
      printf("Parent Process id: %d\n",getpid());
      printf("My Child id: %d\n",p);
   }
   else
   {
      printf("Error in System call\n");
   }
   return 0;
}
```

Output:


anant@anant:~$ cd Desktop
anant@anant:~/Desktop$ gcc Program4.c
anant@anant:~/Desktop$ ./a.out
Parent Process id: 2996
My Child id: 2997
anant@anant:~/Desktop$ Child Process id: 2997
My parent id: 1442

Program no-04

Objective: C program to Implement FCFS CPU Scheduling Algorithm

FCFS (First Come First serve) CPU Scheduling Algorithm: First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.
In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.
Completion Time: Time at which process completes its execution.
Turn Around Time: Time Difference between completion time and arrival time.
Turn Around Time = Completion Time – Arrival Time
Waiting Time (W.T): Time Difference between turn around time and burst time.
Waiting Time = Turn Around Time – Burst Time

Program:

```
#include<stdlib.h>
#include <stdio.h>
struct process
{
   int pid;
   int at;
   int bt;
};
int comparator(const void *p, const void *q)
{
   int l = ((struct process *)p)->at;
   int r = ((struct process *)q)->at;

   int c = ((struct process *)p)->pid;
   int d= ((struct process *)q)->pid;
   if(l==r)
   return (c-d);
   return (l - r);
}
int main()
{
   int n;
   printf("Enter the number of process: ");
   scanf("%d",&n);
   struct process p[n];
   for(int i=0;i<n;i++)
   {
      printf("Process %d\n",i+1);
      printf("Enter the id of process:" );
      scanf("%d",&(p[i].pid));
      printf("Enter the arrival time of process: ");
      scanf("%d",&(p[i].at));
      printf("Enter the Burst time of process: ");
```

```
        scanf("%d",&(p[i].bt));
    }
    qsort((void*)p, n, sizeof(p[0]), comparator);
    int temp=0;
    int ct[n],tat[n],wt[n],rt[n];
    for(int i=0;i<n;i++)
    {
       if(p[i].at<=temp)
       {
          temp+=p[i].bt;
          ct[i]=temp;
          tat[i]=ct[i]-p[i].at;
          wt[i]=tat[i]-p[i].bt;
          rt[i]=wt[i];
       }
       else
       {
          while(temp<p[i].at)
          {
             temp++;

          }
           temp+=p[i].bt;
          ct[i]=temp;
          tat[i]=ct[i]-p[i].at;
          wt[i]=tat[i]-p[i].bt;
          rt[i]=wt[i];
       }
    }
    printf("Pid     AT      BT      CT      TAT     WT      RT\n");
    for(int i=0;i<n;i++)
    {
        printf("P%d      %d      %d      %d      %d      %d
%d\n",p[i].pid,p[i].at,p[i].bt,ct[i],tat[i],wt[i],rt[i]);
    }
    float avgtat,avgwt,avgrt,sum1,sum2,sum3;
    for(int i=0;i<n;i++)
    {
       sum1+=tat[i];
       sum2+=wt[i];
       sum3+=rt[i];
    }
    avgtat=sum1/n;
    avgwt=sum2/n;
    avgrt=sum3/n;
    printf("Average Turn Around Time: %f\n",avgtat);
    printf("Average Waiting Time: %f\n",avgwt);
    printf("Average Response Time: %f\n",avgrt);
    return 0;
}
```

Output:

```
anant@anant:~$ cd Desktop
anant@anant:~/Desktop$ gcc Program5.c
anant@anant:~/Desktop$ ./a.out
Enter the number of process: 4
Process 1
Enter the id of process:1
Enter the arrival time of process: 0
Enter the Burst time of process: 2
Process 2
Enter the id of process:2
Enter the arrival time of process: 1
Enter the Burst time of process: 2
Process 3
Enter the id of process:3
Enter the arrival time of process: 5
Enter the Burst time of process: 3
Process 4
Enter the id of process:4
Enter the arrival time of process: 6
Enter the Burst time of process: 4
Pid       AT       BT      CT     TAT     WT     RT
P1        0        2       2       2       0       0
P2        1        2       4       3       1       1
P3        5        3       8       3       0       0
P4        6        4       12       6       2       2
Average Turn Around Time: 3.500000
Average Waiting Time: 0.750000
Average Response Time: 0.750000
anant@anant:~/Desktop$
```

Program no-05

Objective: C program to implement SRTF algorithm.

SRTF(Shortest Remaining Time First) Scheduling Algorithm: This Algorithm is the **preemptive version** of **SJF scheduling**. 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.Once all the processes are available in the **ready queue**, No preemption will be done and the algorithm will work as **SJF scheduling**. The context of the process is saved in the **Process Control Block** when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the **next execution** of this process.

Program:

