LAB NO :1
IMPLEMENTATION OF FORK FUNCTION IN C/C++

OBJECTIVE:

- To understand and implement the fork() system call in C/C++ and observe how processes are created and managed in a Unix-like operating system.

THEORY:

The fork() function is a system call used in Unix-like operating systems to create a new process. The new process created by fork() is called the child process, and the process that called fork() is the parent process. Both processes execute the same code following the fork() call but with different process IDs (PIDs).

- **Parent Process:** The original process that initiates the fork() call.
- **Child Process:** The new process created by the fork() call. It is a copy of the parent process, except for the PID and some other process-specific details.

The fork() function returns:

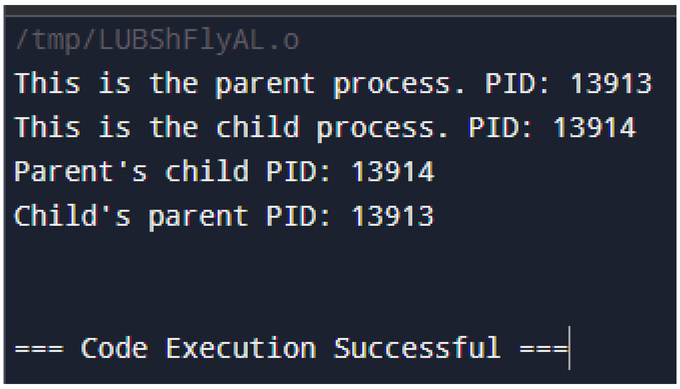
- 0 in the child process.
- The child's PID in the parent process.
- A negative value if the fork fails (indicating an error).

This behavior allows processes to perform different tasks based on whether they are the parent or the child.

DEMONSTRATION:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
   pid_t pid;
   pid = fork();

   if (pid < 0) {
      fprintf(stderr, "Fork failed\n");
      return 1;
   } else if (pid == 0) {
      printf("This is the child process. PID: %d\n", getpid());
      printf("Child's parent PID: %d\n", getppid());
   } else {
      printf("This is the parent process. PID: %d\n", getpid());
      printf("Parent's child PID: %d\n", pid);
   }

   return 0;
}
```

OUTPUT AND DISCUSSION:



```
/tmp/LUBShFlyAL.o
This is the parent process. PID: 13913
This is the child process. PID: 13914
Parent's child PID: 13914
Child's parent PID: 13913


