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.

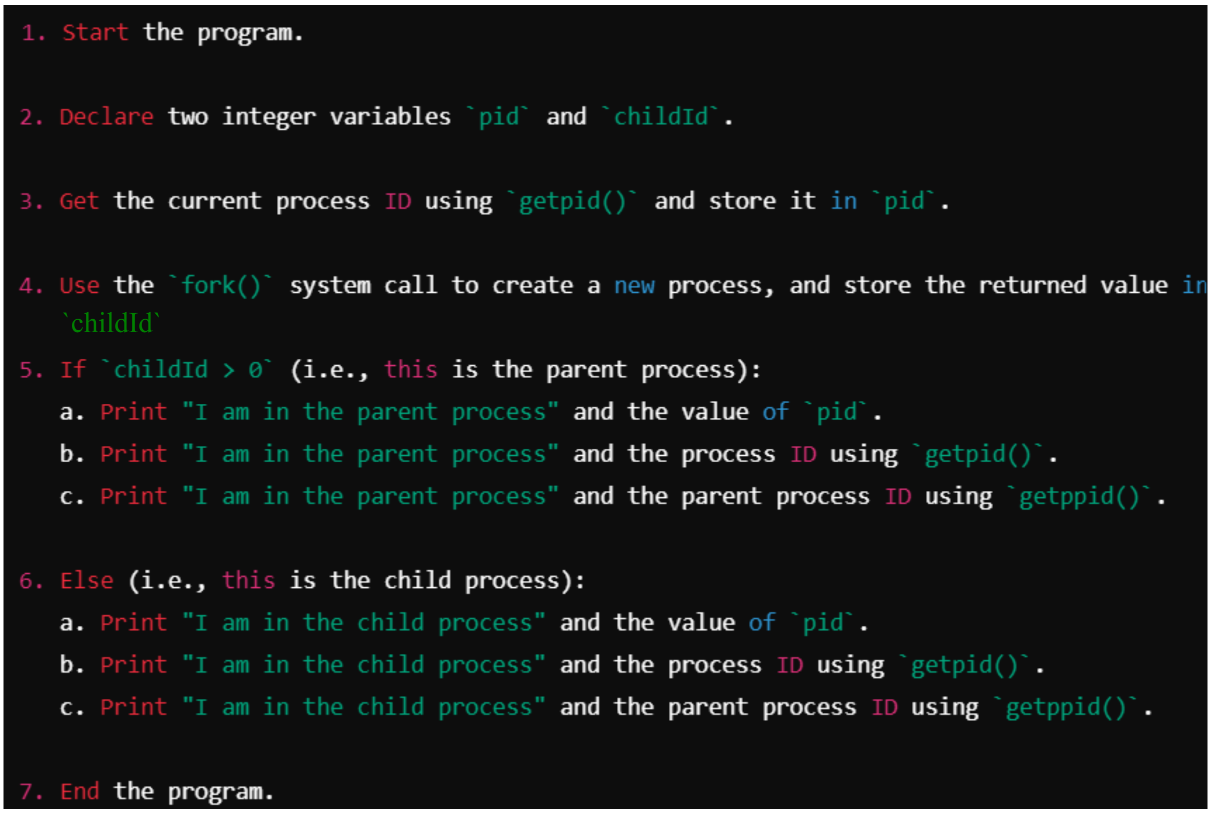
Note: These system calls declarations are present in **unistd.h** library.