```
#include<stdlib.h>
#include <stdio.h>
struct process
{
   int i;
   int pid;
   int at;
   int bt;

};
int comparator(const void *p, const void *q)
{
  int l = ((struct process *)p)->bt;
   int r = ((struct process *)q)->bt;

   int a = ((struct process *)p)->at;
    int b = ((struct process *)q)->at;

   int c = ((struct process *)p)->pid;
   int d= ((struct process *)q)->pid;
   if(l==r)
   {
       if(a==b)
      return c-d;
     return a-b;
   }
   return (l - r);
}
int main()
{
   int n;
   printf("Enter the number of process: ");
   scanf("%d",&n);
   struct process p[n];
```

```
for(int i=0;i<n;i++)
{
    printf("Enter the process number:");
    scanf("%d",&p[i].i);
    printf("Enter the id of process:" );
    scanf("%d",&(p[i].pid));
    printf("Enter the arrival time of process: ");
    scanf("%d",&(p[i].at));
    printf("Enter the Burst time of process: ");
    scanf("%d",&(p[i].bt));
}
int at[n],bt[n],pid[n];
 for(int i=0;i<n;i++)
 {
    bt[i]=p[i].bt;
    at[i]=p[i].at;
    pid[i]=p[i].pid;
 }
 qsort((void*)p, n, sizeof(p[0]), comparator);
 int temp=0;
 int ct[n],tat[n],wt[n],rt[n];
 int comp[n];
 for(int i=0;i<n;i++)
 comp[i]=0;
 int count=0;


 while(count!=n)
 {
 for(int i=0;i<n;i++)
 {
    if(comp[p[i].i-1]==1)
    continue;
    else if(p[i].at<=temp)
    {
        if(p[i].bt==bt[p[i].i-1])
        {   if(temp>0)
            rt[p[i].i-1]=temp-at[p[i].i-1];
            else
            rt[p[i].i-1]=temp;
        }
        temp+=1;
        p[i].bt-=1;
        if(p[i].bt==0)
        {
        ct[p[i].i-1]=temp;
        tat[p[i].i-1]=ct[p[i].i-1]-at[p[i].i-1];
        wt[p[i].i-1]=tat[p[i].i-1]-bt[p[i].i-1];
        comp[p[i].i-1]=1;
        count++;
```

```
    }
     break;
}
else
{
    int t=i,tim,c=0;
     for(int j=i+1;j<n;j++)
    {
        if(comp[p[j].i-1]==0&&p[j].at<=temp)
        {
            t=j;
            c=1;
            break;
        }
    }
    if(c!=1)
    {
        tim=p[0].at;
        t=0;
    for(int j=1;j<n;j++)
    {
        if(comp[p[j].i-1]==0&&p[j].at<tim)
        {
            t=j;
            tim=p[j].at;
        }
    }

    while(temp<p[t].at)
    {
        temp++;
    }
    }
    if(p[t].bt==bt[p[t].i-1])
    {   if(temp>0)
        rt[p[t].i-1]=temp-at[p[t].i-1];
        else
        rt[p[t].i-1]=temp;
    }
     temp+=1;
     p[t].bt-=1;
     if(p[t].bt==0)
     {
    ct[p[t].i-1]=temp;
    tat[p[t].i-1]=ct[p[t].i-1]-at[p[t].i-1];
    wt[p[t].i-1]=tat[p[t].i-1]-bt[p[t].i-1];
    count++;
    comp[p[t].i-1]=1;
     }
     break;
```

```
        }
    }
    qsort((void*)p, n, sizeof(p[0]), comparator);
    }
    printf("Pid     AT      BT      CT      TAT     WT      RT\n");
    for(int i=0;i<n;i++)
    {
       printf("P%d      %d      %d      %d      %d      %d
%d\n",pid[i],at[i],bt[i],ct[i],tat[i],wt[i],rt[i]);
    }
    float avgtat,avgwt,avgrt,sum1,sum2,sum3;
    for(int i=0;i<n;i++)
    {
      sum1+=tat[i];
      sum2+=wt[i];
      sum3+=rt[i];
    }
    avgtat=sum1/n;
    avgwt=sum2/n;
    avgrt=sum3/n;
    printf("Average Turn Around Time: %f\n",avgtat);
    printf("Average Waiting Time: %f\n",avgwt);
    printf("Average Response Time: %f\n",avgrt);

   return 0;
}
```

Output:

```
anant@anant:~$ cd Desktop
anant@anant:~/Desktop$ gcc Program6.c
anant@anant:~/Desktop$ ./a.out
Enter the number of process: 4
Enter the process number:1
Enter the id of process:1
Enter the arrival time of process: 0
Enter the Burst time of process: 5
Enter the process number:2
Enter the id of process:2
Enter the arrival time of process: 1
Enter the Burst time of process: 3
Enter the process number:3
Enter the id of process:3
Enter the arrival time of process: 2
Enter the Burst time of process: 4
Enter the process number:4
Enter the id of process:4
Enter the arrival time of process: 4
Enter the Burst time of process: 1
Pid        AT        BT       CT      TAT      WT      RT
P1         0         5        9        9        4         0
P2         1         3        4        3        0         0
P3         2         4        13        11        7         7
P4         4         1        5        1        0         0
Average Turn Around Time: 6.000000
Average Waiting Time: 2.750000
Average Response Time: 1.750000
anant@anant:~/Desktop$
```

Program no-06

Objective: C program to Implement Round Robin CPU scheduling algo.

Round Robin CPU scheduling Algorithm: **Round Robin** is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is basically the preemptive version of First come First Serve CPU Scheduling algorithm.
Round Robin CPU Algorithm generally focuses on Time Sharing technique.
The period of time for which a process or job is allowed to run in a pre-emptive method is called time **quantum**.
Each process or job present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will **end** else the process will go back to the **waiting table** and wait for the its next turn to complete the execution.

Program:

```
#include <stdio.h>
#include <stdlib.h>
# define SIZE 100
struct process
{
   int pid;
   int at;
   int bt;
};
void enqueue(int);
int dequeue();
int arr[SIZE];
int Rear = - 1;
int Front = - 1;

int comparator(const void *p, const void *q)
{
   int l = ((struct process *)p)->at;
   int r = ((struct process *)q)->at;

   int c = ((struct process *)p)->pid;
   int d= ((struct process *)q)->pid;
   if(l==r)
   return (c-d);
   return (l - r);
}

int main()
{

   int n,timequanta;
   printf("Enter the number of process: ");
   scanf("%d",&n);
   struct process p[n];
```

```
for(int i=0;i<n;i++)
{
    printf("process number:%d\n",i+1);
    printf("Enter the id of process:" );
    scanf("%d",&(p[i].pid));
    printf("Enter the arrival time of process: ");
    scanf("%d",&(p[i].at));
    printf("Enter the Burst time of process: ");
    scanf("%d",&(p[i].bt));
}
printf("Enter the timequanta:");
scanf("%d",&timequanta);
int bt[n];

qsort((void*)p, n, sizeof(p[0]), comparator);
int temp=0,count=0,comp[n];
for(int i=0;i<n;i++)
{
    comp[i]=0;
    bt[i]=p[i].bt;
}
int ct[n],tat[n],wt[n],rt[n],i=0;
while(count<n)
{

   if(Front==-1)
   {

      if(p[i].at<=temp)
       {
        if(p[i].bt==bt[i])
        {
           if(temp!=0)
           rt[i]=temp-p[i].at;
           else
           rt[i]=temp;
        }
       if(p[i].bt>=timequanta)
        {
           temp+=timequanta;
           p[i].bt-=timequanta;
        }
       else
        {
           temp+=p[i].bt;
           p[i].bt=0;
        }
       if(p[i].bt==0)
        {
           ct[i]=temp;
```

```
        tat[i]=ct[i]-p[i].at;
        wt[i]=tat[i]-bt[i];
        comp[i]=1;
        count++;
     }
     int c=i;
    for(int j=i+1;j<n;j++)
    {
        if(p[j].at<=temp)
        {
            enqueue(j);
            i++;
        }
        else break;
    }

    if(p[c].bt!=0)
    {
        enqueue(c);
    }

   }
  else
  {   int t=i;

         while(temp<p[t].at)
         {
             temp+=1;
         }
         break;

  }
}

else
{   int k;
    while((k=dequeue())!=-1)
    {


    if(p[k].bt==bt[k])
    {
        if(temp!=0)
        rt[k]=temp-p[k].at;
        else
        rt[k]=temp;
    }
    if(p[k].bt>=timequanta)
    {
        temp+=timequanta;
```

