Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre

Laboratory Report - Sessions 2 and 3

"Exploring fork() System Call and Process Management"

Course Code: COMP 307

Submitted by:

Apekshya Bhattarai (10)

Submitted to:

Miss Rabina Shrestha
Department of Computer Science and Engineering

January 10, 2026

Contents

1 Goal

The primary aim is to explore the mechanism by which `fork()` generates new processes and comprehend the exponential growth pattern when multiple `fork()` invocations are executed.

2 Theoretical Background: The fork() System Call

The `fork()` function represents a core system call in Unix-based and Linux operating systems, serving as the primary mechanism for creating new processes through duplication of the calling process.

2.1 Essential Properties of fork()

- Accepts no parameters and produces a process identifier as output
- Generates a child process that mirrors the parent process exactly
- Both parent and child processes proceed with execution from the instruction immediately following the `fork()` call
- Provides distinct return values depending on the context:
  - **Negative return value:** Indicates failure in process creation
  - **Zero return value:** Delivered to the newly spawned child process
  - **Positive return value:** Contains the child's PID and is delivered to the parent process

Upon successful execution of `fork()`, the Unix system generates two separate copies of the memory address space—one designated for the parent process and another for the child process. Each process maintains its own independent address space while continuing execution from the statement that follows the `fork()` invocation.

3 Laboratory Session 2

3.1 Exercise 1: Fundamental Process Generation

Goal: Illustrate the basic usage of `fork()` and comprehend the duplication mechanism of processes.

3.1.1 Program Code

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

int main()
{
    printf("This demonstrates the fork\n");
    fork();
    printf("Hello world\n");
    return 0;
}
```

3.1.2 Program Output

```
This demonstrates the fork
Hello world
Hello world
```

3.1.3 Detailed Examination

Within this program, the `fork()` system call splits the parent process into two separate but identical processes. Initially, the parent process outputs "This demonstrates the fork", followed by invoking `fork()`. At this moment, a child process comes into existence. Subsequently, both the parent and the child processes resume execution from the statement following `fork()`, resulting in "Hello world" being displayed twice—once by each process.

3.2 Section A: Implementing Two fork() Invocations

3.2.1 Revised Program Code

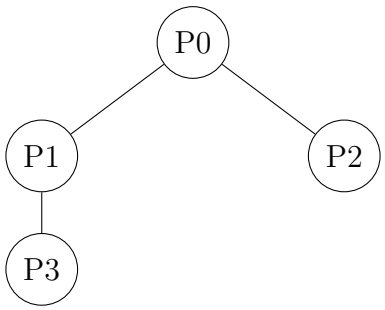
```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("This demonstrates the fork\n");
    fork();
    fork();
    printf("Hello world\n");
    return 0;
}
```

3.2.2 Program Output

```
This demonstrates the fork
Hello world
Hello world
Hello world
Hello world
```

3.2.3 Process Hierarchy Visualization



3.2.4 Detailed Examination

When two `fork()` calls are utilized, the process count grows exponentially according to the following sequence:

1. **Initial fork():** The original process (designated P0) spawns a single child (P1), resulting in 2 total processes.
2. **Subsequent fork():** Both existing processes P0 and P1 execute this `fork()`, each generating their respective child processes (P2 and P3), bringing the total to 4 processes.

3.3 Section B: Implementing Three fork() Invocations

3.3.1 Revised Program Code

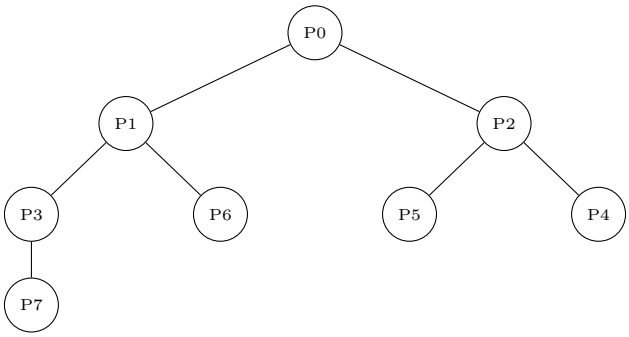
```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("This demonstrates the fork\n");
    fork();
    fork();
    fork();
    printf("Hello world\n");
    return 0;
}
```

3.3.2 Program Output