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
```

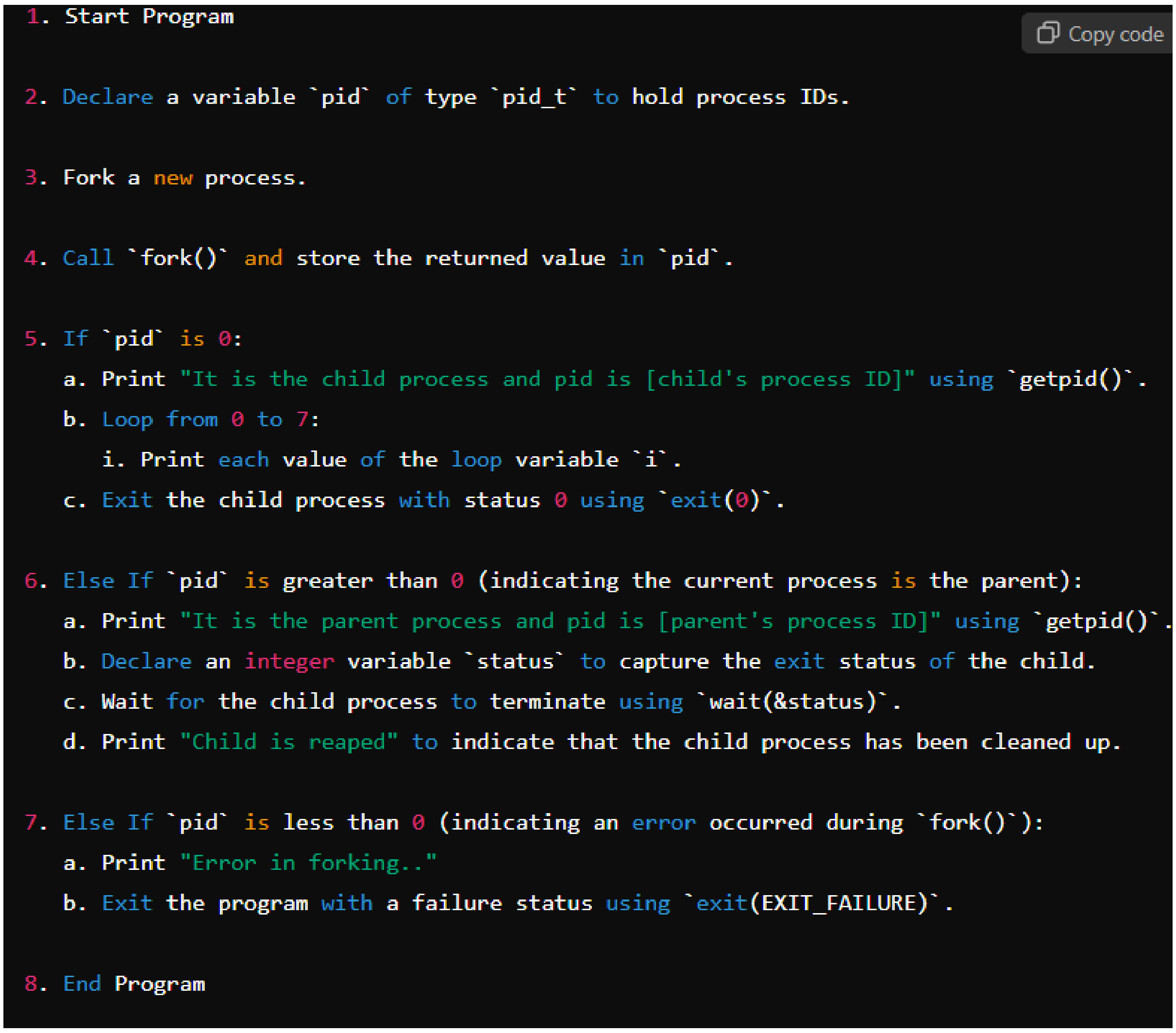
RESULT:

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
```

