Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre

A Lab Report

on

"Understanding fork() and Process Creation and Simulation of Process Scheduling Algorithm"

[Code No.: COMP 304]

Submitted by

Miraj Sapkota
Roll No.: 46

Submitted to
Rabina Shrestha
Department of Computer Science and Engineering

January 11, 2026

Lab2: Understanding fork() and Process Creation

Objective

To study how `fork()` creates new processes and understand why multiple `fork()` calls increase the number of processes exponentially.

1 Program 1: Creation of Process

1.1 Two fork() calls

1.1.1 Modification

The program is modified to include exactly two `fork()` calls.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("hello\n");

    fork();
    fork();

    printf("Hello world\n");
    return 0;
}
```

1.1.2 Output Analysis

When we run this program, ”Hello world” is printed a total of 4 times.

1.1.3 Process Tree Diagram and Analysis

```
       (Parent P)
           |
      (Fork 1)
       /      \
     (P)       (C1)
      |          |
   (Fork 2)   (Fork 2)
    /   \      /    \
  (P)   (C2) (C1)   (C3)
```

- **First fork():** The original parent process (P) creates a new child (C1). (Total: 2 process).

- **Second fork():** Both the parent (P) and the first child (C1) execute the instruction. P creates C2 and C1 creates C3. (Total: 4 process).
- As a result, the print statement is executed by all 4 process.

1.2 Three fork() calls

1.2.1 Modification

The program was modified to include three `fork()` calls.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("hello\n");

    fork();
    fork();
    fork();

    printf("Hello world\n");
    return 0;
}
```

1.2.2 Output Analysis

When we run this program, ”Hello world” is printed 8 times.

1.2.3 Process Tree Diagram

```
                 (P)
                  |
             (Fork 1)
            /        \
          (P)        (C1)
          / \        /  \
        (     Fork 2      )
      /   \        /    \
    (P)  (C2)    (C1)  (C3)
    / \   / \    / \    / \
   (        Fork 3         )
 /  \  /  \  /  \  /  \
P  C4 C2  C6 C1  C5 C3  C7
```

1.2.4 Analysis

The number of processes grows exponentially based on the formula 2^n, where n is the number of fork calls.

- **Fork 1:** 2 processes.

- **Fork 2:** 4 processes.
- **Fork 3:** 8 processes.

Each `fork()` duplicates every currently running process. In each step no. of process doubles.

2 Understanding fork() with PID

Objective: A program that shows the use of forking and identifying Parent vs. Child processes using PIDs.

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid;
    printf("parent Process with ID %d\n", getpid());

    printf("before fork\n");

    pid = fork();

    printf("after fork\n");

    if (pid == 0) {
        printf("child process with ID %d\n", getpid());
    }
    else {
        printf("parent process with PID %d\n", getpid());
    }
    return 0;
}
```

Output