=== Code Execution Successful ===
```

In this experiment, the fork() system call was successfully used to create a child process. The fork() function duplicates the parent process, creating a nearly identical child process. The child process has its unique PID but shares most of its environment with the parent. Both processes continue executing the same program following the fork(), but they can be distinguished by the return value of fork().

CONCLUSION:

The fork() system call is an essential function for process creation in Unix-like systems. This lab report demonstrated how to implement the fork() function in C, showing the relationship between parent and child processes.

LAB NO :2

IMPLEMENTATION OF FIRST-COME, FIRST-SERVED (FCFS) PROCESS SCHEDULING ALGORITHM IN C/C++

OBJECTIVE:

➢ To implement the First-Come, First-Served (FCFS) scheduling algorithm in C/C++ and analyze its performance in terms of waiting time, turnaround time, and overall process management.

THEORY:

First-Come, First-Served (FCFS) is one of the simplest CPU scheduling algorithms. In FCFS, the process that arrives first in the ready queue is allocated the CPU first. The processes are executed in the order they arrive without preemption, which means once a process starts execution, it runs to completion.

Key Terms:

- **Arrival Time (AT):** The time at which a process arrives in the ready queue.
- **Burst Time (BT):** The total time required by a process for execution on the CPU.
- **Waiting Time (WT):** The total time a process has been in the ready queue.
- **Turnaround Time (TAT):** The total time taken from submission to completion of the process.

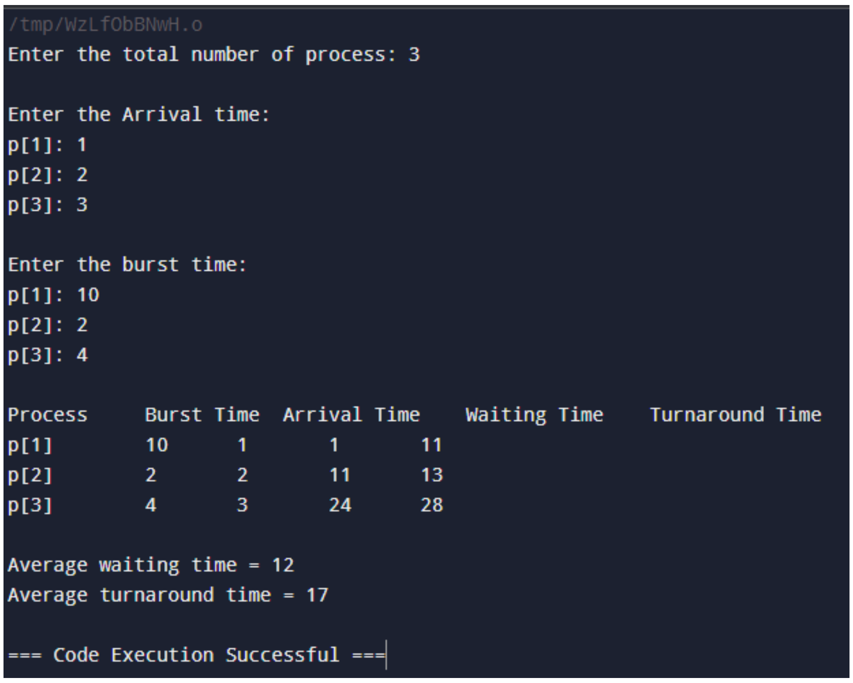
Formulas:

- **Waiting Time (WT) = Turnaround Time (TAT) - Burst Time (BT)**
- **Turnaround Time (TAT) = Completion Time (CT) - Arrival Time (AT)**

DEMONSTRATION:

```
#include <stdio.h>

int main(){
int n,bt[20],arvl_time[20],wt[20],tat[20],avwt = 0, avtat = 0, i,j;
printf("Enter the total number of process: ");
scanf("%d",&n);
printf("\nEnter the Arrival time: \n");
for(i=0;i<n;i++){
printf("p[%d]: ",i+1);
scanf("%d",&arvl_time[i]);
}
printf("\nEnter the burst time: \n");
for(i=0;i<n;i++){
printf("p[%d]: ",i+1);
scanf("%d",&bt[i]);
};
wt[0] = arvl_time[0];
for (i = 1; i < n; i++) {
wt[i] = 0;
for (j = 0; j < i; j++)
wt[i] += (bt[j] + wt[j]);
}
printf("\nProcess\t\tBurst Time\tArrival Time\tWaiting Time\tTurnaround Time");
for (i = 0; i < n; i++) {
tat[i] = bt[i] + wt[i];
avwt += wt[i];
avtat += tat[i];
printf("\np[%d]\t\t%d\t\t%d\t\t%d\t\t%d", i + 1, bt[i], arvl_time[i], wt[i], tat[i]);
}
printf("\n\nAverage waiting time = %d", avwt / n);
printf("\nAverage turnaround time = %d", avtat / n);
return 0;
}
```

OUTPUT AND DISCUSSION:



```
/tmp/WzLfObBNwH.o
Enter the total number of process: 3

Enter the Arrival time:
p[1]: 1
p[2]: 2
p[3]: 3

Enter the burst time:
p[1]: 10
p[2]: 2
p[3]: 4

Process     Burst Time  Arrival Time    Waiting Time    Turnaround Time
p[1]        10      1       1       11
p[2]        2       2       11      13
p[3]        4       3       24      28

