# Experiment : 2

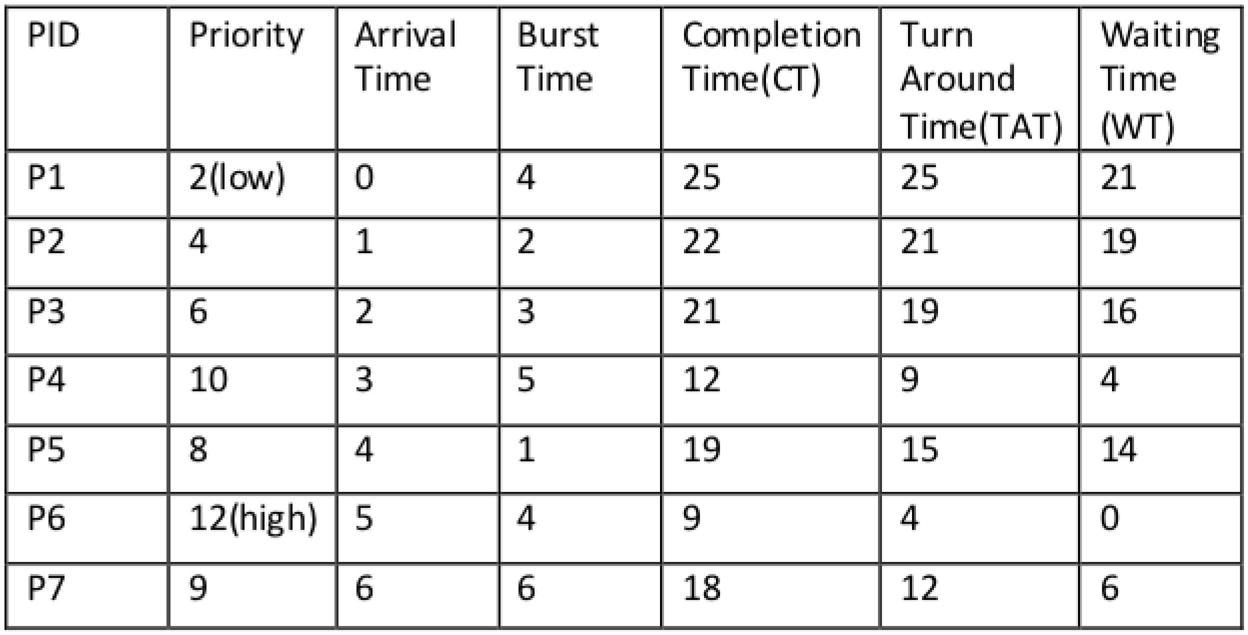
## AIM: Simulate the following CPU scheduling algorithms:(a) Priority (b) Round Robin

### (A).Priority

**Description:**

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

**Example:**



| PID | Priority | Arrival Time | Burst Time | Completion Time(CT) | Turn Around Time(TAT) | Waiting Time (WT) |
|---|---|---|---|---|---|---|
| P1 | 2(low) | 0 | 4 | 25 | 25 | 21 |
| P2 | 4 | 1 | 2 | 22 | 21 | 19 |
| P3 | 6 | 2 | 3 | 21 | 19 | 16 |
| P4 | 10 | 3 | 5 | 12 | 9 | 4 |
| P5 | 8 | 4 | 1 | 19 | 15 | 14 |
| P6 | 12(high) | 5 | 4 | 9 | 4 | 0 |
| P7 | 9 | 6 | 6 | 18 | 12 | 6 |

**Algorithm:**

**Step 1**: Start the process

**Step 2:** Accept the number of processes in the ready Queue.

**Step 3:** For each process in the ready Q, assign the process id and accept the CPU burst time .

**Step 4:** Sort the ready queue according to the priority number.

**Step 5:** Set the waiting of the first process as _0‘ and its burst time as its turnaround time.

**Step 6:** Arrange the processes based on process priority.

**Step 7:** For each process in the Ready Q calculate a) Waiting time(n)= waiting time (n-1) + Burst time (n-1) b) Turnaround time (n)= waiting time(n)+Burst time(n) .

**Step 8:** Calculate a) Average waiting time = Total waiting Time / Number of process b) Average Turnaround time = Total Turnaround Time / Number of process Print the results in an order.