```
parent Process with ID 323345
before fork
after fork
parent process with PID 323345
after forking
child process with ID 323346
```

Questions

Q1: Explain the difference between `pid == 0` and `pid > 0`. Which one corresponds to the child and the parent?

- `pid == 0`: This means child process. The fork() system call returns 0 to the newly created child process so it knows it is the child.
- `pid > 0`: This corresponds to the Parent Process. The `fork()` system call returns the Process ID (a positive integer) of the new child to the parent process.

Q2: Why do the parent and child processes print their own PID differently?

Answer: They print different PIDs because they are two separate processes handled by the operating system. The parent retains its original PID (394681), while the OS assigns a new unique PID (394682) to the child process.

Q3: Explain why the output order is parent lines first, child lines after. Can the order change? Why?

- Current Order: The parent printed first because the CPU scheduler allowed the parent to continue execution immediately after the fork while the child was being initialized.
- It can change the order is non-deterministic.
- It depends on the OS Scheduling algorithm because if a context switching occurs right after `fork()`, the child might run before the parent.

Lab Report 3: Simulation of Process Scheduling Algorithm

3 Scheduling

3.1 Preemptive vs. Non-preemptive Scheduling

- **Non-preemptive Scheduling:** In this mode, once a process has been allocated the CPU, it keeps the CPU until it releases it either by terminating or by switching to the waiting state. The operating system cannot forcibly take the CPU away. Examples include First-Come-First-Serve (FCFS) and standard Shortest Job First (SJF).

- **Preemptive Scheduling:** In this mode, the operating system can interrupt a currently running process and move it to the ready queue to allocate the CPU to another process. This is often done based on priority or time quantum. Examples include Round Robin (RR) and Shortest Remaining Time First (SRTF).

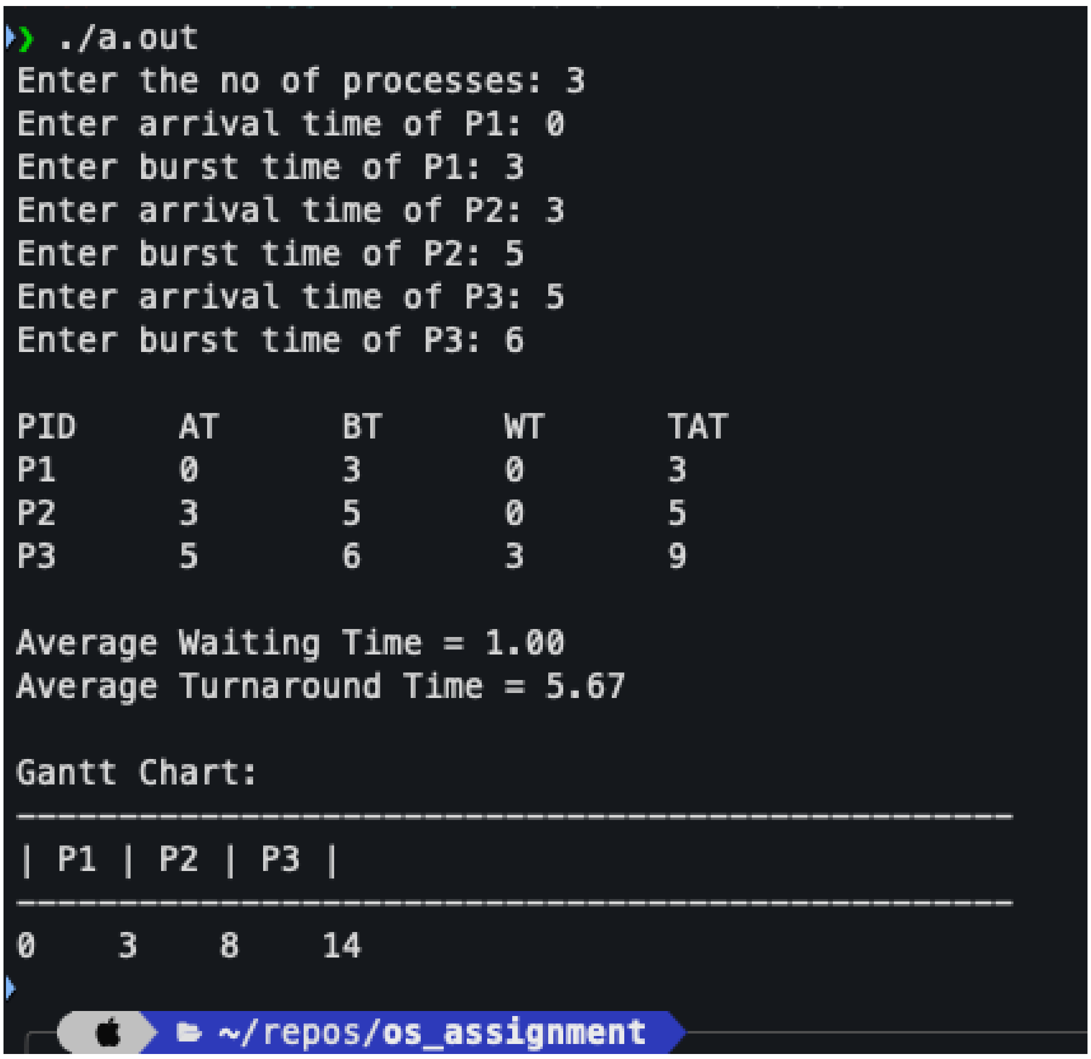
4 Shortest Job First (SJF)

Algorithm Overview: SJF is a non-preemptive scheduling algorithm that selects the waiting process with the smallest execution time (burst time) to execute next. This approach is optimal for minimizing the average waiting time for a given set of processes. However, it can lead to starvation for long processes if shorter processes keep arriving.

4.1 Source Code