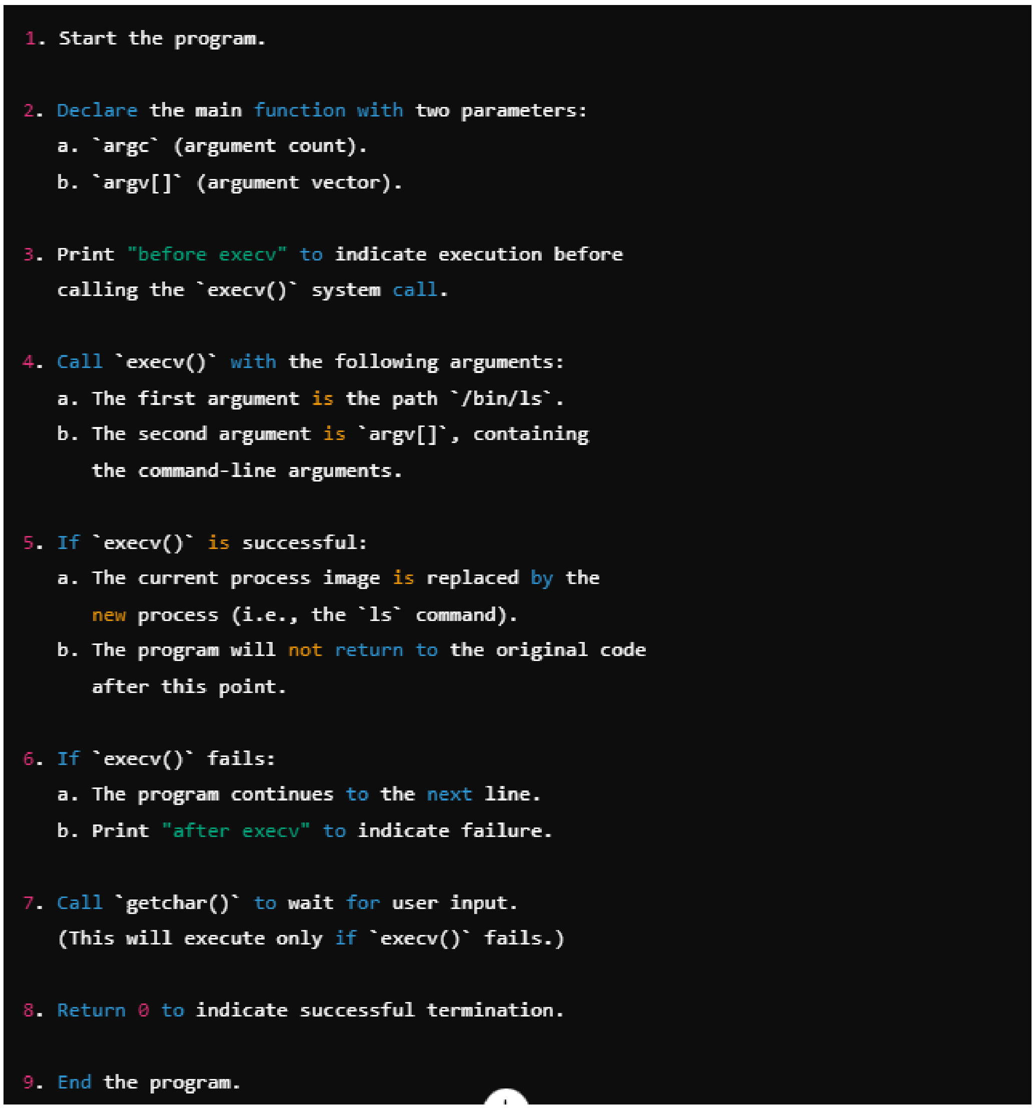
RESULT:

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. **Input:**
    - Read n (number of processes).
    - For each process i from 0 to n-1:
        - Input process name pn[i], arrival time arr[i], and burst time bur[i].
2. **Calculate times:**
    - For each process i:
        - If i == 0 (first process):
            - Set start time star[i] = arr[i].
            - Set waiting time wt[i] = 0.
            - Set finish time finish[i] = star[i] + bur[i].
            - Set turnaround time tat[i] = finish[i] - arr[i].
        - For each process i > 0:
            - Set start time star[i] = finish[i-1].
            - Set waiting time wt[i] = star[i] - arr[i].
            - Set finish time finish[i] = star[i] + bur[i].
            - Set turnaround time tat[i] = finish[i] - arr[i].
3. **Output process details:**
    - For each process i, print:
        - Process name pn[i],
        - Arrival time arr[i],
        - Burst time bur[i],
        - Start time star[i],
        - Turnaround time tat[i],
        - Finish time finish[i].
4. **Calculate and print averages:**
    - Compute total waiting time totwt by summing up wt[i].
    - Compute total turnaround time tottat by summing up tat[i].

- Calculate average waiting time: totwt / n.
- Calculate average turnaround time: tottat / n.
- Print the average waiting time and turnaround time.

INPUT:

- n: Total number of processes.
- pn[i]: Process name.
- arr[i]: Arrival time for each process.
- bur[i]: Burst time (execution time) for each process.