**Step9:**Stop

## Experiment: 2 (A).Priority Program

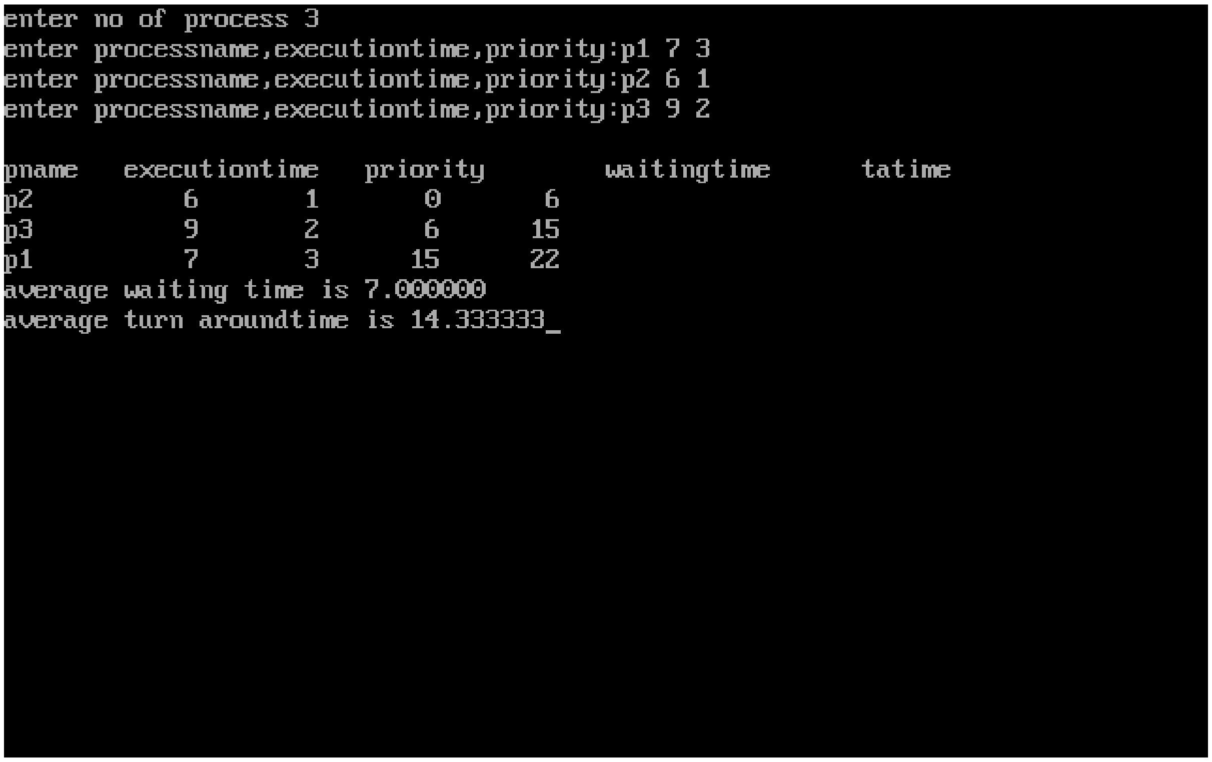
**Aim:** Simulate the following CPU scheduling algorithms:(a) Priority (b) Round Robin

**Program:**

```
#include <stdio.h>

#include <string.h>

#include <conio.h>

void main() {

    int et[10], temp, n, i, j, p[10], st[10], ft[10], wt[10], ta[10];

    int totwt = 0, tota = 0;

    float awt, ata;

    char pn[10][10], t[10];

    clrscr();

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

    scanf("%d", &n);

    // Input process name, execution time, and priority for each process

    for (i = 0; i < n; i++) {

        printf("Enter process name, execution time, priority: ");

        scanf("%s%d%d", &pn[i], &et[i], &p[i]);

    }

    // Sort the processes based on priority (ascending order)

    for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        if (p[i] < p[j]) {
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;
            temp = et[i];
            et[i] = et[j];
            et[j] = temp;
            strcpy(t, pn[i]);
            strcpy(pn[i], pn[j]);
            strcpy(pn[j], t);
        }
    }

// Calculate start time, waiting time, finish time, and turnaround time
for (i = 0; i < n; i++) {
    if (i == 0) {
        st[i] = wt[i] = 0;
        ft[i] = st[i] + et[i];
        ta[i] = ft[i];
    } else {
        st[i] = ft[i - 1];
        wt[i] = st[i];
```

```
            ft[i] = st[i] + et[i];

            ta[i] = ft[i];

        }

        totwt += wt[i];

        tota += ta[i];

    }


    awt = (float)totwt / n;

    ata = (float)tota / n;


    // Display process details and averages

    printf("\nProcess\tExecution Time\tPriority\tWaiting Time\tTurnaround Time");

    for (i = 0; i < n; i++)

        printf("\n%s\t%5d\t%5d\t%5d\t%5d", pn[i], et[i], p[i], wt[i], ta[i]);


    printf("\nAverage Waiting Time: %f", awt);

    printf("\nAverage Turnaround Time: %f", ata);


    getch();

}
```