```
            p[k].bt-=timequanta;
          }
          else
          {
            temp+=p[k].bt;
            p[k].bt=0;
          }
          if(p[k].bt==0)
          {
            ct[k]=temp;
            tat[k]=ct[k]-p[k].at;
            wt[k]=tat[k]-bt[k];
            comp[k]=1;
            count++;
          }

           int c=k;

          for(int j=i+1;j<n;j++)
          {
            if(p[j].at<=temp)
            {
              enqueue(j);
              i++;
            }
            else
            break;
          }
          if(p[c].bt!=0)
          {
            enqueue(c);
          }
          }
        }

    }
    printf("Pid     AT      BT      CT      TAT     WT      RT\n");
    for(int i=0;i<n;i++)
    {
       printf("P%d       %d       %d       %d       %d       %d
%d\n",p[i].pid,p[i].at,bt[i],ct[i],tat[i],wt[i],rt[i]);
    }
     float avgtat,avgwt,avgrt,sum1,sum2,sum3;
     for(int i=0;i<n;i++)
     {
       sum1+=tat[i];
       sum2+=wt[i];
       sum3+=rt[i];
     }
     avgtat=sum1/n;
```

```
    avgwt=sum2/n;
    avgrt=sum3/n;
    printf("Average Turn Around Time: %f\n",avgtat);
    printf("Average Waiting Time: %f\n",avgwt);
    printf("Average Response Time: %f\n",avgrt);


   return 0;
}

void enqueue(int insert_item)
{
   if (Rear == SIZE - 1)
      printf("Overflow \n");
   else
   {
      if (Front == - 1)
      Front = 0;
      Rear = Rear + 1;
      arr[Rear] = insert_item;
   }
}

int dequeue()
{
   if (Front == - 1 || Front > Rear)
   {
      printf("Underflow \n");
      return -1;
   }
   else
   {
      int temp=arr[Front];
      Front = Front + 1;
      return temp;
   }

}
```

Output:

```
anant@anant:~$ cd Desktop
anant@anant:~/Desktop$ gcc Program7.c
anant@anant:~/Desktop$ ./a.out
Enter the number of process: 4
process number:1
Enter the id of process:1
Enter the arrival time of process: 0
Enter the Burst time of process: 5
process number:2
Enter the id of process:2
Enter the arrival time of process: 1
Enter the Burst time of process: 4
process number:3
Enter the id of process:3
Enter the arrival time of process: 2
Enter the Burst time of process: 2
process number:4
Enter the id of process:4
Enter the arrival time of process: 4
Enter the Burst time of process: 1
Enter the timequanta:2
Underflow
Pid        AT        BT       CT      TAT      WT      RT
P1         0         5         12        12         7          0
P2         1         4         11        10         6          1
P3         2         2         6         4         2          2
P4         4         1         9         5         4          4
Average Turn Around Time: 7.750000
Average Waiting Time: 4.750000
Average Response Time: 1.750000
anant@anant:~/Desktop$
```

Program no-07

Objective: C program to Implement Preemptive Priority CPU scheduling algo

Preemptive Priority CPU scheduling Algorithm: **Preemptive Priority CPU Scheduling Algorithm** is a pre-emptive method of CPU scheduling algorithm that works **based on the priority** of a process. In this algorithm, the scheduler schedules the tasks to work as per the priority, which means that a higher priority process should be executed first. In case of any conflict, i.e., when there is more than one process with equal priorities, then the pre-emptive priority CPU scheduling algorithm works on the basis of FCFS (First Come First Serve) algorithm.

Program:

```
#include<stdlib.h>
#include <stdio.h>
struct process
{
   int i;
   int pid;
   int pt;
   int at;
   int bt;
};
int comparator(const void *p, const void *q)
{
  int l = ((struct process *)p)->pt;
   int r = ((struct process *)q)->pt;

   int a = ((struct process *)p)->at;
   int b= ((struct process *)q)->at;

   int c= ((struct process *)p)->pid;
   int d= ((struct process *)q)->pid;

   if(l==r)
   {
      if(a==b)
      return (c-d);
      return (a-b);
   }
   return (r-l);
}
int main()
{
```