OUTPUT:

- Start time, finish time, waiting time, turnaround time for each process.
- Total and average waiting time and turnaround time.

VARIABLES:

- arr[i]: Arrival time of process i.
- bur[i]: Burst time (execution time) of process i.
- star[i]: Start time of process i.
- finish[i]: Finish time of process i.
- tat[i]: Turnaround time for process i (calculated as finish[i] - arr[i]).
- wt[i]: Waiting time for process i (calculated as star[i] - arr[i]).
- totwt: Total waiting time of all processes.
- tottat: Total turnaround time of all processes.

SOURCE CODE:

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

```
#include<stdio.h>
int main()
{
        char pn[10][10];
        int arr[10],bur[10],star[10],finish[10],tat[10],wt[10],i,n;
        int totwt=0,tottat=0;
        printf("Enter the number of processes:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter the Process Name, Arrival Time & Burst Time:");
                scanf("%s%d%d",&pn[i],&arr[i],&bur[i]);
        }
        for(i=0;i<n;i++)
        {
                if(i==0)
                {
                        star[i]=arr[i];
                        wt[i]=star[i]-arr[i];
                        finish[i]=star[i]+bur[i];
                        tat[i]=finish[i]-arr[i];
                }
                else
                {
                        star[i]=finish[i-1];
                        wt[i]=star[i]-arr[i];
                        finish[i]=star[i]+bur[i];
                        tat[i]=finish[i]-arr[i];
```

```
            }
        }
        printf("\nPName Arrtime Burtime Start TAT Finish");
        for(i=0;i<n;i++)
        {
            printf("\n%s\t%6d\t\t%6d\t%6d\t%6d\t%6d",pn[i],arr[i],bur[i],star[i],tat[i],fini
        sh[i]);
            totwt+=wt[i];
            tottat+=tat[i]
        }
        printf("\nAverage Waiting time:%f", (float)totwt/n);
        printf("\nAverage Turn Around Time:%f", (float)tottat/n);
}
```

OUTPUT:

```
$ cc fcfs.c
$ ./a.out
Enter the number of processes: 3
Enter the Process Name, Arrival Time & Burst Time: 1 2 3
Enter the Process Name, Arrival Time & Burst Time: 2 5 6
Enter the Process Name, Arrival Time & Burst Time: 3 6 7
PName   Arrtime   Burtime   Start   TAT   Finish
  1        2         3        2      3      5
  2        5         6        5      6      11
  3        6         7        11     12     18
Average Waiting time: 1.666667
Average Turn Around Time: 7.000000
```

PROGRAM 2.2

Write a C program to implement SJF CPU scheduling algorithm.

ALGORITHM: Shortest Job First (Non-Preemptive)

1. **Input:**
   - Read n (number of processes).
   - For each process i (from 0 to n-1):
     - Input burst time bt[i].
2. **Sort burst times** (Shortest Job First):
   - For i from 0 to n-2:
     - For j from i+1 to n-1:
       - If bt[i] > bt[j], swap bt[i] with bt[j] and also swap pno[i] with pno[j].
3. **Calculate waiting times:**
   - Set wt[0] = 0.
   - For each process i from 1 to n-1:
     - wt[i] = wt[i-1] + bt[i-1].
   - Sum up all wt[i] to get the total waiting time.
4. **Calculate turnaround times:**
   - For each process i from 0 to n-1:
     - tt[i] = bt[i] + wt[i].
   - Sum up all tt[i] to get the total turnaround time.
5. **Output process details**:
   - For each process i, print:
     - Process number pno[i]
     - Burst time bt[i]
     - Waiting time wt[i]
     - Turnaround time tt[i].
6. **Calculate and print average times**:
   - Average Waiting Time = (Total waiting time) / n.
   - Average Turnaround Time = (Total turnaround time) / n.

- Print the average waiting time and turnaround time.

Summary of Steps:

- **Step 1:** Input the number of processes and their burst times.
- **Step 2:** Sort the processes based on burst times (Shortest Job First).
- **Step 3:** Calculate waiting times for each process.
- **Step 4:** Calculate turnaround times for each process.
- **Step 5:** Print process information and compute average waiting and turnaround times.