```
#include <stdio.h>

int main() {
    int n, i, time = 0, completed = 0;

    int burst_time[20], arrival_time[20], waiting_time[20], turnaround_ti
    int is_completed[20] = {0};


    int gantt_pid[20], gantt_start[20], gantt_end[20];
    int gantt_index = 0;

    float avg_waiting_time = 0;
    float avg_turn_around_time = 0;

    printf("Enter the no of processes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("Enter arrival time of P%d: ", pid[i]);
        scanf("%d", &arrival_time[i]);
        printf("Enter burst time of P%d: ", pid[i]);
        scanf("%d", &burst_time[i]);
    }

    while (completed != n) {
        int index = -1;
        int min_burst_time = 99999;

        for (i = 0; i < n; i++) {
            if (arrival_time[i] <= time && !is_completed[i]) {
                if (burst_time[i] < min_burst_time) {
                    min_burst_time = burst_time[i];
                    index = i;
                }
            }
        }

        if (index == -1) {
            time++;
        } else {
            gantt_pid[gantt_index] = pid[index];
            gantt_start[gantt_index] = time;

            waiting_time[index] = time - arrival_time[index];
            time += burst_time[index];
            turnaround_time[index] = waiting_time[index] + burst_time[ind

            gantt_end[gantt_index] = time;
            gantt_index++;

            is_completed[index] = 1;
            completed++;
        }
    }

    printf("\nPID\tAT\tBT\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        avg_waiting_time += waiting_time[i];
        avg_turn_around_time += turnaround_time[i];

        printf("P%d\t%d\t%d\t%d\t%d\n",
               pid[i], arrival_time[i], burst_time[i],
               waiting_time[i], turnaround_time[i]);
    }

    printf("\nAverage Waiting Time = %.2f", avg_waiting_time / n);
    printf("\nAverage Turnaround Time = %.2f\n", avg_turn_around_time / n


    printf("\nGantt Chart:\n");
    printf("-----------------------------------------------------\n");

    for (i = 0; i < gantt_index; i++) {
        printf("| P%d ", gantt_pid[i]);
    }
    printf("|\n");

    printf("-----------------------------------------------------\n");
    printf("%d", gantt_start[0]);
    for (i = 0; i < gantt_index; i++) {
        printf("    %d", gantt_end[i]);
    }
    printf("\n");

    return 0;
}
```

4.2 Output



```
❯ ./a.out
Enter the no of processes: 3
Enter arrival time of P1: 0
Enter burst time of P1: 3
Enter arrival time of P2: 3
Enter burst time of P2: 5
Enter arrival time of P3: 5
Enter burst time of P3: 6

PID     AT      BT      WT      TAT
P1      0       3       0       3
P2      3       5       0       5
P3      5       6       3       9