Average waiting time = 12
Average turnaround time = 17

=== Code Execution Successful ===
```

CONCLUSION:

The FCFS scheduling algorithm is simple to implement but can result in suboptimal waiting times, especially when processes with shorter burst times arrive after a long process. While FCFS is easy to understand and implement, it is not always the most efficient scheduling algorithm in terms of process management and resource utilization.

LAB NO :3
IMPLEMENTATION OF SHORTEST JOB FIRST (SJF) PROCESS SCHEDULING ALGORITHM IN C

OBJECTIVE:

- To implement both non-preemptive and preemptive versions of the Shortest Job First (SJF) scheduling algorithm in C, and analyze their performance in terms of waiting time, turnaround time, and overall process efficiency.

THEORY:

Shortest Job First (SJF) is a CPU scheduling algorithm that selects the process with the smallest burst time for execution next. This algorithm can be implemented in two variants:

- **Non-Preemptive SJF**: Once a process is selected, it runs to completion without being interrupted.
- **Preemptive SJF (Shortest Remaining Time First - SRTF)**: The currently running process can be preempted if a new process arrives with a burst time shorter than the remaining time of the current process.

Key Terms:

- **Arrival Time (AT):** The time at which a process arrives in the ready queue.
- **Burst Time (BT):** The total time required by a process for execution on the CPU.
- **Waiting Time (WT):** The total time a process has been in the ready queue.
- **Turnaround Time (TAT):** The total time taken from submission to completion of the process.

Formulas:

- **Waiting Time (WT) = Turnaround Time (TAT) - Burst Time (BT)**
- **Turnaround Time (TAT) = Completion Time (CT) - Arrival Time (AT)**

DEMONSTRATION:

Program 1:(NonPreemptive)

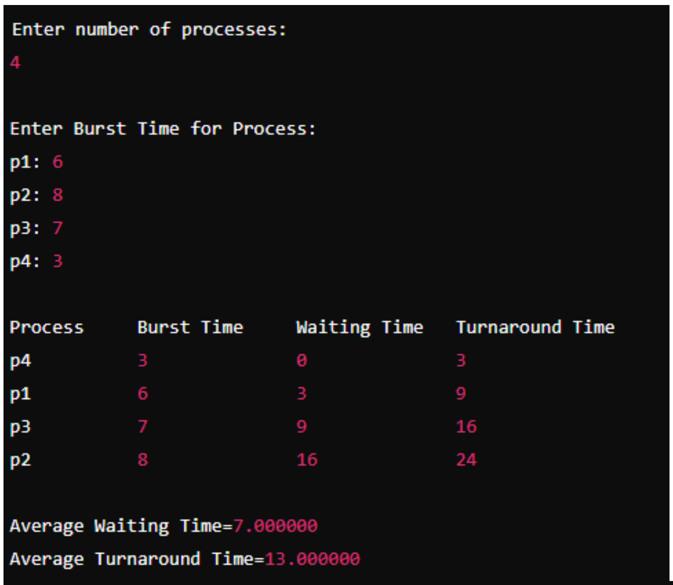
```
#include <stdio.h>
int main() {
   int bt[20], p[20], wt[20], tat[20], i, j, n, total = 0, pos, temp;
   float avg_wt, avg_tat;

   printf("Enter number of processes:\n");
   scanf("%d", &n);
   printf("\nEnter Burst Time for Process:\n");
   for(i = 0; i < n; i++) {
      printf("p%d: ", i + 1);
      scanf("%d", &bt[i]);
      p[i] = i + 1;  // Process number
   }

   // Sorting processes by burst time using selection sort
   for(i = 0; i < n; i++) {
      pos = i;
```

```
        for(j = i + 1; j < n; j++) {
            if(bt[j] < bt[pos]) {
                pos = j;
            }
        }

        temp = bt[i];
        bt[i] = bt[pos];
        bt[pos] = temp;

        temp = p[i];
        p[i] = p[pos];
        p[pos] = temp;
    }

    wt[0] = 0; // Waiting time for the first process is 0

    // Calculating waiting time
    for(i = 1; i < n; i++) {
        wt[i] = 0;
        for(j = 0; j < i; j++) {
            wt[i] += bt[j];
        }
        total += wt[i];
    }

    avg_wt = (float)total / n;
    total = 0;

    printf("\nProcess \t Burst Time \t Waiting Time \t Turnaround Time\n");
    for(i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
        total += tat[i];
        printf("p%d\t\t %d\t\t %d\t\t\t%d\n", p[i], bt[i], wt[i], tat[i]);
    }

