CPE 345: Operating Systems Laboratory

Lab02

Processes

Submitted by: Gianna Foti

Date of Experiment:______15 January 2025_______________

Report Deadline:_______22 January 2025________________

Lab Section: ______CPE345-03______

Demonstration Deadline: _____22 January 2025___________

Introduction

In this lab, we focused on understanding Linux process creation and management using system calls such as fork(), exit(), and wait(). Through a series of 4 programming exercises, we explored the behavior of parent and child processes, memory independence, and process synchronization. We implemented programs to observe how processes interact, manage variables independently, and execute hierarchical tasks involving arithmetic operations across multiple processes. Additionally, we created programs to spawn multiple child processes and validated input constraints to ensure proper execution. The lab also covered special process states, including orphan, zombie, and sleeping beauty processes. By analyzing program outputs and addressing related theoretical questions, we gained a deeper understanding of process control mechanisms and their practical applications in operating systems.

Theory Topics

1. Process: A process is an executing instance of a program that performs a specific task. It consists of program code, data, and system resources such as memory and file handles. Each process operates independently, with its own address space and control flow. Processes are managed by the operating system, which schedules their execution and allocates resources to ensure efficient multitasking.
    a. New: The process is being created but is not yet ready to execute.
    b. Ready: The process is prepared to run and is waiting to be assigned to the CPU.
    c. Running: The process is currently being executed by the CPU.
    d. Waiting: The process is waiting for an event to complete before it can resume execution.
    e. Terminated: The process has completed execution or has been terminated and is waiting to be removed from the system.
2. Orphan Process: An orphan process is a child process whose parent has terminated before it completes execution. Once the parent process exits, the child process continues running and is

automatically adopted by the init process, which becomes its new parent. This ensures that the orphan process can continue execution and eventually terminate properly without becoming a zombie. An orphan process demonstrates how the operating system handles processes whose parent is no longer available.

3. Zombie Process: A zombie process is a process that has completed execution but still has an entry in the process table because its exit status has not been collected by the parent. This occurs when the parent process does not call wait() to read the child's termination status, leaving the child process in an inactive state. Zombie processes consume minimal system resources but can clutter the process table if not properly managed by the parent process.
4. fork():
    a. The fork() system call in Linux and UNIX is used to create a new process by duplicating the calling process. This results in two processes: the parent process (original) and the child process (newly created). The parent process continues execution while the child process starts execution from the point of the fork() call. The return value of fork() helps distinguish between the parent and child processes; the child process receives a return value of 0, while the parent receives the PID of the child process. If fork() fails, it returns -1, indicating an error in process creation. The newly created child process has an identical memory image as the parent but operates independently.
5. exit():
    a. The exit() function is used by a process to terminate its execution and return an exit status to the operating system. The parent process can retrieve this status using the wait() function. When exit() is called, resources allocated to the process are freed. Additionally, the process enters the terminated state. It is commonly used by child processes to indicate their termination status to their parent.
6. wait():

   a. The wait() function is used by a parent process to pause execution until one of its child processes terminates. It collects the exit status of the terminated child and returns its process ID. This function helps prevent zombie processes by ensuring that the parent's acknowledgment of the child's termination removes it from the process table. When wait() is called, the parent process enters a blocked state until a child finishes execution, at which point the function returns the child's PID and stores its exit status in a specified variable. If a parent has multiple child processes, wait() will return when any one of them terminates.

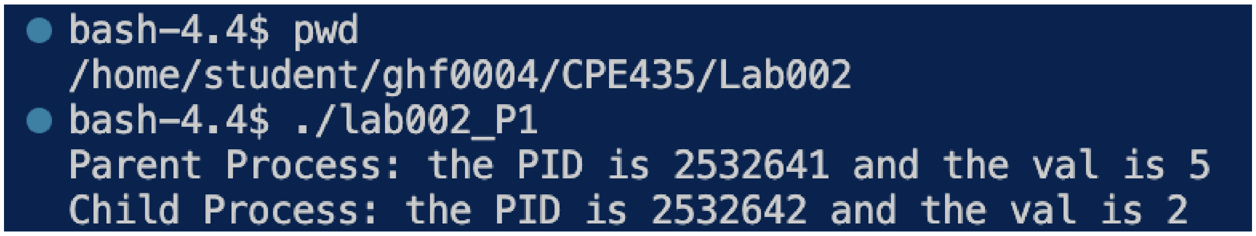
Part 1:

Program Description: This program uses the fork() system call to create a child process. The child process adds 2 to a variable and prints its PID and value. The parent process adds 5 to the variable and prints its PID and value.

