Aim: To implement FCFS CPU scheduling algorithm in C.

Algorithm:

1. Start
2. Declare the following array to maximum size 15
   a. wt[max] to store waiting time of each process.
   b. bt[max] to store burst time of each process.
   c. tt[max] to store turnaround time of each process
3. Read the number of processes.
4. Read the burst time for all the processes.
5. Calculate waiting time : wt[i] = wt[i-1] + bt[i-1]
6. Calculate turnaround time : tt[i] = wt[i] + bt[i]
7. Calculate average waiting time and turnaround time
8. Display all the values
9. Stop.

Program:

```
#include<stdio.h>
void main()
{ int p, bt[10], wt[10]=0, tat[10], totalwt=0, totaltat=0;

printf("Enter no of process");
scanf("%d",&p);

printf("Enter burst time");
for(int i=0; i<p; i++)
scanf("%d",&bt[i]);
```

```
wt[0]=0;

for(int i=1 ; i<n ; i++)
  wt[i] = bt[i-1]+wt[i-1];

for( int i=0 ; i<n; i++)
   tat[i]= bt[i]+wt[i];

printf(" Process  Burst time    waiting time   Turnaround time");

for(int i=0; i<n;i++)
{
    totalwt = totalwt+wt[i];
    totaltat = totaltat + tat[i];
    printf(" P%d",(i+1));
    printf(" %d", bt[i]);
    printf(" %d", wt[i]);
    printf(" %d", tat[i]);
}
printf(" Average waiting time = %d",(totalwt/p));
printf("\n Average turnaround time= %d", (totaltat/p));
}
```

Aim: To implement SJF CPU scheduling algorithm in C

Algorithm:

1. Start
2. Declare process structure with following members
   a. PID: to store process ID
   b. WT: to store waiting time of process.
   c. BT: to store burst time of process
   d. TT: to store turnaround time of process.
3. Read the number of processes.
4. Read burst time and PID for all the processes
5. Sort the array of process structure based on burst time
6. Calculate the waiting time of each process.
   a. After storing first process in the list will have waiting time 0

      i. Proc[0].wt = 0

   b. Find waiting time for all other process.

      i. Proc[i].wt = Proc[i-1].wt + Proc[i-1].bt.

7. Calculate the turnaround time of each process.

   i. Proc[i].tt = Proc[i].wt + Proc[i].bt

8. Calculate average waiting time = total waiting time / no. of process
9. Calculate average turnaround time = total turnaround time / no. of process.
10. Display all the values
11. Stop

MGIT

Program:

```
# include <stdio.h>
# define MAX 15
struct process
 {
   int pid, bt, wt, tt;
 };
 int main()
{
struct process proc[MAX], temp;
 int n, i, j;
int total_wt=0, total_tt=0;
float avg_wt, avg_tt;
 printf("Enter no. of process");
 scanf("%d", &n);
 printf("Enter PID and Burst time of the process");
 for(i=0;i<n;i++)
  { scanf("%d %d", &proc[i].pid, &proc[i].bt);

//Bubble sort

for(i=0;i<n-1;i++)
 for(j=0;j<n-i-1;j++)
 if(proc[j].bt > proc[j+1].bt)
  { temp = proc[j];
    proc[j] = proc[j+1];
    proc[j+1] = temp; }
```

```
for(i=0;i<n;i++)
{
  if(i==0)
  {
    proc[0].wt=0
    proc[0].tt=proc[0].bt;
  }
else
{
  proc[i].wt=proc[i-1].wt+proc[i-1].bt;
      total_wt+=proc[i].wt;
  proc[i].tt=proc[i].wt+proc[i].bt;
      total_tt+=proc[i].tt;
  }
}
  avg_wt=(float)total_wt/n;
  avg_tt=(float)total_tt/n;
printf("PID  Burst time  Waiting time  Turnaround time\n");
for(i=0;i<n;i++)
printf("%d %d %d %d\n",proc[i].pid,proc[i].bt,proc[i].wt,
  proc[i].tt);
  printf("Average waiting time and Turnaround time are:%f %f",avg_wt,
                                                    avg_tt);
return 0;
}
```

Output:

Enter no of process: 4

Enter PID and Burst time of the process:

2 45
3 56
1 78
4 7

PID	Burst time	Waiting time	Turnaround time
4	7	0	7
3	45	7	52
2	56	52	108
1	78	108	186

Aim: To implement Round Robin CPU scheduling algorithm in C

Algorithm:

1. Start
2. Declare process structure with Pid, wt, bt, tt, rem_bt.
3. Read number of process.
4. Read the burst time and process id for all process.
5. Read time quantum value.
6. The process are selected from the queue in FCFS order and following steps are repeated until remaining burst time for all process is zero.

i) If burst time > time quantum

The process is preempted after time quantum and added to the end of the queue. Remaining burst time is recorded. CPU execution time is also recorded.

a. t = t + quantum

b. Proc[i].rem_bt = Proc[i].rem_bt - quantum.

ii) If burst time <= time quantum

Process remaining burst time is set to zero and waiting time is calculated.

a. t = t + Proc[i].rem_bt

b. Proc[i].rem_bt = 0

c. Proc[i].wt = t - Proc[i].bt

7. Calculate turnaround time of each process.

Proc[i].tt = Proc[i].wt + Proc[i].bt

8. Calculate and display average waiting time, average turnaround time

9. Stop.

MGIT

Program:

```
#include <stdio.h>
#define MAX 15
struct process
{
  int pid, bt, wt, tt, rem_bt;
};
int main()
{
  struct process proc[MAX];
  int n, i, j, quantum, t;
  int total_wt = 0, total_tt = 0;
  float avg_wt, avg_tt;
  int done = 0;
  printf("Enter pid no and bt of process");
  scanf("%d", &n);
  printf("Enter pid and burst time of the process");
  for(i=0; i<n; i++)
  {
    scanf("%d%d", &proc[i].pid, &proc[i].bt);
    proc[i].rem_bt = proc[i].bt;
  }
  printf("Enter time quantum");
  scanf("%d", &quantum);
  t = 0;
// keep traversing process in RR manner until all of them are not done.
  printf("Order of process Execution is");
```

```
do
{
  done=1;
  for(int i=0; i<n; i++)  // Traverse all processes one by one repeatedly
  {
// Burst time should greater 0 to get executed.
    if(proc[i].rem_bt>0)
    {
      done=0;   // There is a pending proces.
      printf("%d:", proc[i].pid);
      if(proc[i].rem_bt>quantum)
      {
          t+=quantum;
        proc[i].rem_bt=proc[i].rem_bt-quantum;
      }
      else
      {
        t=t+proc[i].rem_bt;
        proc[i].wt=t-proc[i].bt;
        proc[i].rem_bt=0;
      }
    }
  }
} while(done!=1)
printf("\n");
for(i=0;i<n;i++)
  total_wt=total_wt+proc[i].wt;
  proc[i].tt=proc[i].wt+proc[i].bt;
```

```
    total_tt += proc[i].tt;
}
avg_wt = (float) total_wt/n;
avg_tt = (float) total_tt/n;
printf("\nPID\t BT \t WT \t TAT \n");
for(i=0;i<n;i++)
printf("\n %d \t %d \t %d \t %d \n", proc[i].pid, proc[i].bt, proc[i].wt,
proc[i].tt);
printf("\nAverage waiting time %f \n", avg_wt);
printf("\nAverage TAT time %f \n", avg_tt);
return 0;
}
```

Output :

Enter no. of proces : 4

Enter pid and burst time of the proces

1	4
2	3
3	5
4	6

Enter time Quantum : 2

Order of proces Execution is 1 : 2 : 3 : 4 : 1 : 2 : 3 : 4 : 3 : 4 :

Processes are executed in following order.

PID	BT	WT	TAT
1	4	6	10
2	3	8	11
3	5	11	16
4	6	12	18

Average waiting time 9.250000

Average TAT time 13.750000

Aim: To implement priority CPU scheduling algorithm in C.

Algorithm:

1. Start
2. Declare process structure with pid, priority, wt, bt, tt
3. Read the no. of processes
4. Read the pid, burst time and priority for all the processes
5. Sort the array of process structure based on priority
6. Calculate the waiting time of each process.
   a. After sorting first process in the list will have waiting time 0.
      i. proc[0].wt = 0.
   b. Find waiting time for all other process
      i. proc[i].wt = proc[i-1].wt + pro[i-1].bt
7. Calculate the turnaround time of each process
   i. proc[i].tt = proc[i].wt + pro[i].bt
8. Calculate average waiting and turnaround time and display all the values
9. Stop.