Output:



```
enter no of process 3
enter processname,executiontime,priority:p1 7 3
enter processname,executiontime,priority:p2 6 1
enter processname,executiontime,priority:p3 9 2

pname   executiontime   priority        waitingtime      tatime
p2          6       1       0       6
p3          9       2       6      15
p1          7       3      15      22
average waiting time is 7.000000
average turn aroundtime is 14.333333_
```

**Suppose you have entered the following inputs:**

**Enter the number of processes: 3**

**Enter process name, execution time, priority:**

**P1 5 2**

**P2 3 1**

**P3 6 3**

**The output would be something like:**

| Process | Execution Time | Priority | Waiting Time | Turnaround Time |
|---|---|---|---|---|
| **P2** | **3** | **1** | **0** | **3** |
| **P1** | **5** | **2** | **3** | **8** |
| **P3** | **6** | **3** | **8** | **14** |

**Average Waiting Time: 3.666667**

**Average Turnaround Time: 8.333333**

This output shows the execution details for each process, including its name, execution time, priority, waiting time, and turnaround time. The averages for waiting time and turnaround time are also displayed at the end. Please note that the actual output may vary based on the input values you provide.

## (B). Round Robin

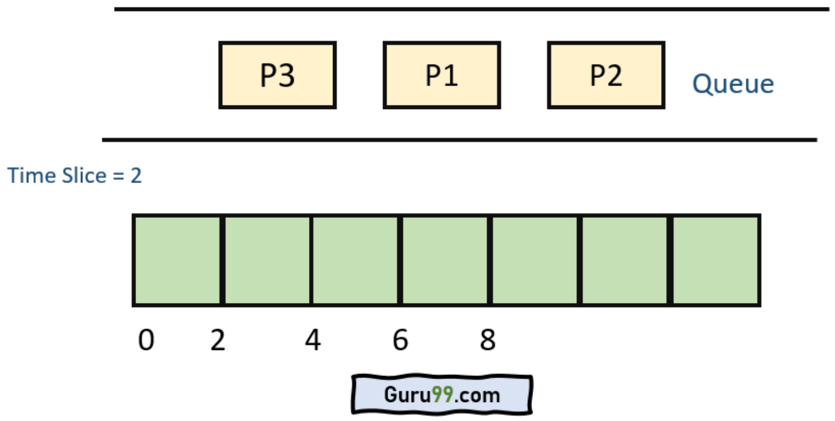
**AIM:** CPU scheduling algorithms: (b) Round Robin

**Description :**

To aim is written in C and implements a basic Round Robin CPU scheduling algorithm. This program takes the number of processes, time slice, and burst times of each process as input, and it calculates and displays the waiting time and turnaround time for each process, as well as the averages.

This program takes the number of processes, a time slice for the round-robin algorithm, and the burst times of each process. It then calculates the start time (**st**), end time (**et**), waiting time (**wt**), turnaround time (**tat**), and other related values for each process. Finally, it calculates and prints the average waiting time and average turnaround time for all processes.

**Example**



**ALGORITHM:**

**Step 1:** Start the process

**Step 2:** Accept the number of processes in the ready Queue and time quantum (or) time slice

**Step 3:** For each process in the ready Q, assign the process id and accept the CPU burst time

**Step 4:** Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice

**Step 5:** If the burst time is less than the time slice then the no. of time slices =1.

**Step 6:** Consider the ready queue is a circular Q, calculate a) Waiting time for process (n) = waiting time of process(n-1)+ burst timeof process(n-1 ) + the time difference in getting the CPU fromprocess(n-1) b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n)+ the time difference in getting CPU from process(n).