Process:

1. First, I initialized val to 0.
   a. This sets the starting value for both the parent and child processes.
2. Then, I called the fork() system call.
   a. This creates a new child process that is an exact copy of the parent process.
3. In the child process (pid == 0), I added 2 to val.
   a. The child process prints its own PID and the updated value of val.
4. In the parent process (pid > 0), I added 5 to val.
   a. The parent process prints its own PID and the updated value of val.
5. Finally, both processes run independently with their own copies of val.
   a. Changes to val in one process do not affect the other, demonstrating memory isolation.

Program Output:



```
● bash-4.4$ pwd
  /home/student/ghf0004/CPE435/Lab002
● bash-4.4$ ./lab002_P1
  Parent Process: the PID is 2532641 and the val is 5
  Child Process: the PID is 2532642 and the val is 2
```

Figure 1: Program 1 Output

Part 1 Questions:

1. What can you conclude about the values of the variable val?
    a. After the fork() call, both the parent and child processes have separate and independent copies of the variable val. Changes made to val in one process do not affect the other because each process has its own memory space.

Part 2:

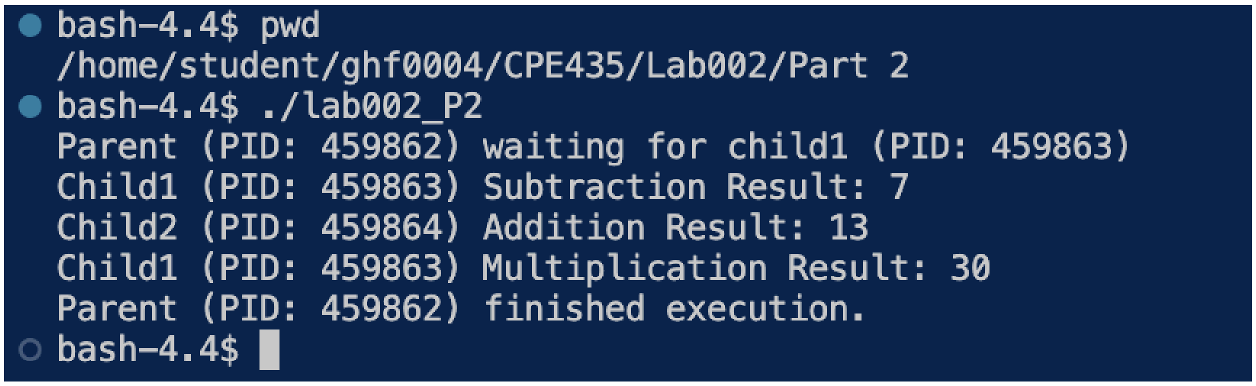
Program Description: This program creates a hierarchical process structure using the fork() system call. The parent process creates child1 and waits for it to finish. Child1 performs a subtraction, prints the result with its PID, then creates child2 to perform an addition and print its result. After child2 terminates, child1 performs a multiplication, prints the result, and exits. The parent process then resumes and terminates. The arguments that I use are 10 and 3.

Process:

1. First, I created a child process (child1) using the fork() system call.
    a. This created a duplicate of the parent process, allowing both to run independently. The parent waited for child1 to finish execution.
2. In the child1 process, I performed a subtraction operation and printed the result along with its process ID.
    a. Then, child1 created another child process (child2) using fork().