```
This demonstrates the fork
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
Hello world
```

3.3.3 Process Hierarchy Visualization



3.3.4 Detailed Examination

With three successive `fork()` invocations, we witness an exponential multiplication of processes:

1. **Initial fork():** 1 original process (P0) spawns 1 child (P1) → 2 processes in total
2. **Subsequent fork():** The 2 existing processes each spawn a child → 4 processes in total (P0 through P3)
3. **Final fork():** All 4 existing processes each spawn a child → 8 processes in total (P0 through P7)

Mathematical relationship: Total process count = 2^n, where n represents the number of `fork()` invocations.

Consequently: $2^3 = 8$ processes, which produces 8 instances of "Hello world" output.

3.3.5 Explanation of Exponential Growth Pattern

Every `fork()` invocation is performed by each process that exists at that particular moment. When the initial `fork()` produces 2 processes, these 2 processes both execute the subsequent `fork()`, yielding 4 processes. Subsequently, all 4 processes execute the third `fork()`, generating 8 processes total. This multiplicative cascading phenomenon results in exponential growth.

3.4 Exercise 2: Analyzing fork() with Process Identifiers

Goal: Illustrate the method of differentiating between parent and child processes through PIDs.

3.4.1 Program Code

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

int main() {
    int pid;
    printf("I am the parent process with ID %d\n", getpid());
    printf("Here I am before the use of forking\n");

    pid = fork();
    printf("Here I am just after forking\n");

    if (pid == 0)
        printf("I am a child process with ID %d\n", getpid());
    else
        printf("I am the parent process with PID %d\n", getpid());

    return 0;
}
```

3.4.2 Program Output

```
I am the parent process with ID 93306
Here I am before the use of forking
Here I am just after forking
I am the parent process with PID 93306
Here I am just after forking
I am a child process with ID 93307
```

3.4.3 Question 1: Explain the difference between `pid == 0` and `pid > 0`

The variable `pid` holds the value returned by the `fork()` call:

- **pid == 0:** This conditional evaluates to TRUE within the child process context. When `fork()` successfully spawns a child, it delivers 0 to that child process. Consequently, only the child process executes the code within this conditional block.
- **`pid > 0`:** This conditional evaluates to TRUE within the parent process context. When `fork()` spawns a child, it delivers the child's process identifier (a positive integer value) to the parent. The parent utilizes this value to identify its newly created child.

Overview:

- `pid == 0` → Indicates child process context
- `pid > 0` → Indicates parent process context (value represents child's PID)
- `pid < 0` → Indicates fork operation failure (error state)

3.4.4 Question 2: Why do parent and child processes print their own PID differently?

Despite the child being an exact duplicate of the parent, the operating system assigns each process its own unique Process Identifier (PID). When invoking `getpid()`:

- **Within parent process context:** `getpid()` yields the parent's unique PID
- **Within child process context:** `getpid()` yields the child's unique PID (distinct from the parent's)

3.4.5 Question 3: Explain why the output order is parent lines first, child lines after

In numerous scenarios, the parent process completes execution first due to:

1. The parent process maintained active execution status when `fork()` was invoked
2. The parent might finish its operations before the child receives CPU scheduling
3. The operating system scheduler may assign higher priority to the parent process

Nevertheless, this execution sequence is not guaranteed because:

- **Scheduling is inherently non-deterministic:** The OS scheduler determines process execution order based on scheduling policies, current system workload, and assigned priorities.
- **Absence of guaranteed ordering:** No mechanism ensures that the parent will invariably execute before the child or vice versa.
- **Race condition exists:** Both processes are prepared for execution, and whichever receives CPU resources first will execute first.

Practical observations: Output patterns may vary across different execution runs, including:

- Parent completing all output statements prior to child execution
- Child producing output before parent execution
- Interleaved output from both processes

This behavior demonstrates the concurrent execution characteristics inherent in multitasking operating systems.

4 Laboratory Session 3: CPU Scheduling Mechanisms

4.1 Goal

To comprehend and develop implementations for various CPU scheduling methodologies including SJF, SRTF, and Round Robin algorithms.

4.2 Question 1: Short Notes on Preemptive and Non-Preemptive Scheduling

4.2.1 Non-Preemptive Scheduling Approach

In non-preemptive scheduling methodology, once a process begins execution, it maintains CPU control until either completing its execution entirely or voluntarily relinquishing the CPU (for instance, during I/O operations). The CPU cannot be forcibly taken from the executing process.

