PROGRAM 1

Develop a C program to implement the Process System Calls (fork(), exec(), wait(), Create process, terminate process).

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.

In general, system calls are required in the following situations:

- If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.
- Creation and management of new processes.
- Network connections also require system calls. This includes sending and receiving packets.
- Access to a hardware device such as a printer, scanner etc. requires a system call.

Types of System Calls

There are mainly five types of system calls:

System Call Type	Description
Process Control	These system calls deal with processes such as process creation, process termination etc.
File Management	These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.
Device Management	These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.
Information Maintenance	These system calls handle information and its transfer between the operating system and the user program.
Communication	These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

System Calls used in this program

1. **fork ():** Used to create new process. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.
   Syntax: fork ();

2. **wait ():**The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.
   Syntax: wait (NULL);

3. **exit ():**A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).
   Syntax: exit (0);

4. **exec():**In execl() function, the parameters of the executable file is passed to the function as different arguments. With execv(), you can pass all the parameters in a NULL terminated array **argv**. The first element of the array should be the path of the executable file. Otherwise, execv() function works just as execl() function. If execv is successful it replaces the current process image with the one to be started and does not return. If execv returns then it failed.
   Syntax: int execv (const char *path, char *const argv[]);

5. **getppid():** returns the process ID of the parent of the calling process.

6. **getpid():** returns the process ID of the calling process.

Note: These system calls declarations are present in **unistd.h** library.

PROGRAM1.1 Fork () System Call

DESCRIPTION: Used to create new processes. The new process consists of a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

Syntax: fork( );

PSEUDOCODE:

```
1. Start the program.

2. Declare two integer variables `pid` and `childId`.

3. Get the current process ID using `getpid()` and store it in `pid`.

4. Use the `fork()` system call to create a new process, and store the returned value in
   `childId`
5. If `childId > 0` (i.e., this is the parent process):
   a. Print "I am in the parent process" and the value of `pid`.
   b. Print "I am in the parent process" and the process ID using `getpid()`.
   c. Print "I am in the parent process" and the parent process ID using `getppid()`.

6. Else (i.e., this is the child process):
   a. Print "I am in the child process" and the value of `pid`.
   b. Print "I am in the child process" and the process ID using `getpid()`.
   c. Print "I am in the child process" and the parent process ID using `getppid()`.

7. End the program.
```

SOURCE CODE:

```
/* fork system call */
#include<stdio.h>
#include <unistd.h>
#include<sys/types.h>
int main()
{
   int pid,childId;
   pid=getpid();
   fflush(stdout);
   if((childId=fork())>0)
   {
      printf("\n I am in the parent process %d",pid);
      printf("\n I am in the parent process %d",getpid());
      printf("\n I am in the parent process %d\n",getppid());
   }
   else
   {
      printf("\n I am in child process %d",pid);
      printf("\n I am in the child process %d",getpid());
      printf("\n I am in the child process %d",getppid());
   }
}
```

OUTPUT:

```
$ vi fork.c
$ cc fork.c
$ ./a.out
I am in the parent process 255
I am in the parent process 255
I am in the parent process 254
I am in child process 255
I am in the child process 259
I am in the child process 255
```

RESULT:

Thus, the program was executed and verified successfully.

PROGRAM1.2 Wait() System Call

DESCRIPTION: Parent process is waiting for the child process to complete.

PSEUDOCODE:

```
1. Start Program

2. Declare a variable `pid` of type `pid_t` to hold process IDs.

3. Fork a new process.

4. Call `fork()` and store the returned value in `pid`.

5. If `pid` is 0:
   a. Print "It is the child process and pid is [child's process ID]" using `getpid()`.
   b. Loop from 0 to 7:
      i. Print each value of the loop variable `i`.
   c. Exit the child process with status 0 using `exit(0)`.

6. Else If `pid` is greater than 0 (indicating the current process is the parent):
   a. Print "It is the parent process and pid is [parent's process ID]" using `getpid()`.
   b. Declare an integer variable `status` to capture the exit status of the child.
   c. Wait for the child process to terminate using `wait(&status)`.
   d. Print "Child is reaped" to indicate that the child process has been cleaned up.

7. Else If `pid` is less than 0 (indicating an error occurred during `fork()`):
   a. Print "Error in forking.."
   b. Exit the program with a failure status using `exit(EXIT_FAILURE)`.

8. End Program
```