Program:

```
#include <stdio.h>
#define MAX 15
struct process
{
    int pid;
    int priority;
    int bt;
    int wt; int tt; };
```

```
int main()
{
 struct process proc[MAX], temp;
 int n, i, j;
 int total_wt=0, total_tt=0;
 float avg_wt, avg_tt;
 printf("Enter no. of process");
 scanf("%d", &n);
 printf("Enter pid and burst time and priority of the process");
  for(i=0; i<n; i++)
  {
    scanf("%d %d %d", &proc[i].pid, &proc[i].bt, &proc[i].priority);
  }
 //Bubble sort, to sort the process
  for(i=0; i<n-1; i++)
   for(j=0; j<n-i-1; j++)
    if(proc[j].priority > proc[j+1].priority)
    {
      temp = proc[j];
      proc[j]= proc[j+1];
      proc[j+1]= temp;
    }
    for(i=0; i<n; i++)
    {
      If(i==0)
      {
        proc[0].wt = 0;
        proc[0].tt = proc[0].bt;
      }
```

```
else
{
  proc[i].wt = proc[i-1].wt + proc[i-1].tt;
  total_wt += proc[i].wt;
  proc[i].tt = proc[i].wt + proc[i].bt;
// total turnaround
  total_tt += proc[i].tt;
  }
}
avg_wt = (float) total_wt/n;
avg_tt = (float) total_tt/n;
printf("\nPid | priority | Burst time | waiting time | turnaround time |\n");
for(i=0;i<n;i++)
printf("\n %d | %d | %d | %d | %d |\n", proc[i].pid, proc[i].priority,
proc[i].bt, proc[i].wt, proc[i].tt);
return 0;
}
```

Output:

Enter no. of process: 4

Enter pid, burst time and priority

3 55 6

4 5 9

3 10 2

1 20 1

pid	priority	B.T	W.T	TAT
1	1	20	0	20
3	2	10	20	30
3	6	55	50	105
4	9	5	155	160

Aim: To implement IPC using pipes

Algorithm:

1. Create a pipe
2. Create a child process
3. Parent process writes to the pipe
4. Child process retrieves the message from pipe and writes it to the standard output
5. Repeat step 3 and step 4 once again

Program:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
  int fd[2],n;
  char buffer[100];
  pid_t p;
  pipe(fd);
  p=fork();
  if(p>0)
  {
    printf("Parent passing value to child");
    write(fd[1],"hello\n",6);
    printf("waiting child to complete");
    wait(0);
  }printf("End of parent process");
```

```
else
{
 printf ("child printing recieved val");
 n = read( fd[0], buffer, 100);
 write(1, buffer, n);
 }
}
```

Output:

```
Parent passing value to child
waiting child to complete
child printing recieved val.
Hello
End of parent process.
```

Aim: To implement IPC using fifo's (Named pipes)

Algorithm:

Program:

```
// fifo1.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{
   int fd;
   fd = mkfifo ("fifo1", 0777);
   printf ("Named pipe created");
}
```

```
// fifo2.c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{
  int fd, n;
```

```
    fd = open("fifo1", O_WRONLY);
    write(fd, "Welcome", 100);
    printf("Sender process %d sent the data", getpid());
}

//fifo3.c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{
    int fd;
    char buffer[100];
    fd = open("fifo1", O_RDONLY);
    read(fd, buffer, 100);
    printf("Reader process %d started \n", getpid());
    printf("Data Recieved by reciever %d is: %s \n", getpid(), buffer);
}
```

Output:

Named piped created

Sender process 3971 sent the data

Reader process 3972 started

Data Recieved by reciever 3972 is: Welcome

Aim: To implement IPC using shared memory technique

Program:

```
// header.h
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

// shm_write.c
#include "header.h"
int main()
{
  int shmid;
  char str[100];
  void *sh;
 shmid= shmget((key_t) 6, 1024, IPC_CREAT|0666);
   sh = shmat(shmid, NULL, 0);
   printf("Enter data to shared memory");
   read(0, str, 100);
   strcpy(sh, str);
   printf("Data written to shared memory is %s", (char*) sh);
   return 0;
}
```

```
// shm_read.c
#include "header.h"
int main()
{
  int shmid;
  char str[100];
  void *sh;
 shmid = shmget (Key_t) 6, 1024, IPC_CREAT |0666);
 sh = shmat (shmid, NULL, 0);
 printf (" Data recieved from shared memory is %s", (char*) sh);
 return 0;
}
```

Output:

Enter data to shared memory : operating.
Data recieved from shared memory is operating.