```
int n;
printf("Enter the number of process: ");
scanf("%d",&n);
struct process p[n];
for(int i=0;i<n;i++)
{
   printf("Enter the process number:");
   scanf("%d",&p[i].i);
   printf("Enter the id of process:" );
   scanf("%d",&(p[i].pid));
   printf("Enter the priority of the process:");
   scanf("%d",&p[i].pt);
   printf("Enter the arrival time of process: ");
   scanf("%d",&(p[i].at));
   printf("Enter the Burst time of process: ");
   scanf("%d",&(p[i].bt));
}
int at[n],bt[n],pid[n],pt[n];
 for(int i=0;i<n;i++)
 {
   bt[i]=p[i].bt;
   at[i]=p[i].at;
   pid[i]=p[i].pid;
   pt[i]=p[i].pt;
 }
 qsort((void*)p, n, sizeof(p[0]), comparator);
 int temp=0;
 int ct[n],tat[n],wt[n],rt[n];
 int comp[n];
 for(int i=0;i<n;i++)
 comp[i]=0;
 int count=0;
 while(count!=n)
 {
 for(int i=0;i<n;i++)
 {
   if(comp[p[i].i-1]==1)
   continue;
   else if(p[i].at<=temp)
   {
      if(p[i].bt==bt[p[i].i-1])
      {   if(temp>0)
          rt[p[i].i-1]=temp-at[p[i].i-1];
          else
```

```
        rt[p[i].i-1]=temp;
    }
    temp+=1;
    p[i].bt-=1;
    if(p[i].bt==0)
    {
    ct[p[i].i-1]=temp;
    tat[p[i].i-1]=ct[p[i].i-1]-at[p[i].i-1];
    wt[p[i].i-1]=tat[p[i].i-1]-bt[p[i].i-1];
    comp[p[i].i-1]=1;
    count++;
    }
     break;
}
else
{
    int t=i,tim,c=0;
     for(int j=i+1;j<n;j++)
    {
         if(comp[p[j].i-1]==0&&p[j].at<=temp)
        {
            t=j;
            c=1;
            break;
        }
    }
    if(c!=1)
    {
        tim=p[0].at;
        t=0;
    for(int j=1;j<n;j++)
    {
         if(comp[p[j].i-1]==0&&p[j].at<tim)
        {
            t=j;
            tim=p[j].at;
        }
    }

    while(temp<p[t].at)
    {
        temp++;
    }
    }
```

```
            if(p[t].bt==bt[p[t].i-1])
            {   if(temp>0)
                rt[p[t].i-1]=temp-at[p[t].i-1];
                else
                rt[p[t].i-1]=temp;
            }
             temp+=1;
             p[t].bt-=1;
             if(p[t].bt==0)
             {
            ct[p[t].i-1]=temp;
            tat[p[t].i-1]=ct[p[t].i-1]-at[p[t].i-1];
            wt[p[t].i-1]=tat[p[t].i-1]-bt[p[t].i-1];
            count++;
            comp[p[t].i-1]=1;
             }
             break;
          }
       }
       qsort((void*)p, n, sizeof(p[0]), comparator);
       }
       printf("Pid     PT     AT     BT     CT     TAT     WT     RT\n");
       for(int i=0;i<n;i++)
       {
          printf("P%d     %d     %d     %d     %d     %d     %d
%d\n",pid[i],pt[i],at[i],bt[i],ct[i],tat[i],wt[i],rt[i]);
       }
       float avgtat,avgwt,avgrt,sum1,sum2,sum3;
       for(int i=0;i<n;i++)
       {
         sum1+=tat[i];
         sum2+=wt[i];
         sum3+=rt[i];
       }
       avgtat=sum1/n;
       avgwt=sum2/n;
       avgrt=sum3/n;
       printf("Average Turn Around Time: %f\n",avgtat);
       printf("Average Waiting Time: %f\n",avgwt);
       printf("Average Response Time: %f\n",avgrt);
      return 0;
}
```

Output:

```
anant@anant:~$ cd Desktop
anant@anant:~/Desktop$ gcc Program8.c
anant@anant:~/Desktop$ ./a.out
Enter the number of process: 4
Enter the process number:1
Enter the id of process:1
Enter the priority of the process:10
Enter the arrival time of process: 0
Enter the Burst time of process: 5
Enter the process number:2
Enter the id of process:2
Enter the priority of the process:20
Enter the arrival time of process: 1
Enter the Burst time of process: 4
Enter the process number:3
Enter the id of process:3
Enter the priority of the process:30
Enter the arrival time of process: 2
Enter the Burst time of process: 2
Enter the process number:4
Enter the id of process:4
Enter the priority of the process:40
Enter the arrival time of process: 4
Enter the Burst time of process: 1
Pid       PT       AT       BT       CT      TAT      WT      RT
P1       10       0        5        12       12        7         0
P2       20       1        4        8        7        3         0
P3       30       2        2        4        2        0         0
P4       40       4        1        5        1        0         0
Average Turn Around Time: 5.500000
Average Waiting Time: 2.500000
Average Response Time: 0.000000
anant@anant:~/Desktop$
```