SOURCE CODE:

```
/* wait system call */
#include<stdio.h> // printf()
#include<stdlib.h> // exit()
#include<sys/types.h> // pid_t
#include<sys/wait.h> // wait()
#include<unistd.h> // fork
int main(int argc, char **argv)
{
        pid_t pid;
        fflush(stdout);
        pid = fork();
        if(pid==0)
        {
            printf("It is the child process and pid is %d\n",getpid());
            int i=0;
            for(i=0;i<8;i++)
            {
                printf("%d\n",i);
            }
            exit(0);
        }
        else if(pid > 0)
        {
            printf("It is the parent process and pid is %d\n",getpid());
            int status;
            wait(&status);
            printf("Child is reaped\n");
        }
        else
        {
            printf("Error in forking..\n");
            exit(EXIT_FAILURE);
        }
        getchar();
        return 0;
}
```

OUTPUT:

It is the parent process and pid is 19646
It is the child process and pid is 19650
0
1
2
3
4
5
6
7
Child is reaped

RESULT:

Thus, the program was executed and verified successfully.

PROGRAM1.3 Execv() system call

DESCRIPTION: Execute a linux command using execv() system call

PSEUDOCODE:

```
1. Start the program.

2. Declare the main function with two parameters:
   a. `argc` (argument count).
   b. `argv[]` (argument vector).

3. Print "before execv" to indicate execution before
   calling the `execv()` system call.

4. Call `execv()` with the following arguments:
   a. The first argument is the path `/bin/ls`.
   b. The second argument is `argv[]`, containing
      the command-line arguments.

5. If `execv()` is successful:
   a. The current process image is replaced by the
      new process (i.e., the `ls` command).
   b. The program will not return to the original code
      after this point.

6. If `execv()` fails:
   a. The program continues to the next line.
   b. Print "after execv" to indicate failure.

7. Call `getchar()` to wait for user input.
   (This will execute only if `execv()` fails.)

8. Return 0 to indicate successful termination.

9. End the program.
```

SOURCE CODE:

```
/* execv system call */
#include<stdio.h>
#include<sys/types.h>
int main(int argc,char *argv[])
{
        printf("before execv\n");
        execv("/bin/ls",argv);
        printf("after execv\n");

        getchar();
        return 0;
}
```

OUTPUT:

```
$ vi execv.c
$ cc execv.c
$ ./a.out
before execv
a1 aaa aaa.txt abc a.out b1 b2 comm.c db db1 demo2 dir1 direc.c execl.c execv.c f1.txt
fflag.c file1 file2 fork.c m1 m2 wait.c xyz
```

RESULT:

Thus, the program was executed and verified successfully.

PROGRAM 2

Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

Below are different times with respect to a process.

Arrival Time: Time at which the process arrives in the ready queue.

Completion Time: Time at which process completes its execution.

Burst Time: Time required by a process for CPU 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 turnaround time and burst time.

Waiting Time = Turn Around Time - Burst Time.

PROGRAM 2.1

Write a C program to implement FCFS CPU scheduling algorithm.

ALGORITHM

1. Initialize Variables:
   - n - Number of processes (integer).
   - pcb - Array of structures to hold each process's information (id, arrivalTime, burstTime, startTime, completionTime, turnAroundTime, waitTime).
   - prevProcCompTime - Tracks the completion time of the last scheduled process, initialized to 0.
   - totalWaitTime and totalTurnAroundTime - Variables to accumulate the wait and turn-around times for all processes, initialized to 0.
2. Input Process Data:
   - Prompt the user: “Enter the number of processes.”
   - Store the input in n.
   - For each process i from 0 to n-1:
     - Prompt the user: “Enter Process ID, Arrival Time, and Burst Time.”
     - Read and store id, arrivalTime, and burstTime in pcb[i].
3. Calculate Scheduling Details Using FCFS:
   - For each process i from 0 to n-1:
     - Set startTime for pcb[i] to the greater of pcb[i].arrivalTime or prevProcCompTime.
     - Calculate completionTime for pcb[i] as startTime + burstTime.
     - Calculate turnAroundTime for pcb[i] as completionTime - arrivalTime.
     - Calculate waitTime for pcb[i] as turnAroundTime - burstTime.
     - Update prevProcCompTime to completionTime.
     - Add turnAroundTime to totalTurnAroundTime.
     - Add waitTime to totalWaitTime.
4. Calculate Averages:
   - avgWaitTime = totalWaitTime / n
   - avgTurnAroundTime = totalTurnAroundTime / n
5. Display Results:
   - For each process i from 0 to n-1, display:

- id, arrivalTime, burstTime, startTime, completionTime, turnAroundTime, and waitTime for each process.

- Display avgWaitTime and avgTurnAroundTime.

SOURCE CODE:

/* A program to simulate the FCFS CPU scheduling algorithm */

```
/*Program : First Come First Served Scheduling
  Input      : Process details(Id, Arrival time(int) and Burst time(int))in the ascending order
               of the arrival time
  Output   : Each Process - Start time, Completion time, Turn Around Time, Wait Time
               Average Turn around time and average wait time*/

#include <stdio.h>
struct processControlBlock {
   int id;
   int arrivalTime, burstTime;
   int startTime, completionTime, waitTime, turnAroundTime;
} pcb[20] = {0};

int main()
{
   int i, n, prevProcCompTime = 0;
   int totalWaitTime = 0, totalTurnAroundTime = 0;
   float avgWaitTime, avgTurnAroundTime;

   /*Read the process data */
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   printf("Enter the process details:\nProcessId (integer), Arrival Time, Burst Time\n");
   for (i = 0; i < n; i++)
   {
      scanf("%d%d%d", &pcb[i].id, &pcb[i].arrivalTime, &pcb[i].burstTime);
```

```
    }

    /*Using FCFS algo: compute start, completition, turn around and wait time. */
    for (i = 0; i < n; i++)
    {
        pcb[i].startTime        = prevProcCompTime;
        pcb[i].completionTime = pcb[i].startTime + pcb[i].burstTime;
        pcb[i].turnAroundTime = pcb[i].completionTime - pcb[i].arrivalTime;
        pcb[i].waitTime         = pcb[i].turnAroundTime - pcb[i].burstTime;
        prevProcCompTime    = pcb[i].completionTime;

        totalWaitTime         += pcb[i].waitTime;
        totalTurnAroundTime  += pcb[i].turnAroundTime;
    }

    /*Diplay details of each process*/
    printf("\nPId \t Arrivaltime \t BurstTime \t StartTime \t CompletitionTime
            \t TurnAroundTime \t WaitTime");
    for (i = 0; i < n; i++)
    {
        printf("\n%d\t%6d\t\t%6d\t\t%6d\t\t%6d\t\t\t%6d \t\t\t%6d",
                pcb[i].id, pcb[i].arrivalTime, pcb[i].burstTime, pcb[i].startTime,
pcb[i].completionTime, pcb[i].turnAroundTime, pcb[i].waitTime);
    }
    avgWaitTime = (float)totalWaitTime / n;
    avgTurnAroundTime = (float)totalTurnAroundTime / n;
    printf("\nAverage Waiting time: %f", avgWaitTime);
    printf("\nAverage Turn Around Time: %f\n", avgTurnAroundTime);
    return 0;
}
```

OUTPUT1:

```
Enter the number of processes: 4
Enter the process details:
ProcessId (integer), Arrival Time, Burst Time
1 0 5
2 0 7
3 0 2
4 0 1

PId      Arrivaltime     BurstTime       StartTime       CompletitionTime        TurnAroundTime          WaitTime
1           0               5               0               5                       5                       0
2           0               7               5               12                      12                      5
3           0               2               12              14                      14                      12
4           0               1               14              15                      15                      14
Average Waiting time: 7.750000
Average Turn Around Time: 11.500000
```

OUTPUT2:

```
Enter the number of processes: 3
Enter the process details:
ProcessId (integer), Arrival Time, Burst Time
1 0 4
2 1 3
3 2 6

PId      Arrivaltime     BurstTime       StartTime       CompletitionTime        TurnAroundTime          WaitTime
1           0               4               0               4                       4                       0
2           1               3               4               7                       6                       3
3           2               6               7               13                      11                      5
Average Waiting time: 2.666667
Average Turn Around Time: 7.000000
```

PROGRAM 2.2

Write a C program to implement SJF CPU scheduling algorithm.