**Step 7:** Calculate a) Average waiting time = Total waiting Time / Number of process b) Average Turnaround time = Total Turnaround Time / Number ofprocess

**Step 8:** Stop the process

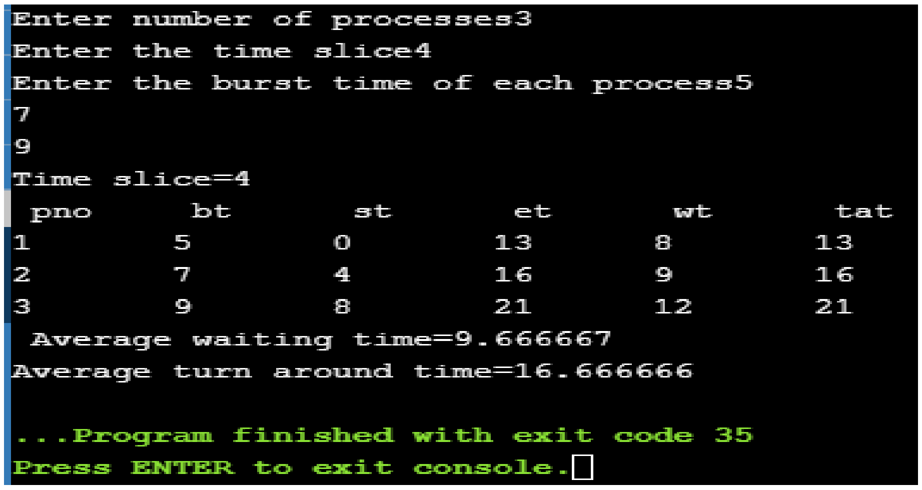
**(b) Round Robin:**

Below is an explanation of the program:

```
#include<stdio.h>

#include<conio.h>


void main() {

    int b[10], pno[10], ts, n, s[10], e[10], w[10], t[10], r[10];

    int i, c = 0, x = 0;

    float aw = 0, at = 0;


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

    scanf("%d", &n);


    // Assign process numbers

    for (i = 0; i < n; i++)

        pno[i] = i + 1;


    printf("Enter the time slice: ");

    scanf("%d", &ts);


    printf("Enter the burst time of each process:\n");

    for (i = 0; i < n; i++)
        scanf("%d", &b[i]);

    s[0] = 0;
    x = 0;
    c = 0;

    for (i = 0; i < n; i++) {
        if (b[i] < ts) {
            e[i] = x + b[i];
            r[i] = 0;
        } else {
            e[i] = ts + x;
            r[i] = b[i] - ts;
        }
        x = e[i];
        s[i + 1] = e[i];
        t[i] = e[i];
        w[i] = s[i];
    }

    while (c >= 0) {
        for (i = 0; i < n; i++) {
            if (r[i] != 0) {
                w[i] = w[i] + x - e[i];
                if (r[i] < ts) {
                    e[i] = x + r[i];
                    r[i] = 0;
                } else {
                    e[i] = x + ts;
                    r[i] = r[i] - ts;
                }
                x = e[i];
                t[i] = e[i];
            }
            if (r[i] != 0)
                c++;
        }
        c--;
    }

    for (i = 0; i < n; i++) {
        aw = aw + w[i];
        at = at + t[i];
    }

    aw = aw / n;
```

```
    at = at / n;


    printf("Time slice = %d", ts);

    printf("\npno \t bt \t st \t et \t wt \t tat\n");

    for (i = 0; i < n; i++)

        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pno[i], b[i], s[i], e[i], w[i], t[i]);


    printf("Average waiting time = %f\n", aw);

    printf("Average turnaround time = %f\n", at);

}
```

**Output:**



```
Enter number of processes3
Enter the time slice4
Enter the burst time of each process5
7
9
Time slice=4
 pno      bt       st       et       wt       tat
1        5        0        13       8        13
2        7        4        16       9        16
3        9        8        21       12       21
 Average waiting time=9.666667
Average turn around time=16.666666

...Program finished with exit code 35
Press ENTER to exit console.
```

OR

The program you provided is a C program for implementing the Round Robin scheduling algorithm. Below is an example of the program's output for a sample input:

Suppose you input the following values:

```
Enter the number of processes: 4

Enter the time slice: 3

Enter the burst time of each process:

5 9 12 7
```

**The program will produce an output similar to this:**

```
Time slice = 3

pno    bt    st    et    wt    tat

1      5     0     3     8     3

2      9     3     12    15    12

3      12    12    15    15    3

4      7     15    18    16    4

Average waiting time = 11.750000

Average turnaround time = 5.500000
```

Here's what each column in the output represents:

- **pno**: Process Number
- **bt**: Burst Time (input)
- **st**: Start Time
- **et**: End Time
- **wt**: Waiting Time
- **tat**: Turnaround Time