INPUT:

- n: Total number of processes.
- bt[i]: Burst time (execution time) for each process.
- pno[i]: Process numbers.

OUTPUT:

- Process numbers, burst time, waiting time, and turnaround time for each process.
- Average waiting time and turnaround time.

VARIABLES:

- bt[i]: Burst time of process i.
- pno[i]: Process number.
- wt[i]: Waiting time for process i.
- tt[i]: Turnaround time for process i.
- temp: Temporary variable used for sorting.
- sum: Total waiting time.
- at: Total turnaround time.

SOURCE CODE:

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

```
#include<stdio.h>
#include<string.h>
void main()
{
        int i=0,pno[10],bt[10],n,wt[10],temp=0,j,tt[10];
        float sum=0,at=0;
        printf("\n Enter the no of process ");
        scanf("\n %d",&n);
        printf("\n Enter the burst time of each process");
        for(i=0;i<n;i++) {
                printf("\n p%d",i);
                scanf("%d",&bt[i]);
        }
        for(i=0;i<n-1;i++)
        {
                for(j=i+1;j<n;j++)
                {
                        if(bt[i]>bt[j])
                        {
                                temp=bt[i];
                                bt[i]=bt[j];
                                bt[j]=temp;
                                temp=pno[i];
                                pno[i]=pno[j];
                                pno[j]=temp;
                        }
                }
        }
        wt[0]=0;
        for(i=1;i<n;i++)
        {
                wt[i]=bt[i-1]+wt[i-1];
                sum=sum+wt[i];
         }
        printf("\n process no \t burst time\t waiting time \t turn around time\n");
        for(i=0;i<n;i++)
        {
                tt[i]=bt[i]+wt[i];
                at+=tt[i];
                printf("\n p%d\t\t%d\t\t%d\t\t%d",i,bt[i],wt[i],tt[i]);
        }
        printf("\n\n\t Average waiting time%f\n\t Average turn around time%f", sum/n, at/n);
}
```

OUTPUT:

```
$ cc sjf.c

$ ./a.out

Enter the no of process 5
Enter the burst time of each process
p0 1
p1 5
p2 2
p3 3
p4 4
process no        burst time        waiting time        turnaround time
    p0                1                 0                   1
    p1                2                 1                   3
    p2                3                 3                   6
    p3                4                 6                   10
    p4                5                 10                  15

Average waiting time 4.000000
Average turnaround time 7.000000
```

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 quantum for the time slice.
   - For each process, input its burst time (the time it needs to complete).
2. **Initialization:**
   - Each process has a burst time, waiting time (wait), completion time (comp), and a flag f to indicate if the process is still running.
   - Set the total waiting time (totalwait) and total turnaround time (totalturn) to 0.
   - Set the current system time (time) to 0 and a flag (flag) to 1, to control the while loop.
3. **Execution Loop (Round Robin Logic):**
   - Repeat until all processes have finished execution:
     - For each process:
       - If the process is still running (f == 1), check if its remaining burst time (p[i].burst - p[i].comp) is greater than the quantum.
         - If it is, increment the comp (completion) of that process by the quantum.
         - If not, set the waiting time for the process (wait), mark the process as finished (f = 0), and update the remaining time to 0.
       - Print the process ID, starting time, ending time, and remaining time for each process.
4. **Calculate Turnaround Time and Waiting Time:**
   - For each process, calculate:
     - Waiting time (wait = time - comp).
     - Turnaround time (turnaround = burst + wait).
   - Print these values for each process.
5. **Calculate Averages:**
   - Calculate and print the average waiting time and turnaround time by dividing the total waiting time and turnaround time by the number of processes.

INPUT:

- n: Total number of processes.
- quantum: Time slice (quantum) allocated for each process in milliseconds.
- p[i].burst: Array of burst times for each process.

OUTPUT:

- Execution order with starting time, ending time, and remaining burst time for each process after each quantum.
- Waiting time, turnaround time, and average waiting and turnaround times for all processes.