Objective: Schedule processes based on burst times, calculating each process's start time, completion time, turn-around time, and wait time. Also, calculate the average turn-around and wait times.

ALGORITHM: Shortest Job First (Non-Preemptive)

1. Initialize Variables:
    - n - Number of processes (integer).
    - pcb - Array of structures to hold each process's information (id, burstTime, startTime, completionTime, turnAroundTime, waitTime).
    - prevProcCompTime - Tracks the completion time of the last scheduled process, initialized to 0.
    - totalWaitTime and totalTurnAroundTime - Variables to accumulate the wait and turn-around times for all processes, initialized to 0.
2. Input Process Data:
    - Prompt the user: "Enter the number of processes."
    - Store the input in n.
    - For each process i from 0 to n-1:
        - Prompt the user: "Enter Process ID and Burst Time."
        - Read and store id and burstTime in pcb[i].
3. Sort Processes by Burst Time:
    - Use a simple sorting algorithm (e.g., bubble sort) to sort pcb based on burstTime in ascending order.
    - For each pair of processes i and j where j > i:
        - If pcb[i].burstTime > pcb[j].burstTime, swap pcb[i] and pcb[j].
4. Calculate Scheduling Details Using SJF:
    - For each process i from 0 to n-1:
        - Set startTime for pcb[i] to prevProcCompTime.
        - Calculate completionTime for pcb[i] as startTime + burstTime.
        - Calculate turnAroundTime for pcb[i] as completionTime.
        - Calculate waitTime for pcb[i] as turnAroundTime - burstTime.
        - Update prevProcCompTime to completionTime.

- Add turnAroundTime to totalTurnAroundTime.
- Add waitTime to totalWaitTime.

5. Calculate Averages:
   - avgWaitTime = totalWaitTime / n
   - avgTurnAroundTime = totalTurnAroundTime / n
6. Display Results:
   - For each process i from 0 to n-1, display:
     - id, burstTime, startTime, completionTime, turnAroundTime, and waitTime.
   - Display avgWaitTime and avgTurnAroundTime.

SOURCE CODE:

```
/*Program: SHORTEST JOB FIRST (SJF) Scheduling Algorithm
Input       : Process details(Id, Burst time(int))  assuming Arrival time for all is same and 0 ms.
Output      : Each Process - Start time, Completition time, Turn Around Time, Wait Time
          Average Turn around time and average wait time
*/
#include <stdio.h>
struct processControlBlock {
   int id;
   int burstTime;
   int startTime, completionTime, waitTime, turnAroundTime;
} pcb[20] = {0};
int main()
{
   int i, n, prevProcCompTime = 0;
   int totalWaitTime = 0, totalTurnAroundTime = 0;
   float avgWaitTime, avgTurnAroundTime;

   /*Read the process data */
```

```
printf("Enter the number of processes: ");
scanf("%d", &n);
printf("Enter the process details\nProcessId, Burst Time\n");
for (i = 0; i < n; i++)
{
   scanf("%d %d", &pcb[i].id, &pcb[i].burstTime);
}

/* Sorting pcb array in the ascending order of the burst time*/
for(int i=0;i<n;i++){
   for(int j=i+1;j<n;j++){
      if(pcb[i].burstTime > pcb[j].burstTime){
         int temp = pcb[i].burstTime;
         pcb[i].burstTime = pcb[j].burstTime;
         pcb[j].burstTime = temp;

         temp = pcb[i].id;
         pcb[i].id = pcb[j].id;
         pcb[j].id = temp;
      }
   }
}
/*  Using SJF algo: compute start, completition, turn around and wait time. */
for (i = 0; i < n; i++)
{
   pcb[i].startTime      = prevProcCompTime;
   pcb[i].completionTime = pcb[i].startTime + pcb[i].burstTime;
   pcb[i].turnAroundTime = pcb[i].completionTime;
   pcb[i].waitTime       = pcb[i].turnAroundTime - pcb[i].burstTime;
   prevProcCompTime      = pcb[i].completionTime;
```

```
        totalWaitTime        += pcb[i].waitTime;
        totalTurnAroundTime   += pcb[i].turnAroundTime;
    }
    /*Diplay details of each process*/
     printf("\nPId  \t BurstTime \t StartTime \t CompletitionTime \t TurnAroundTime \t WaitTime");
    for (i = 0; i < n; i++)
    {
        printf("\n%d\t%6d\t\t%6d\t\t%6d\t\t\t%6d \t\t\t%6d", pcb[i].id,pcb[i].burstTime,
          pcb[i].startTime, pcb[i].completionTime, pcb[i].turnAroundTime, pcb[i].waitTime);
    }
    avgWaitTime = (float)totalWaitTime / n;
    avgTurnAroundTime = (float)totalTurnAroundTime / n;
    printf("\nAverage Waiting time: %f", avgWaitTime);
    printf("\nAverage Turn Around Time: %f\n", avgTurnAroundTime);
    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 4
Enter the process details
ProcessId, Burst Time
1 5
2 7
3 2
4 1

PId      BurstTime       StartTime       CompletitionTime        TurnAroundTime          WaitTime
4           1               0               1                       1                       0
3           2               1               3                       3                       1
1           5               3               8                       8                       3
2           7               8               15                      15                      8
Average Waiting time: 3.000000
Average Turn Around Time: 6.750000
```