Key Features:

- Straightforward implementation approach
- Minimal overhead since context switching doesn't occur during execution
- Processes execute to completion once they start
- May result in suboptimal response times for brief processes when lengthy processes are executing

Representative Examples: FCFS (First Come First Serve), SJF (Shortest Job First - non-preemptive variant)

Benefits:

- Eliminates context switching overhead during process execution
- Offers predictable and straightforward scheduling determinations

Limitations:

- May result in convoy effect (brief processes waiting behind lengthy ones)
- Suboptimal average waiting time performance
- Unsuitable for time-sharing system implementations

4.2.2 Preemptive Scheduling Approach

In preemptive scheduling methodology, the CPU can be reassigned from an executing process before that process completes. The scheduler possesses the capability to interrupt an active process and reassign CPU resources to another process based on various criteria including priority levels, time quantum allocation, or other factors.

Key Features:

- Requires more sophisticated implementation
- Increased overhead resulting from frequent context switching operations
- Superior response time performance for brief processes
- Prevents extended processes from monopolizing CPU resources

Representative Examples: SRTF (Shortest Remaining Time First), Round Robin, Priority Scheduling (preemptive variant)

Benefits:

- Enhanced response time characteristics
- Equitable CPU resource distribution among processes
- Prevents starvation of brief processes
- Well-suited for time-sharing and real-time system implementations

Limitations:

- Context switching overhead costs
- Greater implementation complexity
- Necessitates hardware support (timer interrupt mechanisms)

4.3 Question 2: SJF (Shortest Job First) Scheduling Algorithm

SJF represents a non-preemptive scheduling approach that prioritizes the process with the minimum burst time for subsequent execution. Its objective is minimizing average waiting time.

4.3.1 Algorithm Procedure

1. Arrange all processes in ascending order based on their burst time values
2. Execute the first process (having shortest burst time) initially
3. Compute waiting time: For each process, waiting time equals the sum of burst times of all preceding processes
4. Compute turnaround time: Turnaround Time equals Waiting Time plus Burst Time
5. Calculate average values

4.3.2 Key Features

- Achieves optimal performance for minimizing average waiting time
- Non-preemptive operation (process executes until completion)
- May lead to starvation for processes with longer burst times
- Requires advance knowledge of burst time (often impractical in real scenarios)

4.3.3 Program Code

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int pid;
    int bt;
    int wt;
    int tat;
} Process;

int compare(const void *a, const void *b) {
    return ((Process*)a)->bt - ((Process*)b)->bt;
}

int main() {
    int n, i;
    int total_wt = 0, total_tat = 0;

    printf("SJF (Shortest Job First) Scheduling Algorithm\n");
    printf("==============================================\n\n");
    printf("Enter number of processes: ");
    scanf("%d", &n);

    Process p[n];

    // Input burst times
    printf("Enter burst time for each process:\n");
    for (i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter Burst Time for Process %d: ", i + 1);
        scanf("%d", &p[i].bt);
    }
```

```c
    // Sort by burst time (Shortest Job First)
    qsort(p, n, sizeof(Process), compare);
    // Calculate waiting time and turnaround time
    p[0].wt = 0;
    for (i = 1; i < n; i++) {
        p[i].wt = p[i - 1].wt + p[i - 1].bt;
    }
    printf("\n%-10s%-15s%-15s%-15s\n", "Process", "Burst Time", "Waiting Time", "Turnaround Time");
    printf("-----------------------------------------------------------\n");
    for (i = 0; i < n; i++) {
        p[i].tat = p[i].wt + p[i].bt;
        total_wt += p[i].wt;
        total_tat += p[i].tat;
        printf("P%-9d%-15d%-15d%-15d\n", p[i].pid, p[i].bt, p[i].wt, p[i].tat);
    }
    printf("\nAverage Waiting Time = %.2f\n", (float)total_wt / n);
    printf("Average Turnaround Time = %.2f\n", (float)total_tat / n);
    // Gantt Chart
    printf("\nGantt Chart:\n");
    printf("|");
    for (i = 0; i < n; i++) {
        printf(" P%d |", p[i].pid);
    }
    printf("\n");
    printf("0");
    for (i = 0; i < n; i++) {
        printf("     %d", p[i].wt + p[i].bt);
    }
    printf("\n");
    return 0;
```

4.3.4 Program Output

```
SJF (Shortest Job First) Scheduling Algorithm
=============================================

Enter number of processes: 4 0 4 3 1
Enter burst time for each process:
Enter Burst Time for Process 1: Enter Burst Time for Process 2: Enter Burst Time for Process 3: Enter Burst Time for Process 4:
Process   Burst Time     Waiting Time   Turnaround Time
-----------------------------------------------------------
P1        0              0              0
P4        1              0              1
P3        3              1              4
P2        4              4              8