Average Waiting Time = 1.00
Average Turnaround Time = 5.67

Gantt Chart:
-------------------------------------------------
| P1 | P2 | P3 |
-------------------------------------------------
0    3    8    14
```

Figure 1: SJF Simulation

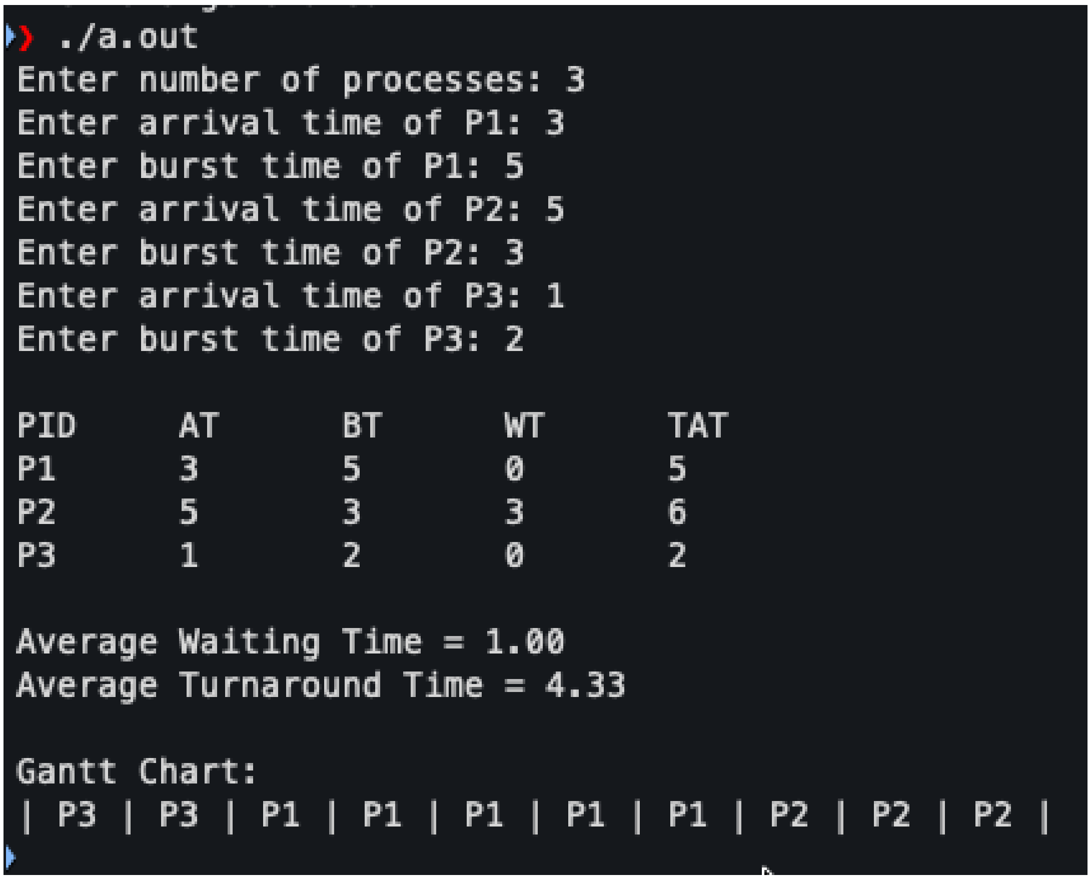
5 Shortest Remaining Time First (SRTF)

Algorithm Overview: SRTF is the **preemptive** version of the SJF algorithm. In this method, the process with the smallest amount of time remaining until completion is selected to execute. If a new process arrives with a CPU burst time less than the remaining time of the currently executing process, the current process is preempted.

5.1 Source Code

```
int main() {
    int n, i, time = 0, completed = 0;
    int burst_time[20], remaining_time[20], arrival_time[20];
    int waiting_time[20], turnaround_time[20], pid[20];
    int is_completed[20] = {0};

    // Gantt chart
    int gantt_pid[100], gantt_time[100];
    int g = 0;

    float avg_wt = 0, avg_tat = 0;

    printf("Enter number of processes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("Enter arrival time of P%d: ", pid[i]);
        scanf("%d", &arrival_time[i]);
        printf("Enter burst time of P%d: ", pid[i]);
        scanf("%d", &burst_time[i]);
        remaining_time[i] = burst_time[i];
    }

    while (completed != n) {
        int index = -1;
        int min_rt = 99999;