PROGRAM 2.3 Write a C program to implement Round Robin CPU scheduling algorithm.

ALGORITHM: Round Robin Scheduling

1. Input:
   - Read the number of processes, n.
   - Read the quantum time, QuantumTime.
   - For each process i (from 0 to n−1):
     - Read pcb[i].id and pcb[i].burstTime.
     - Initialize pcb[i].startTime to -1 to track when the process first starts.
2. Initialize Remaining Burst Time:
   - Create an array remainingBurstTime where remainingBurstTime[i] = pcb[i].burstTime.
3. Round Robin Execution:
   - Initialize GanttChart to 0 to represent the current time.
   - Set allProcessesCompleted to 0 to track when all processes are done.
   - While allProcessesCompleted is not 1:
     - Set allProcessesCompleted to 1 at the beginning of each cycle.
     - For each process i (from 0 to n−1):
       - If remainingBurstTime[i] > 0 (process still needs execution time):
         - Set allProcessesCompleted to 0.
         - If pcb[i].startTime is -1, set pcb[i].startTime = GanttChart to record the time when the process first starts.
         - Execute the process for either QuantumTime or its remaining burst time:
           - If remainingBurstTime[i] >= QuantumTime:
             - Decrease remainingBurstTime[i] by QuantumTime.
             - Increase GanttChart by QuantumTime.
           - Else:
             - Increase GanttChart by remainingBurstTime[i].
             - Set remainingBurstTime[i] to 0.
         - If remainingBurstTime[i] == 0, set pcb[i].completionTime = GanttChart.

4. Calculate Times:
   - For each process i (from 0 to n−1):
     - Calculate pcb[i].turnAroundTime = pcb[i].completionTime.
     - Calculate pcb[i].waitTime = pcb[i].turnAroundTime - pcb[i].burstTime.
     - Accumulate totalWaitTime += pcb[i].waitTime and totalTurnAroundTime += pcb[i].turnAroundTime.
5. Output:
   - Display each process's ID, burst time, completion time, turn-around time, and wait time.
   - Calculate avgWaitTime = totalWaitTime / n.
   - Calculate avgTurnAroundTime = totalTurnAroundTime / n.
   - Display avgWaitTime and avgTurnAroundTime.