VARIABLES:

- n: Number of processes.
- quantum: The quantum time (in milliseconds) allocated to each process.
- p[i].burst: The burst time (execution time) of process i.
- p[i].wait: The waiting time for each process.
- p[i].comp: Completion time for each process.
- p[i].f: Flag indicating if the process is still unfinished (1) or finished (0).
- totalwait: Total accumulated waiting time for all processes.
- totalturn: Total accumulated turnaround time for all processes.
- time: Current system time during the execution of processes.
- flag: A flag that checks if any processes remain unfinished during the next iteration.
- j: Time slice used for the current execution (either full quantum or remaining burst time).

SOURCE CODE:

```
#include <stdio.h>
struct process
{
    int burst, wait, comp, f;
} p[20] = {0, 0};
int main()
{
    int n, i, j, totalwait = 0, totalturn = 0, quantum, flag = 1, time = 0;
    printf("\nEnter The No Of Process :");
    scanf("%d", &n);
    printf("\nEnter The Quantum time (in ms) :");
    scanf("%d", &quantum);
    for (i = 0; i < n; i++)
    {
        printf("Enter The Burst Time (in ms) For Process #%2d :", i + 1);
        scanf("%d", &p[i].burst);
        p[i].f = 1;
    }
    printf("\nOrder Of Execution \n");
    printf("\nProcess Starting Ending Remaining");
    printf("\n\t\tTime \tTime \t Time");
    while (flag == 1)
    {
        flag = 0;
        for (i = 0; i < n; i++)
        {
            if (p[i].f == 1)
            {
                flag = 1;
                j = quantum;
                if ((p[i].burst - p[i].comp) > quantum)
                {
                    p[i].comp += quantum;
                }
                else
                {
                    p[i].wait = time - p[i].comp;
                    j = p[i].burst - p[i].comp;
                    p[i].comp = p[i].burst;
                    p[i].f = 0;
                }
                  printf("\nprocess # %-3d %-10d %-10d %-10d", i + 1, time, time + j, p[i].burst -
p[i].comp);
                time += j;
            }
        }
    }
    printf("\n\n-------------------");
    printf("\nProcess \t Waiting Time TurnAround Time ");
    for (i = 0; i < n; i++)
    {
        printf("\nProces%-12d%-15d%-15d", i + 1, p[i].wait, p[i].wait + p[i].burst);
        totalwait = totalwait + p[i].wait;
        totalturn = totalturn + p[i].wait + p[i].burst;
    }
    printf("\n\nAverage\n------------------ ");
    printf("\nWaiting Time: %fms", (float)totalwait/n);
    printf("\nTurnAround Time : %fms\n\n", (float)totalturn/n);
    return 0;
}
```

OUTPUT:

```
Enter The No Of Process :3

Enter The Quantum time (in ms) :5
Enter The Burst Time (in ms) For Process # 1 :25
Enter The Burst Time (in ms) For Process # 2 :30
Enter The Burst Time (in ms) For Process # 3 :54

Order Of Execution

Process      Starting Time    Ending Time    Remaining Time

process # 1       0               5               20
process # 2       5               10              25
process # 3       10              15              49
process # 1       15              20              15
process # 2       20              25              20
process # 3       25              30              44
process # 1       30              35              10
process # 2       35              40              15
process # 3       40              45              39
process # 1       45              50              5
process # 2       50              55              10
process # 3       55              60              34
process # 1       60              65              0
process # 2       65              70              5
process # 3       70              75              29
process # 2       75              80              0
process # 3       80              85              24
process # 3       85              90              19
process # 3         90                    95              14
process # 3         95                    100             9
process # 3        100                    105             4
process # 3        105                    109             0

------------------
Process          Waiting Time TurnAround Time
Proces1                 40                 65
Proces2                 50                 80
Proces3                 55                109

Average
------------------
Waiting Time: 48.333332ms
TurnAround Time : 84.666664ms
```