        for (i = 0; i < n; i++) {
            if (arrival_time[i] <= time && remaining_time[i] > 0) {
                if (remaining_time[i] < min_rt) {
                    min_rt = remaining_time[i];
                    index = i;
                }
            }
        }

        if (index == -1) {
            time++;
        } else {
            gantt_pid[g] = pid[index];
            gantt_time[g++] = time;

            remaining_time[index]--;
            time++;

            if (remaining_time[index] == 0) {
                completed++;
                turnaround_time[index] = time - arrival_time[index];
                waiting_time[index] = turnaround_time[index] - burst_time
            }
        }
    }

    printf("\nPID\tAT\tBT\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        avg_wt += waiting_time[i];
        avg_tat += turnaround_time[i];
        printf("P%d\t%d\t%d\t%d\t%d\n",
               pid[i], arrival_time[i], burst_time[i],
               waiting_time[i], turnaround_time[i]);
    }

    printf("\nAverage Waiting Time = %.2f", avg_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", avg_tat / n);

    // Gantt Chart
    printf("\nGantt Chart:\n");
    for (i = 0; i < g; i++) {
        printf("| P%d ", gantt_pid[i]);
    }
    printf("|\n");

    return 0;
}
```

5.2 Output



```
❯ ./a.out
Enter number of processes: 3
Enter arrival time of P1: 3
Enter burst time of P1: 5
Enter arrival time of P2: 5
Enter burst time of P2: 3
Enter arrival time of P3: 1
Enter burst time of P3: 2

PID     AT      BT      WT      TAT
P1      3       5       0       5
P2      5       3       3       6
P3      1       2       0       2

Average Waiting Time = 1.00
Average Turnaround Time = 4.33

Gantt Chart:
| P3 | P3 | P1 | P1 | P1 | P1 | P1 | P2 | P2 | P2 |
```

Figure 2: SRTF Simulation

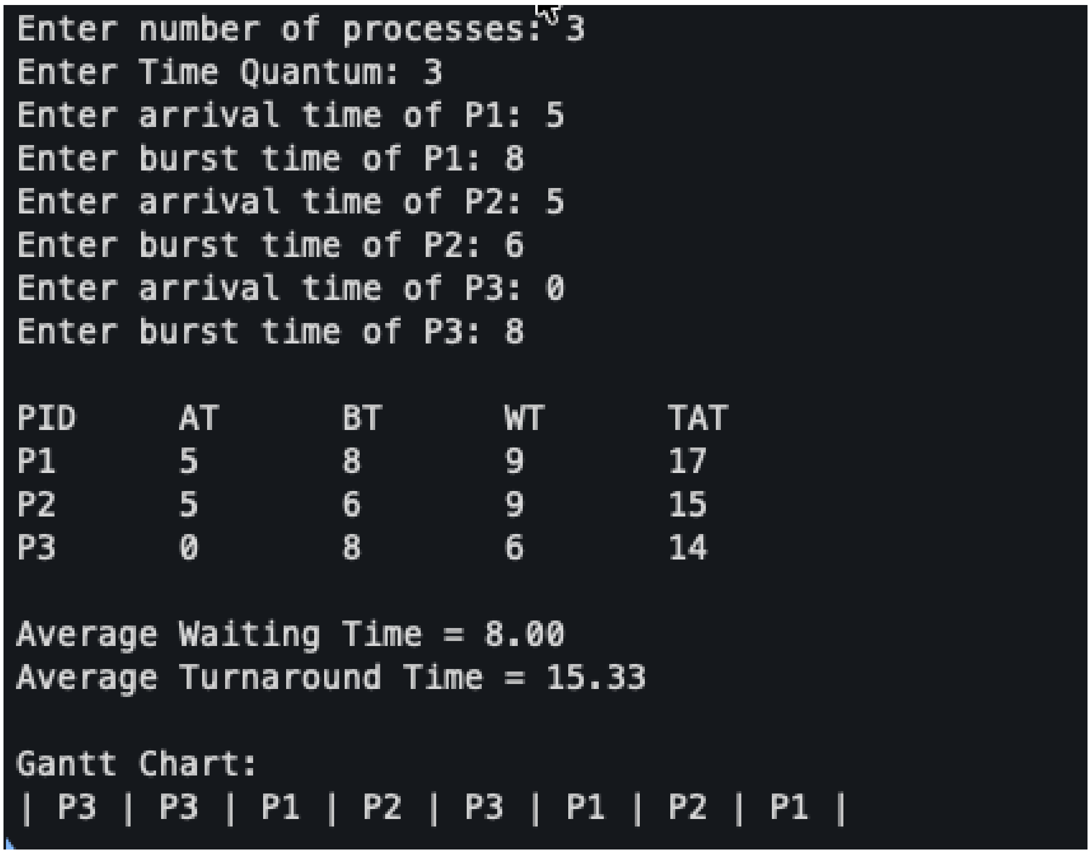
6 Round Robin (RR)

Algorithm Overview: Round Robin is a preemptive scheduling algorithm designed specifically for time-sharing systems. Each process is assigned a small unit of time called a Time Quantum (or Time Slice). The CPU scheduler cycles through the ready queue, allocating the CPU to each process for a time interval of up to one time quantum. If the process does not finish within the quantum, it is preempted and placed at the back of the ready queue.

6.1 Source Code