SOUR CE CODE:

```
/*Program:  Round Robin Scheduling Algorithm
Input     :  Process details(Id, Burst time(int))  assuming Arrival time for all is
             same and 0 ms.
Output    :  Each Process - Start time, Completition time, Turn Around Time,
             Wait Time, Average Turn around time and average wait time
*/
#include <stdio.h>

struct processControlBlock {

   int id;

   int burstTime;

   int startTime, completionTime, waitTime, turAroundTime;

} pcb[20] = {0};

int main() {

   int i, n, GanttChart = 0;    // GanttChart keeps track of the current time

   int QuantumTime;

   int totalWaitTime = 0, totalTurnAroundTime = 0;

   float avgWaitTime, avgTurnAroundTime;


   // Input process data

   printf("Enter the number of processes: ");
```

```
    scanf("%d", &n);
    printf("Enter Quantum Time: ");
    scanf("%d", &QuantumTime);
    printf("Enter Process ID and Burst Time:\n");
    for (i = 0; i < n; i++) {
        scanf("%d %d", &pcb[i].id, &pcb[i].burstTime);
        pcb[i].startTime = -1;    // Initialize start time to -1 to track first start
    }
    // Temporary array to store remaining burst time for each process
    int remainingBurstTime[20];
    for (i = 0; i < n; i++) {
        remainingBurstTime[i] = pcb[i].burstTime;
    }
    int allProcessesCompleted = 0;  // Flag to check if all processes are completed
    while (!allProcessesCompleted) {
       allProcessesCompleted = 1;  // Assume all processes are complete at the beginning of each
cycle
        for (i = 0; i < n; i++) {
            if (remainingBurstTime[i] > 0) {  // Process still has burst time left
                allProcessesCompleted = 0;    // At least one process needs more time

                // Record the start time for the process the first time it starts
                if (pcb[i].startTime == -1) {
                    pcb[i].startTime = GanttChart;
                }

                 // Process executes for either Quantum Time or remaining burst time, whichever is
smaller
                if (remainingBurstTime[i] >= QuantumTime) {
                    GanttChart += QuantumTime;
                    remainingBurstTime[i] -= QuantumTime;
```

```
            } else {
                GanttChart += remainingBurstTime[i];
                remainingBurstTime[i] = 0;
            }

            // If process is completed, set its completion time
            if (remainingBurstTime[i] == 0) {
                pcb[i].completionTime = GanttChart;
            }
        }
    }
}
// Calculate turn-around and wait times for each process
for (i = 0; i < n; i++) {
    pcb[i].turnAroundTime = pcb[i].completionTime;
    pcb[i].waitTime = pcb[i].turnAroundTime - pcb[i].burstTime;

    totalWaitTime += pcb[i].waitTime;
    totalTurnAroundTime += pcb[i].turnAroundTime;
}

// Display process information
printf("\nProcess ID\tBurstTime\tCompletionTime\tTurnAroundTime\tWaitTime");
for (i = 0; i < n; i++) {
    printf("\n%d\t\t%6d\t\t%6d\t\t%6d\t\t%6d",
            pcb[i].id, pcb[i].burstTime, pcb[i].completionTime, pcb[i].turnAroundTime,
pcb[i].waitTime);
}
// Calculate and display average waiting time and turn-around time
avgWaitTime = (float)totalWaitTime / n;
avgTurnAroundTime = (float)totalTurnAroundTime / n;
```

```
    printf("\n\nAverage Waiting Time: %.2f", avgWaitTime);
    printf("\nAverage Turn Around Time: %.2f\n", avgTurnAroundTime);
    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 4
Enter Quantum Time: 2
Enter Process ID and Burst Time:
1 7
2 2
3 1
4 4

Process ID      BurstTime       CompletionTime  TurnAroundTime  WaitTime
1                   7               14              14              7
2                   2                4              4               2
3                   1                5              5               4
4                   4               11              11              7

Average Waiting Time: 5.00
Average Turn Around Time: 8.50
```