PROGRAM 2.4 Write a C program to implement Priority based scheduling

ALGORITHM

1. **Step 1**: Input the total number of processes (n).
   - Prompt the user to input the number of processes to be scheduled.
2. **Step 2**: Input the burst time (pt[]) and priority (pp[]) for each process.
   - Initialize an array p[] to store the process number for reference.
   - For each process, input the burst time and priority, and store them in arrays pt[] and pp[] respectively.
   - Store the process number in array p[].
3. **Step 3**: Sort the processes based on priority in descending order (higher priority first).
   - Use nested loops to compare the priority of each process.
   - If a process with lower priority comes before a process with higher priority, swap them.
   - Ensure the burst time (pt[]), priority (pp[]), and process number (p[]) are all swapped together to maintain correct data.
4. **Step 4**: Initialize waiting time and turnaround time for the first process.
   - The waiting time for the first process is always 0 since no other processes were scheduled before it.
   - The turnaround time for the first process is equal to its burst time.
5. **Step 5**: Calculate the waiting time and turnaround time for each process.
   - For each subsequent process i:
     - The waiting time (w[i]) is equal to the turnaround time of the previous process (t[i-1]).
     - The turnaround time (t[i]) is the waiting time plus the burst time (w[i] + pt[i]).
6. **Step 6**: Accumulate the waiting times and turnaround times for calculating the averages.
   - Keep track of the total waiting time (awt) and total turnaround time (atat) as you calculate them for each process.
7. **Step 7**: Compute the average waiting time and average turnaround time.
   - Divide the accumulated waiting time and turnaround time by the total number of processes (n) to get the averages.
8. **Step 8**: Display the results.

  - Output the process number, burst time, waiting time, turnaround time, and priority for each process in a table format.
  - Print the average waiting time and average turnaround time.

INPUT:

- n: Number of processes.
- pt[]: Array of burst times for each process.
- pp[]: Array of priorities for each process.
- p[]: Array of process numbers (to track the original process sequence).

OUTPUT:

- Waiting time (w[]) and turnaround time (t[]) for each process.
- Average waiting time (awt) and average turnaround time (atat).

VARIABLES:

- n: Total number of processes.
- p[]: Array to store process numbers for reference.
- pp[]: Array to store the priority of each process.
- pt[]: Array to store the burst time (execution time) of each process.
- w[]: Array to store the waiting time for each process.
- t[]: Array to store the turnaround time for each process.
- awt: Average waiting time for all processes.
- atat: Average turnaround time for all processes.
- x: Temporary variable used for swapping during the sorting process.
- i, j: Loop variables for iteration over the processes.

SOURCE CODE:

```
#include <stdio.h>
void main()
{
    int x, n, p[10], pp[10], pt[10], w[10], t[10], awt, atat, i;
    printf("Enter the number of process : ");
    scanf("%d", &n);
    printf("\n Enter process : time priorities \n");
    for (i = 0; i < n; i++)
    {
        printf("\nProcess no %d : ", i + 1);
        scanf("%d %d", &pt[i], &pp[i]);
        p[i] = i + 1;
    }
    for (i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (pp[i] < pp[j])
            {
                x = pp[i];
                pp[i] = pp[j];
                pp[j] = x;
                x = pt[i];
                pt[i] = pt[j];
                pt[j] = x;
                x = p[i];
                p[i] = p[j];
                p[j] = x;
            }
        }
    }
    w[0] = 0;
    awt = 0;
    t[0] = pt[0];
    atat = t[0];
    for (i = 1; i < n; i++)
    {
        w[i] = t[i - 1];
        awt += w[i];
        t[i] = w[i] + pt[i];
        atat += t[i];
    }
    printf("\n\n Job \t Burst Time \t Wait Time \t Turn Around Time Priority \n");
    for (i = 0; i < n; i++)
        printf("\n %d \t\t %d \t\t %d \t\t %d \t\t %d \n", p[i], pt[i], w[i], t[i], pp[i]);
    awt /= n;
    atat /= n;
    printf("\n Average Wait Time : %d \n", awt);
    printf("\n Average Turn Around Time : %d \n", atat);
}
```

OUTPUT:

```
Enter the number of process : 4
 Enter process : time priorities
Process no 1 : 3 1
Process no 2 : 4 2
Process no 3 : 5 3
Process no 4 : 6 4

 Job     Burst Time     Wait Time     Turn Around Time Priority


 4           6              0              6              4


 3           5              6              11             3


 2           4              11             15             2


 1           3              15             18             1

 Average Wait Time : 8
 Average Turn Around Time : 12
```