3. In the child2 process, I performed an addition operation and printed the result along with its process ID.
    a. Meanwhile, child1 waited for child2 to complete execution before proceeding.
4. After child2 completed, child1 performed a multiplication operation and printed the result along with its process ID.
    a. Once done, child1 terminated, allowing the parent process to resume.
5. Finally, after child1 completed, the parent process printed a completion message and terminated.
    a. This demonstrated the use of fork() for process creation and wait() for synchronization.

Program Output:



```
● bash-4.4$ pwd
  /home/student/ghf0004/CPE435/Lab002/Part 2
● bash-4.4$ ./lab002_P2
  Parent (PID: 459862) waiting for child1 (PID: 459863)
  Child1 (PID: 459863) Subtraction Result: 7
  Child2 (PID: 459864) Addition Result: 13
  Child1 (PID: 459863) Multiplication Result: 30
  Parent (PID: 459862) finished execution.
○ bash-4.4$
```

Figure 2: Program 2 Output

Part 3:

Program Description:

This program prompts the user to input an even number of child processes to create. If the input number is odd, the program displays an error message and terminates. The parent process creates n child processes using the fork() system call, and each child prints its process ID along with the parent's ID before terminating. The parent process waits for all child processes to complete before exiting.

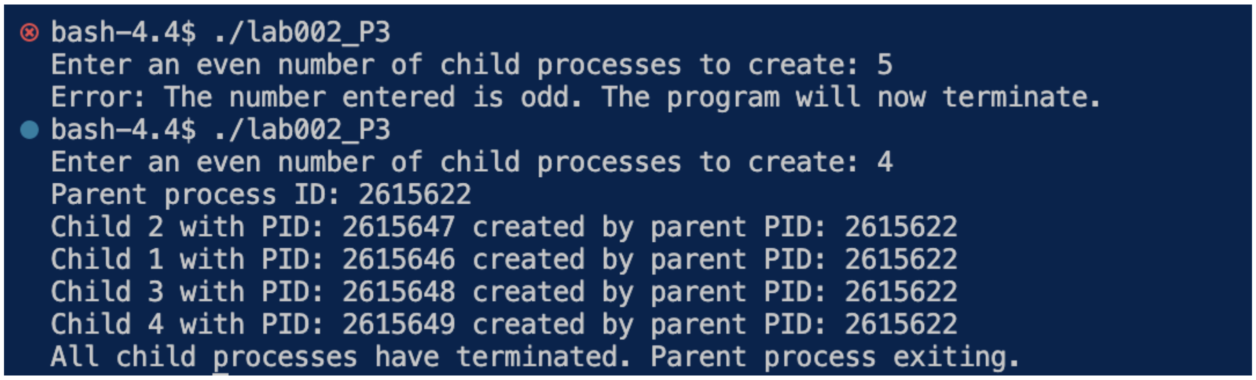
Process:

1. First, I prompted the user to enter an even number of child processes to create.

    a. If the user entered an odd number, the program displayed an error message and terminated immediately.

2. If a valid even number was entered, I created `n` child processes using a loop and the fork() system call.
    a. Each child process printed its process ID (PID) and its parent's PID before terminating.
3. In the parent process, I used the wait() system call to wait for each child process to complete.
    a. This ensured that the parent process did not terminate before all child processes had finished execution.
4. After all child processes had terminated, the parent process printed a message indicating successful completion.
    a. This marked the end of the program execution.
5. Finally, the program exited gracefully, demonstrating proper process creation, termination, and synchronization using fork() and wait() system calls.

Program Output:



```
⊗ bash-4.4$ ./lab002_P3
  Enter an even number of child processes to create: 5
  Error: The number entered is odd. The program will now terminate.
● bash-4.4$ ./lab002_P3
  Enter an even number of child processes to create: 4
  Parent process ID: 2615622
  Child 2 with PID: 2615647 created by parent PID: 2615622
  Child 1 with PID: 2615646 created by parent PID: 2615622
  Child 3 with PID: 2615648 created by parent PID: 2615622
  Child 4 with PID: 2615649 created by parent PID: 2615622
  All child processes have terminated. Parent process exiting.
```