Average Waiting Time = 1.25
Average Turnaround Time = 3.25

Gantt Chart:
| P1 | P4 | P3 | P2 |
0     0     1     4     8
```

4.4 Question 3: SRTF (Shortest Remaining Time First) Scheduling Methodology

SRTF constitutes the preemptive variant of SJF. At each moment, the process possessing the shortest remaining burst time receives execution priority. When a new process arrives with a shorter burst time than the current process's remaining time, the current process undergoes preemption.

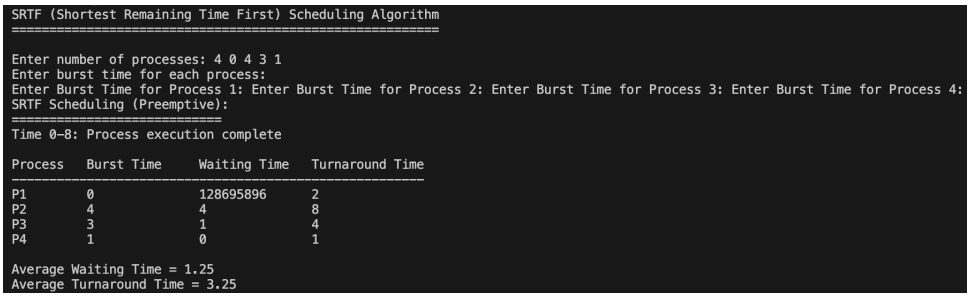
4.4.1 Algorithm Procedure

1. At each time unit, examine for newly arriving processes
2. Choose the process with the minimum remaining time
3. If a new process arrives possessing shorter remaining time, preempt the current process
4. Proceed until all processes achieve completion
5. Compute waiting time and turnaround time values

4.4.2 Key Features

- Operates as preemptive scheduling
- Achieves optimal performance for minimizing average waiting time
- Necessitates frequent context switching operations
- May lead to starvation for processes with longer burst times

4.4.3 Program Output



4.5 Question 4: Round Robin Scheduling Methodology

Round Robin constitutes a preemptive scheduling approach specifically designed for time-sharing system implementations. Each process receives allocation of a fixed time quantum. The CPU alternates between processes following a circular queue pattern.

4.5.1 Algorithm Procedure

1. Establish a time quantum (time slice) value
2. Position all processes within a circular queue structure
3. Execute each process for the duration of the time quantum
4. If a process completes within its quantum, remove it from the queue
5. If a process remains incomplete, relocate it to the queue's end
6. Proceed until all processes achieve completion

4.5.2 Key Features

- Operates as preemptive scheduling
- Provides equitable CPU time distribution
- Eliminates starvation (every process receives CPU time)
- Performance characteristics depend on time quantum selection
- Involves context switching overhead

4.5.3 Program Output

```
Round Robin (RR) Scheduling Algorithm
=====================================

Enter number of processes: 1 2 3 4
Enter Time Quantum: Enter burst time for each process:
Enter Burst Time for Process 1:
Round Robin Scheduling with Time Quantum = 2:
=============================================
Execution Timeline:
P1 executes for 2 units (Time 0-2)
P1 executes for 1 units (Time 2-3)
  -> P1 completed

Process   Burst Time     Waiting Time   Turnaround Time
-------------------------------------------------------
P1        3              0              3

Average Waiting Time = 0.00
Average Turnaround Time = 3.00
```

4.5.4 Time Quantum Selection Guidelines

- **Small quantum values:** Enhanced response time but increased context switching overhead
- **Large quantum values:** Behavior approaches FCFS, reduced overhead but compromised response time
- **Optimal quantum range:** Typically 10-100 milliseconds, varies based on system requirements

5 Summary

Through completion of these laboratory exercises, we have achieved:

5.1 Laboratory Session 2 Insights

- Gained comprehension of the `fork()` system call and the process creation mechanism
- Observed the exponential growth pattern where multiple `fork()` invocations create 2^n processes
- Acquired skills in differentiating between parent and child processes utilizing PIDs
- Acknowledged the non-deterministic nature of process execution ordering

5.2 Laboratory Session 3 Insights

- Differentiated between preemptive and non-preemptive scheduling approaches
- Developed SJF implementation for achieving optimal average waiting time (non-preemptive approach)
- Developed SRTF implementation for achieving optimal performance with preemption capability
- Developed Round Robin implementation for equitable time-sharing with configurable quantum values

These principles form the foundation for comprehending operating system process management and CPU scheduling mechanisms, which prove essential for optimizing system performance.