    avg_tat = (float)total / n;
    printf("\n\nAverage Waiting Time=%f\n", avg_wt);
    printf("\nAverage Turnaround Time=%f\n", avg_tat);

    return 0;
}
```

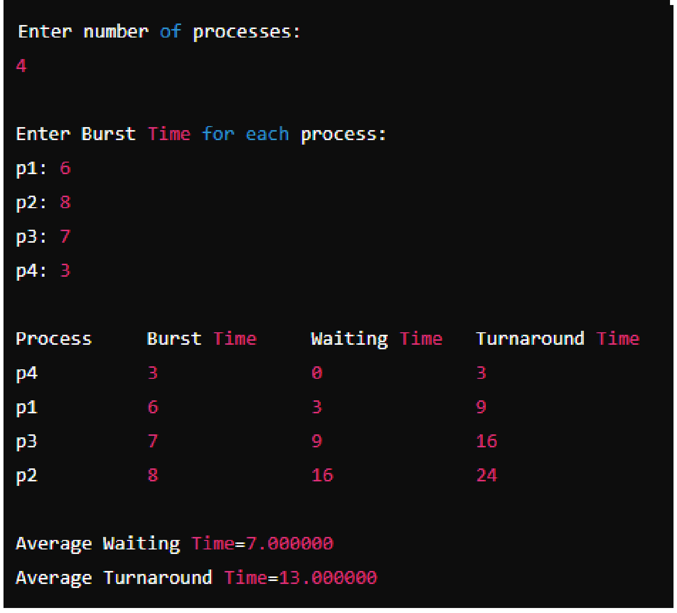
PROGRAM 2:(PREEMPTIVE)

```
#include <stdio.h>

int main() {
    int n, bt[20], p[20], wt[20], tat[20], rem_bt[20], i, min, finish_time, shortest = 0, complete = 0, t = 0;
    float avg_wt = 0, avg_tat = 0;
    printf("Enter number of processes:\n");
    scanf("%d", &n);
    printf("\nEnter Burst Time for each process:\n");
    for(i = 0; i < n; i++) {
        printf("p%d: ", i + 1);
        scanf("%d", &bt[i]);
        rem_bt[i] = bt[i];  // Initialize remaining burst time
        p[i] = i + 1;
    }

    while (complete != n) {
        min = 9999;
        for (i = 0; i < n; i++) {
            if ((rem_bt[i] > 0) && (rem_bt[i] < min)) {
                min = rem_bt[i];
                shortest = i;
            }
        }

        rem_bt[shortest]--;
        if (rem_bt[shortest] == 0) {
            complete++;
            finish_time = t + 1;
            wt[shortest] = finish_time - bt[shortest];
            if (wt[shortest] < 0)
                wt[shortest] = 0;
            tat[shortest] = bt[shortest] + wt[shortest];
            avg_wt += wt[shortest];
            avg_tat += tat[shortest];
        }
        t++;
    }

    avg_wt /= n;
    avg_tat /= n;

    printf("\nProcess \t Burst Time \t Waiting Time \t Turnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("p%d\t\t %d\t\t %d\t\t\t%d\n", p[i], bt[i], wt[i], tat[i]);
    }

    printf("\n\nAverage Waiting Time=%f\n", avg_wt);
    printf("\nAverage Turnaround Time=%f\n", avg_tat);

    return 0;
}
```

OUTPUT AND DISCUSSION:



```
Enter number of processes:
4

Enter Burst Time for Process:
p1: 6
p2: 8
p3: 7
p4: 3

Process       Burst Time      Waiting Time    Turnaround Time
p4            3               0               3
p1            6               3               9
p3            7               9               16
p2            8               16              24

Average Waiting Time=7.000000
Average Turnaround Time=13.000000
```



```
Enter number of processes:
4

Enter Burst Time for each process:
p1: 6
p2: 8
p3: 7
p4: 3

Process       Burst Time      Waiting Time    Turnaround Time
p4            3               0               3
p1            6               3               9
p3            7               9               16
p2            8               16              24