Figure 3: Program 3 Output

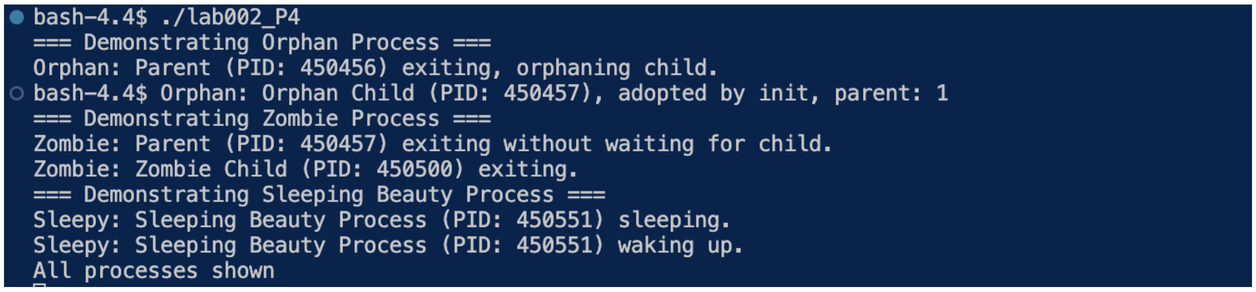
Part 4:

Program Description: This program demonstrates three process states in Linux: orphan, zombie, and sleeping beauty. An orphan process is created when the parent terminates before the child, leaving it to be adopted by the init process. A zombie process occurs when the child exits before the parent, and the parent does not collect its exit status. The sleeping beauty process simulates a child that sleeps for a period while the parent waits for it to complete. The program sequentially showcases these processes, providing insights into how Linux handles process creation and termination.

Process:

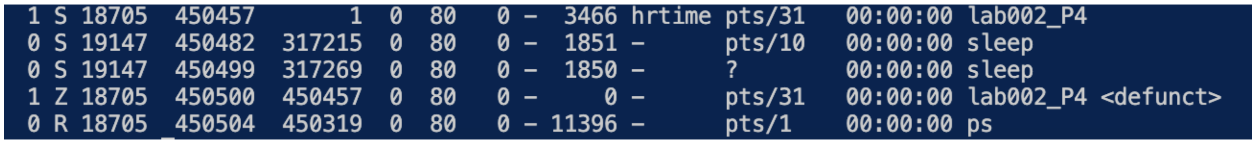
1. First, the aloneOrphan() function is called to demonstrate the orphan process concept. It creates a child process using fork().
    a. The parent process prints a message indicating it is exiting, which leaves the child process orphaned.
    b. The child then sleeps for 2 seconds and prints a message with the prefix “Orphan:” indicating it has been adopted by the init process.
2. Next, I called the createZomboss() function to demonstrate the zombie process concept.
    a. The child process prints a message with the prefix “Zombie:” and exits immediately, becoming a zombie process. Meanwhile, the parent process prints a message with the same prefix and sleeps for 1 second without calling wait(), leaving the terminated child as a zombie process in the process table until the parent exits.
3. Then, I called the sleepyBeauty() function to demonstrate the sleeping beauty process concept.
    a. The child process prints a message with the prefix “Sleepy:” indicating it is going to sleep for 3 seconds. After waking up, it prints another message and exits. The parent waits for the child to complete execution using wait(), ensuring synchronization.
4. Finally, after all processes are demonstrated, the main() function prints a completion message and terminates.

Program Output:



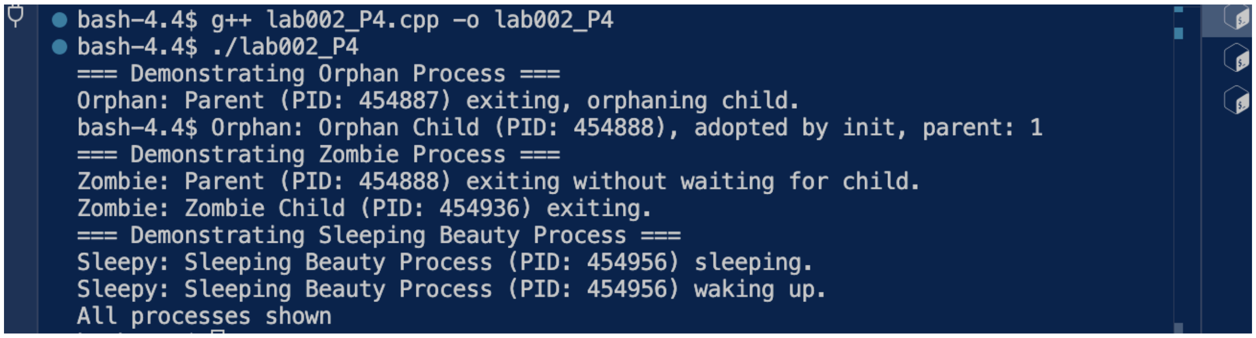
```
bash-4.4$ ./lab002_P4
=== Demonstrating Orphan Process ===
Orphan: Parent (PID: 450456) exiting, orphaning child.
bash-4.4$ Orphan: Orphan Child (PID: 450457), adopted by init, parent: 1
=== Demonstrating Zombie Process ===
Zombie: Parent (PID: 450457) exiting without waiting for child.
Zombie: Zombie Child (PID: 450500) exiting.
=== Demonstrating Sleeping Beauty Process ===
Sleepy: Sleeping Beauty Process (PID: 450551) sleeping.
Sleepy: Sleeping Beauty Process (PID: 450551) waking up.
All processes shown
```

Figure 4.1: Program 4 Orphan and Zombie Process Output



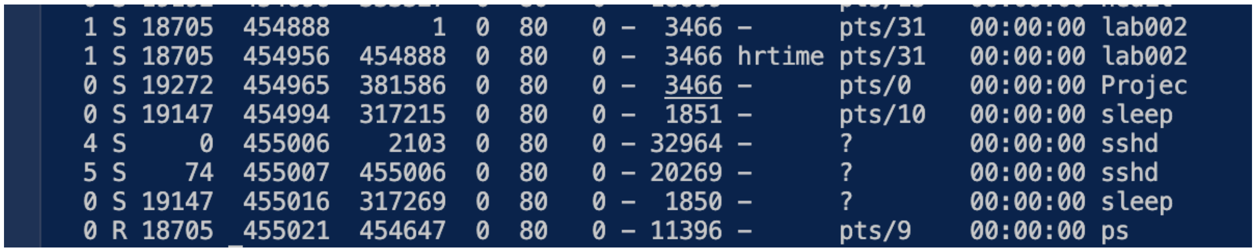
```
1 S 18705  450457       1  0  80   0 -  3466 hrtime pts/31   00:00:00 lab002_P4
0 S 19147  450482  317215  0  80   0 -  1851 -      pts/10   00:00:00 sleep
0 S 19147  450499  317269  0  80   0 -  1850 -      ?        00:00:00 sleep
1 Z 18705  450500  450457  0  80   0 -     0 -      pts/31   00:00:00 lab002_P4 <defunct>
0 R 18705  450504  450319  0  80   0 - 11396 -      pts/1    00:00:00 ps
```

Figure 4.2: Program 4 Orphan and Zombie Process Table



```
bash-4.4$ g++ lab002_P4.cpp -o lab002_P4
bash-4.4$ ./lab002_P4
=== Demonstrating Orphan Process ===
Orphan: Parent (PID: 454887) exiting, orphaning child.
bash-4.4$ Orphan: Orphan Child (PID: 454888), adopted by init, parent: 1
=== Demonstrating Zombie Process ===
Zombie: Parent (PID: 454888) exiting without waiting for child.
Zombie: Zombie Child (PID: 454936) exiting.
=== Demonstrating Sleeping Beauty Process ===
Sleepy: Sleeping Beauty Process (PID: 454956) sleeping.
Sleepy: Sleeping Beauty Process (PID: 454956) waking up.
All processes shown
```