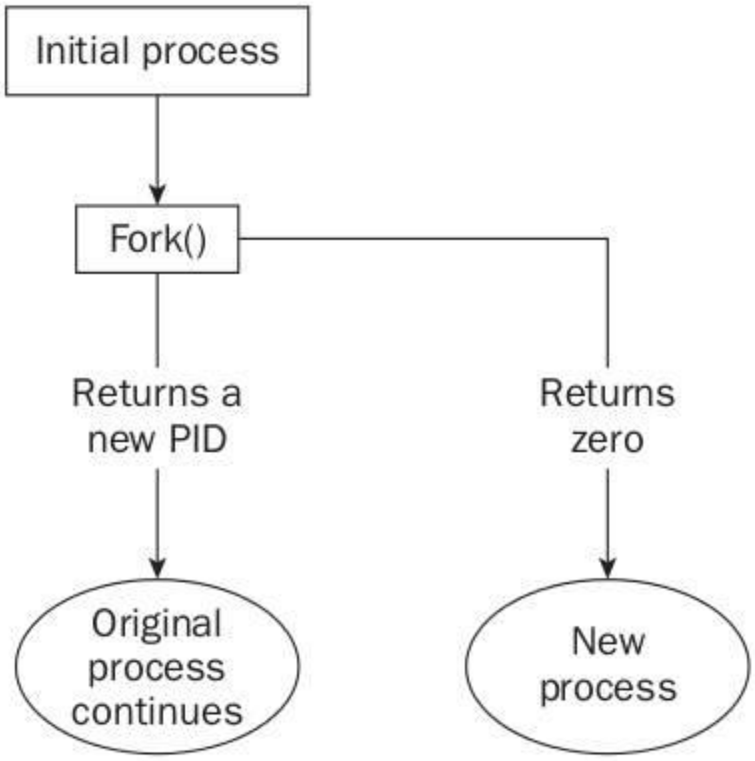
The output provides the scheduling information for each process, including the start and end times, waiting times, and turnaround times. It also calculates and displays the average waiting time and average turnaround time for all processes. The actual output may vary based on the input values you provide.

---

## Experiment: 3

## Aim: Implementation of fork (), wait (), exec() and exit (), System calls.

The program you provided is a simple C program that demonstrates the use of fork() and wait() system calls to create and manage child processes. This program creates a child process, and both the parent and child processes print their process IDs (PID) and parent process IDs (PPID). After that, they each execute a loop to print some messages. Here's the explanation of the program:



Here's what the program does:

1. It forks a new process using the **fork()** system call. This creates a child process that is a copy of the parent process.
2. In the child process (identified by **p == 0**), it prints its own PID and PPID and runs a loop to print numbers 1 to 5.
3. In the parent process (identified by **p != 0**), it prints its own PID and PPID, waits for the child process to terminate using **wait()**, and then prints the PID of the terminated child. After that, it runs a loop to print numbers 1 to 5.

When you run the program, you will see output from both the parent and child processes, demonstrating the creation of a child process and their respective behaviors.

```
#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

int main() {

    pid_t p;

    p = fork(); // Create a new process


    if (p == -1) {

        printf("Fork Error");

        exit(0);

    } else if (p == 0) {// Child process

        int i;
        printf("Child PID is %d and PPID is %d\n", getpid(), getppid());
        for (i = 1; i < 6; i++) {
            printf("Child i is %d\n", i);
        }
        _exit(0); // Terminate the child process
    } else { // Parent process
        int i;
        printf("Parent PID is %d and PPID is %d\n", getpid(), getppid());
        pid_t p1 = wait(0); // Wait for the child process to terminate
        printf("PID=%d child ended\n", p1);
        for (i = 1; i < 6; i++) {
            printf("Parent i is %d\n", i);
        }
        exit(0); // Terminate the parent process
    }
    return 0;
}
```

OUTPUT:

```
Parent PID is 12345 and PPID is 6789
Child PID is 12346 and PPID is 12345
Child i is 1
Child i is 2
Child i is 3
Child i is 4
Child i is 5
PID=12346 child ended
Parent i is 1
Parent i is 2
Parent i is 3
Parent i is 4
Parent i is 5
```

Please note that the actual process IDs (PIDs) and parent process IDs (PPIDs) will be different when you run the program. The child process runs concurrently with the parent process, and you can see that both processes print their respective messages in parallel. The parent process also waits for the child process to terminate using **wait()**.