PROGRAM 2.4 Write a C program to implement Priority based scheduling

ALGORITHM

1. **Input**:
   - Read the number of processes, n.
   - For each process iii (from 0 to n−1n - 1n−1):
     - Read pcb[i].id, pcb[i].Priority, and pcb[i].burstTime.
2. **Processes by Priority**:
   - For each process iii from 0 to n−1n - 1n−1:
     - For each process jjj from i+1i + 1i+1 to n−1n - 1n−1:
       - If pcb[i].Priority > pcb[j].Priority, swap the values of pcb[i] and pcb[j] to sort processes in ascending order of priority.
3. **Calculate Times**:
   - Initialize prevProcCompTime to 0 to track the completion time of the previous process.
   - For each process iii from 0 to n−1n - 1n−1:
     - Set pcb[i].startTime = prevProcCompTime.
     - Calculate pcb[i].completionTime = pcb[i].startTime + pcb[i].burstTime.
     - Calculate pcb[i].turnAroundTime = pcb[i].completionTime.
     - Calculate pcb[i].waitTime = pcb[i].turnAroundTime - pcb[i].burstTime.
     - Update prevProcCompTime = pcb[i].completionTime.
     - Accumulate totalWaitTime += pcb[i].waitTime and totalTurnAroundTime += pcb[i].turnAroundTime.
4. **Output**:
   - Display each process's ID, priority, burst time, start time, completion time, turn-around time, and wait time.
   - Calculate avgWaitTime = totalWaitTime / n.
   - Calculate avgTurnAroundTime = totalTurnAroundTime / n.
   - Display avgWaitTime and avgTurnAroundTime.

SOURCE CODE:

```
#include <stdio.h>
struct processControlBlock {
   int id,Priority;
   int burstTime;
   int startTime, completionTime, waitTime, turnAroundTime;
} pcb[20] = {0};
int main()
{
   int i, n, prevProcCompTime = 0;
   int totalWaitTime = 0, totalTurnAroundTime = 0;
   float avgWaitTime, avgTurnAroundTime;
   /*Read the process data */
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   printf("Enter the process details\nProcessId, Priority, Burst Time:\n");
   for (i = 0; i < n; i++)
   {
      scanf("%d %d %d", &pcb[i].id,&pcb[i].Priority, &pcb[i].burstTime);
   }

   /* Sorting process id, Priority , Burst Time*/
   for(int i=0;i<n;i++){
      for(int j=i+1;j<n;j++){
         if(pcb[i].Priority > pcb[j].Priority){

            int temp = pcb[i].Priority;
            pcb[i].Priority = pcb[j].Priority;
            pcb[j].Priority = temp;
```

```
            temp = pcb[i].burstTime;
            pcb[i].burstTime = pcb[j].burstTime;
            pcb[j].burstTime = temp;

            temp = pcb[i].id;
            pcb[i].id = pcb[j].id;
            pcb[j].id = temp;
          }
       }
    }
 /*  Using Priority algo, compute start, completition, turn around and wait time. */
    for (i = 0; i < n; i++)
    {
       pcb[i].startTime      = prevProcCompTime;
       pcb[i].completionTime = pcb[i].startTime + pcb[i].burstTime;
       pcb[i].turnAroundTime = pcb[i].completionTime;
       pcb[i].waitTime       = pcb[i].turnAroundTime - pcb[i].burstTime;
       prevProcCompTime      = pcb[i].completionTime;

       totalWaitTime         += pcb[i].waitTime;
       totalTurnAroundTime   += pcb[i].turnAroundTime;
    }
    /*Diplay details of each process*/
    printf("\nPId  \t Priority \t BurstTime \t StartTime \t CompletitionTime \t TurnAroundTime
\t WaitTime");
    for (i = 0; i < n; i++)
    {
       printf("\n%d\t%d\t%6d\t\t%6d\t\t%6d\t\t\t%6d\t\t\t%6d",pcb[i].id,pcb[i].Priority,
        pcb[i].burstTime, pcb[i].startTime, pcb[i].completionTime,
        pcb[i].turnAroundTime,   pcb[i].waitTime);
    }
```

```
    avgWaitTime = (float)totalWaitTime / n;
    avgTurnAroundTime = (float)totalTurnAroundTime / n;
    printf("\nAverage Waiting time: %f", avgWaitTime);
    printf("\nAverage Turn Around Time: %f\n", avgTurnAroundTime);
    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 4
Enter the process details
ProcessId, Priority, Burst Time:
1 2 7
2 3 5
3 1 2
4 4 1

PId      Priority        BurstTime       StartTime       CompletitionTime        TurnAroundTime          WaitTime
3       1               2               0               2                       2                       0
1       2               7               2               9                       9                       2
2       3               5               9               14                      14                      9
4       4               1               14              15                      15                      14
Average Waiting time: 6.250000
Average Turn Around Time: 10.000000
```