Figure 4.3: Program 4 Sleeping Beauty Process Output



```
1 S 18705  454888       1  0  80   0 -  3466 -      pts/31   00:00:00 lab002
1 S 18705  454956  454888  0  80   0 -  3466 hrtime pts/31   00:00:00 lab002
0 S 19272  454965  381586  0  80   0 -  3466 -      pts/0    00:00:00 Projec
0 S 19147  454994  317215  0  80   0 -  1851 -      pts/10   00:00:00 sleep
4 S     0  455006    2103  0  80   0 - 32964 -      ?        00:00:00 sshd
5 S    74  455007  455006  0  80   0 - 20269 -      ?        00:00:00 sshd
0 S 19147  455016  317269  0  80   0 -  1850 -      ?        00:00:00 sleep
0 R 18705  455021  454647  0  80   0 - 11396 -      pts/9    00:00:00 ps
```

Figure 4.4: Program 4 Sleeping Beauty Process Table

Conclusion

In conclusion, this lab provided hands-on experience with process management in Linux through four tasks. First, we created a program to demonstrate process forking and memory isolation, showing that parent and child processes maintain separate memory spaces. Next, we implemented a hierarchical process structure, where a parent created a child, which then created another child. This demonstrated process synchronization using wait() to ensure orderly execution. Then, we developed a program to create an even number of child processes based on user input, focusing on dynamic process creation and error handling. Finally, we demonstrated orphan, zombie, and sleeping beauty processes, highlighting process adoption by init, lingering zombie processes, and parent-child synchronization. I did not have many issues with this lab. Overall, the lab reinforced key concepts of process creation, termination, and synchronization in Linux.

Appendix

Table 1: Program 1

```
// Author: Gianna Foti
// Teacher: Trevor McClain
// Course: CPE435-03
// Filename: lab002_P1.cpp
// Due Date: 22 January 2025
// Description: This program uses the fork() system call to create a child process.
//              The child process adds 2 to a variable and prints its PID and value.
//              The parent process adds 5 to the variable and prints its PID and value.


using namespace std;
#include <iostream>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
   // declaring and initialzing variable to 0
   int val = 0;
```

```
    // calling da fork system call
    pid_t pid = fork();

    // child processsssss
    if (pid == 0) {
        // adding 2 to the value of val
        val += 2;
        cout << "Child Process: the PID is " << getpid() << " and the val is " << val << endl;
    }
    // parent process
    else if (pid > 0) {
        // adding 5 to the value of val
        val += 5;
        cout << "Parent Process: the PID is " << getpid() << " and the val is " << val << endl;
    }
    // my just in case error handling darlin
    else {
        cout << "Fork failed! Something wrong brother" << endl;
        return 1;
    }
    return 0;
}
```

Table 2: Program 2

```
// Author: Gianna Foti
// Teacher: Trevor McClain
// Course: CPE435-03
// Due Date: 22 January 2025
// Description: This program demonstrates process creation and synchronization using fork().
//              It creates two child processes to perform subtraction, addition, and multiplication.
//              I used the numbers 10 and 3
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

using namespace std;

// Function to subtract two numbers
void subtract(int a, int b) {
    cout << "Child1 (PID: " << getpid() << ") Subtraction Result: " << (a - b) << endl;
}
```

```
// Function to add two numbers
void add(int a, int b) {
    cout << "Child2 (PID: " << getpid() << ") Addition Result: " << (a + b) << endl;
}

// Function to multiply two numbers
void multiply(int a, int b) {
    cout << "Child1 (PID: " << getpid() << ") Multiplication Result: " << (a * b) << endl;
}

int main() {
    pid_t child1_pid, child2_pid;

    // Create child1 process
    child1_pid = fork();

    if (child1_pid < 0) {
        cerr << "Fork failed for child1" << endl;
        return 1;
    }

    if (child1_pid == 0) {  // Child1 process
        subtract(10, 3);  // Perform subtraction

        // Create child2 process
        child2_pid = fork();

        if (child2_pid < 0) {
            cerr << "Fork failed for child2" << endl;
            return 1;
        }

        if (child2_pid == 0) {  // Child2 process
            add(10, 3);  // Perform addition
            return 0;   // Exit child2
        }

        wait(NULL);  // Wait for child2 to finish
        multiply(10, 3);  // Perform multiplication
        return 0;  // Exit child1
    }
    // Parent process
    cout << "Parent (PID: " << getpid() << ") waiting for child1 (PID: " << child1_pid << ")" << endl;
    wait(NULL);  // Wait for child1 to finish

    cout << "Parent (PID: " << getpid() << ") finished execution." << endl;
    return 0;
}
```