Average Waiting Time=7.000000
Average Turnaround Time=13.000000
```

In this lab, both non-preemptive and preemptive versions of the SJF scheduling algorithm were implemented. The outputs demonstrate how SJF works by selecting the shortest job available, reducing the average waiting and turnaround times.
Non-Preemptive SJF: Processes are executed in sequence based on their burst time, which minimizes the average waiting time but may cause longer processes to wait if shorter jobs continue to arrive.

- **Preemptive SJF (SRTF)**: The process with the shortest remaining burst time is always selected, even if it means preempting the currently running process. This can further reduce the waiting time but may lead to increased context switching overhead.

CONCLUSION:

Both versions of the SJF algorithm are effective in minimizing average waiting and turnaround times compared to FCFS. While the non-preemptive SJF is simpler, the preemptive SJF can yield better performance in dynamic environments but at the cost of complexity and potential process starvation.

LAB NO :4
IMPLEMENTATION OF ROUND ROBIN PROCESS SCHEDULING ALGORITHM IN C

OBJECTIVE:

- ➢ To implement the Round Robin (RR) scheduling algorithm in C and analyze its performance in terms of waiting time, turnaround time, and overall process efficiency.

THEORY:

Round Robin (RR) is a CPU scheduling algorithm where each process is assigned a fixed time slice, known as the time quantum, during which it can execute. If a process does not complete within its allocated time quantum, it is preempted and placed back in the ready queue, allowing the next process to execute. This continues in a cyclic order until all processes are completed.

Key Terms:

- **Arrival Time (AT):** The time at which a process arrives in the ready queue.
- **Burst Time (BT):** The total time required by a process for execution on the CPU.
- **Time Quantum:** The fixed time period allocated to each process.
- **Waiting Time (WT):** The total time a process has been in the ready queue, excluding the time during execution.
- **Turnaround Time (TAT):** The total time taken from submission to completion of the process.

Formulas:

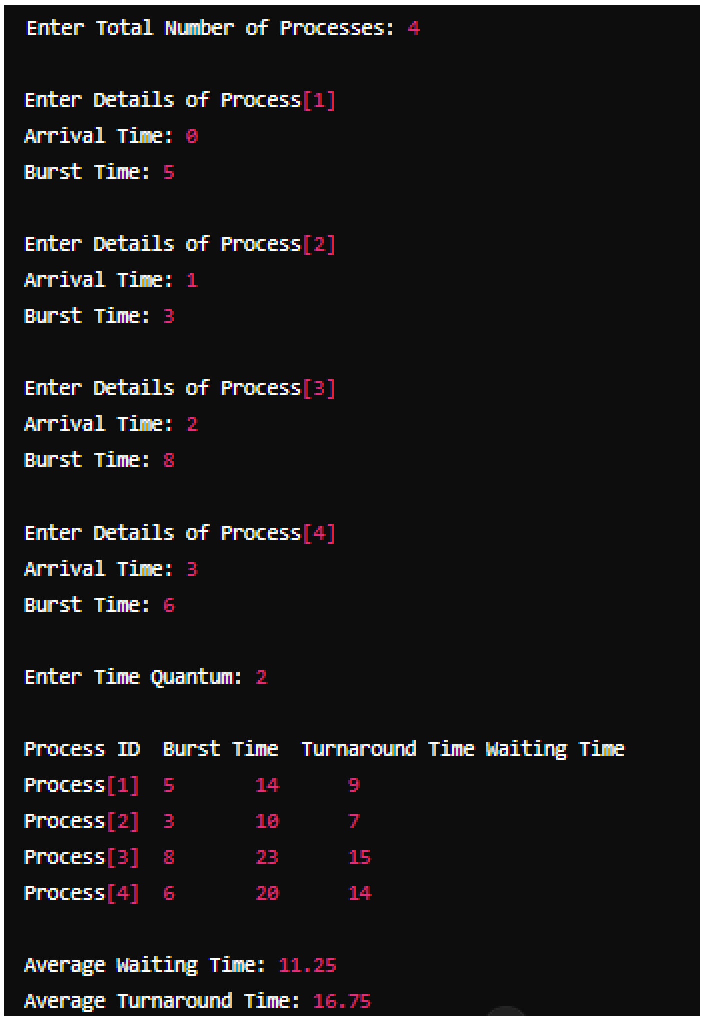
- **Waiting Time (WT) = Turnaround Time (TAT) - Burst Time (BT)**
- **Turnaround Time (TAT) = Completion Time (CT) - Arrival Time (AT)**

DEMONSTRATION:

```
#include <stdio.h>

int main() {
   int i, limit, total = 0, x, counter = 0, time_quantum;
   int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
   float average_wait_time, average_turnaround_time;

   printf("\nEnter Total Number of Processes: ");
   scanf("%d", &limit);
   x = limit;
```

```
    for(i = 0; i < limit; i++) {
        printf("\nEnter Details of Process[%d]\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &arrival_time[i]);
        printf("Burst Time: ");
        scanf("%d", &burst_time[i]);
        temp[i] = burst_time[i];
    }

    printf("\nEnter Time Quantum: ");
    scanf("%d", &time_quantum);

    printf("\nProcess ID\tBurst Time\tTurnaround Time\tWaiting Time\n");

    for(total = 0, i = 0; x != 0;) {
        if(temp[i] <= time_quantum && temp[i] > 0) {
            total += temp[i];
            temp[i] = 0;
            counter = 1;
        } else if(temp[i] > 0) {
            temp[i] -= time_quantum;
            total += time_quantum;
        }

        if(temp[i] == 0 && counter == 1) {
            x--;
            printf("Process[%d]\t%d\t\t%d\t\t%d\n", i + 1, burst_time[i], total - arrival_time[i], total - 
arrival_time[i] - burst_time[i]);
            wait_time += total - arrival_time[i] - burst_time[i];
            turnaround_time += total - arrival_time[i];
            counter = 0;
        }

        if(i == limit - 1) {
            i = 0;
        } else if(arrival_time[i + 1] <= total) {
            i++;
        } else {
            i = 0;
        }
    }

    average_wait_time = (float)wait_time / limit;
    average_turnaround_time = (float)turnaround_time / limit;

    printf("\nAverage Waiting Time: %.2f", average_wait_time);
    printf("\nAverage Turnaround Time: %.2f\n", average_turnaround_time);

    return 0;
}
```

OUTPUT AND DISCUSSION:



```
Enter Total Number of Processes: 4

Enter Details of Process[1]
Arrival Time: 0
Burst Time: 5

Enter Details of Process[2]
Arrival Time: 1
Burst Time: 3

Enter Details of Process[3]
Arrival Time: 2
Burst Time: 8

Enter Details of Process[4]
Arrival Time: 3
Burst Time: 6

Enter Time Quantum: 2

Process ID  Burst Time  Turnaround Time Waiting Time
Process[1]  5       14      9
Process[2]  3       10      7
Process[3]  8       23      15
Process[4]  6       20      14

Average Waiting Time: 11.25
Average Turnaround Time: 16.75
```

In this lab, the Round Robin scheduling algorithm was implemented using a fixed time quantum. The output shows how the algorithm works by cycling through the processes, allowing each to execute for a given time quantum before moving to the next.

- **Time Quantum Selection:** The selection of an appropriate time quantum is crucial. If the time quantum is too small, the algorithm behaves similarly to FCFS with a lot of context switching, leading to high overhead. If it is too large, it behaves like FCFS.
- **Waiting Time and Turnaround Time:** The Round Robin algorithm provides a fair allocation of CPU time, which can reduce the waiting time for processes that arrive later in the ready queue. The average waiting time and turnaround time were calculated and displayed, providing insight into the efficiency of the algorithm for the given time quantum.

CONCLUSION:

The Round Robin scheduling algorithm is effective in providing fair CPU time allocation to all processes, especially in a time-shared system. By carefully selecting the time quantum, this algorithm can reduce the waiting time and improve the overall responsiveness of the system.