Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Labreport - II

[Code No.: COMP 307]

Operating System

Submitted by

Shreyash Mahato

Roll No: 33

Submitted to

Ms. Rabina Shrestha

Department of Computer Science and Engineering

January 11, 2026

1. Introduction

1.1. Overview

Within Unix-like environments, a process represents an active instance of a running program. Every process operates within its own allocated memory and system resources. To initiate new processes, we utilize the `fork()` system call, which essentially clones the existing parent process to generate a new entity known as the child process.

1.2. Learning Goals

1. Grasp the fundamental mechanics of how `fork()` spawns new processes.
2. Explore why the process count grows exponentially with successive `fork()` invocations.
3. Analyze the hierarchical link between parent and child processes.
4. Practice identifying unique processes through their PIDs (Process IDs).
5. Observe and document different patterns of process generation.

2. Theoretical Background

2.1. The Mechanics of fork()

The `fork()` system call is the primary method for process creation in Linux. When this function is executed, the operating system replicates the calling process, resulting in a parent and a nearly identical child process.

2.1.1. Interpreting Return Values

1. **Negative Value:** Indicates a failure in the creation of the process.
2. **Zero (0):** Signals that the code is currently executing within the child process.
3. **Positive Value:** Signals that the code is running in the parent process; the specific value represents the PID of the new child.

2.1.2. Operational Logic

1. The system performs a near-exact duplication of the calling process.
2. Both entities maintain independent memory spaces to ensure isolation.
3. Execution for both starts immediately after the `fork()` call.
4. We can use `getpid()` to retrieve the unique identifier assigned to each process by the kernel.

2.1.3. Exponential Growth

Each additional `fork()` call effectively doubles the current number of processes. Therefore, if we call fork n times, the total number of processes generated will be 2^n.

3. Lab 2: Analyzing fork() and Process Behavior

3.1. Program 1: Basic Process Initiation

3.1.1. Goal

To observe how a single `fork()` call splits the execution path and understand the resulting behavior of the system.

3.1.2. Source Code

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

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

3.1.3. Expected Result

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

3.1.4. Technical Analysis

Because `fork()` is called once, one child process is spawned. The initial `printf` occurs before the split, so it only appears once. However, because both the parent and child continue execution from the point of the fork, the ”Hello world” message is printed by both.

3.2. Part A: Sequential fork() Invocations

3.2.1. Goal

To investigate how two consecutive `fork()` calls impact the total process count.

3.2.2. Modified Program

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

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

3.2.3. Exercises

Question 1: Output Count

Answer:

How many times does the terminal display “Hello world”?

Answer: 4 times.

With two calls, the total count reaches four. The first call splits the parent and a child. Subsequently, both of those processes hit the second fork call, creating two more processes. Consequently, four distinct processes reach the final print statement.

Question 2: Process Tree Visualization

Answer:

Illustrate the process hierarchy.

Answer:

```
After First fork():
        [Parent]
          |
        [Child 1]


After Second fork():
        [Parent] ----> [Child 2 (from Parent)]
          |
        [Child 1] ----> [Child 3 (from Child 1)]


Result: 4 total active processes.
```

Question 3: Logic Summary

Answer:

Provide a brief explanation of the growth.

Answer: The process count doubles with every fork. Starting with 1, the first fork yields 2 processes, and the second fork causes both to replicate, leading to $2 + 2 = 4$. This follows the 2^n rule where n is the number of calls.

3.3. Part B: Triple fork() Growth

3.3.1. Goal

To further validate the exponential growth of processes and map out a more complex process tree.

3.3.2. Modified Program

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

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

3.3.3. Exercises

Question 4: Output Count

Answer:

How many instances of “Hello world” will be seen?

Answer: 8 times.

Based on the formula 2^3, three sequential fork calls result in 8 individual processes, each executing the print command.

Question 5: Process Tree Visualization

Answer:

Illustrate the process hierarchy.

Answer:

```
Total outcome: 8 processes

Level 0: Main Process
Level 1: (Fork 1) -> 2 processes
Level 2: (Fork 2) -> 4 processes
Level 3: (Fork 3) -> 8 processes

Every existing process at each level
replicates itself during the next fork call.
```

Question 6: Analysis

Answer:

Review and explain the output.

Answer: The behavior remains consistent with exponential expansion. By the time the third fork is executed, there are already 4 processes running; each of these creates a new child, bringing the total to 8. This confirms that the number of processes equals 2 raised to the power of the number of `fork()` calls.

3.4. Program 2: Differentiating Between Parent and Child

3.4.1. Goal

To implement logic that allows a program to identify if it is the parent or the child by checking the `fork()` return value and PIDs.

3.4.2. Source Code

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

int main() {
    int pid;
    printf("Initial Parent PID: %d\n", getpid());
    printf("Executing before fork...\n");
    pid = fork();
    printf("Executing after fork...\n");

    if (pid == 0)
        printf("Child process active. My PID is: %d\n", getpid());
    else
        printf("Parent process active. My PID is: %d\n", getpid())
            ;

    return 0;
}
```

3.4.3. Execution Steps

```
$ gcc -o fork_check fork_check.c
$ ./fork_check
```

3.4.4. Example Output

```
Initial Parent PID: 4050
```

```
Executing before fork...
Executing after fork...
Parent process active. My PID is: 4050
Executing after fork...
Child process active. My PID is: 4051
```

3.4.5. Analysis

This exercise highlights how the return value of `fork()` serves as a toggle for process-specific logic. Because the return value differs depending on the context:

1. **Parent context**: The function returns the child's actual PID.
2. **Child context**: The function returns a constant 0.
3. **Failure**: A value of -1 is returned if the system cannot create the process.

By capturing this value in a variable, we can use conditional statements to assign different tasks to the parent and the child.

3.5. Detailed Questions

Question 7: Return Value Significance

Answer:

What is the functional difference between `pid == 0` and `pid > 0`?

Answer: The return value acts as a flag. If `pid` is 0, the CPU is currently executing the child's copy of the code. If `pid` is a positive integer, it is the parent's copy, and that integer is the ID of the child it just created.

Question 8: PID Discrepancy

Answer:

Why are the PIDs printed by the parent and child different?

Answer: This occurs because the OS kernel must ensure every active process has a unique identifier for tracking purposes. Even though the child is a clone, it is a distinct entity in the system's process table with its own unique PID.

Question 9: Execution Order

Answer:

Why does the parent often print first? Is this order permanent?

Answer: Code before the fork is strictly parent-only. Once the fork occurs, both processes compete for CPU time. The order is determined by the OS kernel's scheduler. While the parent might often appear first, it is not a rule; the order is ”non-deterministic” and can change across different runs.

4. Conclusion

Through these practical lab tasks, I have explored the fundamental behavior of the `fork()` system call. The primary takeaways include:

1. `fork()` creates a duplicate of the calling process.
2. While they share code, parent and child processes are assigned unique PIDs.
3. Sequential forks result in 2^n processes, demonstrating exponential growth.
4. Conditional logic based on return values allows for process-specific execution.
5. The timing of output is subject to system scheduling and is not fixed.

These concepts are foundational for mastering systems programming and understanding the process lifecycle in Unix-based operating systems.