```
#include <stdio.h>

int main() {
    int n, i, time = 0, tq;
    int burst_time[20], remaining_time[20], arrival_time[20];
    int waiting_time[20] = {0}, turnaround_time[20], pid[20];
    int completed = 0;

    // Gantt chart
    int gantt_pid[100], gantt_time[100];
    int g = 0;

    float avg_wt = 0, avg_tat = 0;

    printf("Enter number of processes: ");
    scanf("%d", &n);

    printf("Enter Time Quantum: ");
    scanf("%d", &tq);

    for (i = 0; i < n; i++) {
        pid[i] = i + 1;
        printf("Enter arrival time of P%d: ", pid[i]);
        scanf("%d", &arrival_time[i]);
        printf("Enter burst time of P%d: ", pid[i]);
        scanf("%d", &burst_time[i]);
        remaining_time[i] = burst_time[i];
    }

    while (completed != n) {
        int done = 1;

        for (i = 0; i < n; i++) {
            if (arrival_time[i] <= time && remaining_time[i] > 0) {
                done = 0;

                gantt_pid[g] = pid[i];
                gantt_time[g++] = time;

                if (remaining_time[i] > tq) {
                    time += tq;
                    remaining_time[i] -= tq;
                } else {
                    time += remaining_time[i];
                    waiting_time[i] = time - arrival_time[i] - burst_time
                    remaining_time[i] = 0;
                    completed++;
                }
            }
        }

        if (done)
            time++;
    }

    for (i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + burst_time[i];
        avg_wt += waiting_time[i];
        avg_tat += turnaround_time[i];
    }

    printf("\nPID\tAT\tBT\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\n",
               pid[i], arrival_time[i], burst_time[i],
               waiting_time[i], turnaround_time[i]);
    }

    printf("\nAverage Waiting Time = %.2f", avg_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", avg_tat / n);

    // Gantt Chart
    printf("\nGantt Chart:\n");
    for (i = 0; i < g; i++) {
        printf("| P%d ", gantt_pid[i]);
    }
    printf("|\n");

    return 0;
}
```

6.2 Output



```
Enter number of processes: 3
Enter Time Quantum: 3
Enter arrival time of P1: 5
Enter burst time of P1: 8
Enter arrival time of P2: 5
Enter burst time of P2: 6
Enter arrival time of P3: 0
Enter burst time of P3: 8

PID     AT      BT      WT      TAT
P1      5       8       9       17
P2      5       6       9       15
P3      0       8       6       14

Average Waiting Time = 8.00
Average Turnaround Time = 15.33

Gantt Chart:
| P3 | P3 | P1 | P2 | P3 | P1 | P2 | P1 |
```

Figure 3: Robin Simulation