Table 3: Program 3

```
// Author: Gianna Foti
// Teacher: Trevor McClain
// Course: CPE435-03
// Due Date: 22 January 2025
// Description: This program creates 'n' child processes using the fork() system call, where 'n'
//              is an even number provided by the user. If an odd number is entered, the program
//              displays an error message and terminates. The parent process creates all child
//              processes, waits for their completion, and then exits.

#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <cstdlib>

using namespace std;

int main() {
    int n;

    // Prompt the user for input
    cout << "Enter an even number of child processes to create: ";
    cin >> n;

    // Check if n is even; if not, terminate the program
    if (n % 2 != 0) {
        cout << "Error: The number entered is odd. The program will now terminate." << endl;
        return 1;  // Exit with error code
    }

    cout << "Parent process ID: " << getpid() << endl;

    // Loop to create n child processes
    for (int i = 0; i < n; i++) {
        pid_t pid = fork();

        if (pid < 0) {
            cerr << "Fork failed!" << endl;
            exit(1);
        } else if (pid == 0) {
            // Child process
            cout << "Child " << i + 1 << " with PID: " << getpid() << " created by parent PID: " <<
getppid() << endl;
            exit(0);  // Ensure child terminates after printing
        }
    }

    // Parent process waits for all child processes to finish
    for (int i = 0; i < n; i++) {
        wait(NULL);
    }

    cout << "All child processes have terminated. Parent process exiting." << endl;
    return 0;
}
```

Table 4: Program 4

```
// Author: Gianna Foti
// Teacher: Trevor McClain
// Course: CPE435-03
// Due Date: 22 January 2025
// Description: This program demonstrates orphan, zombie, and sleeping beauty process states in Linux.

#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

using namespace std;

// shows an orphan process by terminating the parent and leaving the child to be adopted by init
void aloneOrphan() {
    pid_t pid = fork();

    if (pid == 0) {
        sleep(2);
        cout << "Orphan: Orphan Child (PID: " << getpid() << "), adopted by init, parent: " << getppid()
<< endl;
    }
    else {
        cout << "Orphan: Parent (PID: " << getpid() << ") exiting, orphaning child." << endl;
        exit(0);
    }
}

// shows a zombie process by allowing the child to exit before the parent calls wait()
void createZomboss() {
    pid_t pid = fork();

    if (pid == 0) {
        cout << "Zombie: Zombie Child (PID: " << getpid() << ") exiting." << endl;
        exit(0);
    }
    else {
        cout << "Zombie: Parent (PID: " << getpid() << ") exiting without waiting for child." << endl;
        sleep(1);
       wait(NULL);
    }
}

// creates a child that sleeps for a set time while the parent waits
void sleepyBeauty() {
    pid_t pid = fork();

    if (pid == 0) {
        cout << "Sleepy: Sleeping Beauty Process (PID: " << getpid() << ") sleeping." << endl;
        sleep(3);
        cout << "Sleepy: Sleeping Beauty Process (PID: " << getpid() << ") waking up." << endl;
        exit(0);
    }
    else {
        wait(NULL);
    }
}

// main function with the cout statements
int main() {
    cout << "=== Demonstrating Orphan Process ===" << endl;
    aloneOrphan();
    sleep(1);

    cout << "=== Demonstrating Zombie Process ===" << endl;
    createZomboss();
    sleep(1);

    cout << "=== Demonstrating Sleeping Beauty Process ===" << endl;
    sleepyBeauty();
    sleep(5);

    cout << "All processes shown" << endl;
    return 0;
}
```