**Lab No.: 3**

**System Calls for Basic Process Management**

**1. Objective**

* To write and compile a basic C program on a Linux system.
* To understand and implement the fork() system call for creating child processes.
* To demonstrate how a child process executes concurrently with the parent process.
* To execute one program from another using the exec family of functions.

**2. Theory**

System calls are the primary interface between a user application and the Linux operating system. In process management, system calls like fork() and exec() are crucial:

* **fork()**: Used to create a new process by duplicating the current (parent) process. The new process is called the **child** process. Both continue execution from the point of fork(), but with different process IDs.
* **exec() family**: A set of functions (execl, execp, execv, etc.) used to replace the current process image with a new program. It's commonly used after fork() when a child process is required to run a different program.

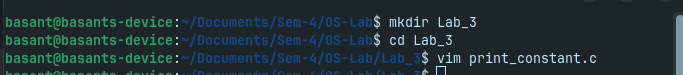
These system calls form the backbone of multitasking and process management in Unix-like systems.

**3. Tools and Commands Used**

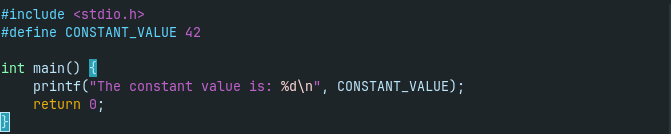
* **gcc** – GNU C Compiler for compiling C programs.
* **Terminal** – Linux command line interface.
* **C programming language** – For writing system-level programs.
* **System Calls** – fork(), exec(), getpid(), getppid().

**4. Procedure**

1. **Basic Program Compilation**
   * Write a simple C program that prints a constant.
   * Compile the program using gcc filename.c -o outputname.
   * Run the executable using ./outputname.
2. **Creating a Child Process using fork()**
   * Write a C program using fork().
   * Use getpid() and getppid() to display process IDs.
   * Observe the behavior of parent and child processes.
3. **Demonstrating fork() Execution**
   * Modify the program to include different outputs for the parent and child.
   * Run the program and note simultaneous execution.
4. **Executing Another Program from a Program**
   * Use fork() to create a child process.
   * In the child, use exec() to execute another program (e.g., ls, another C executable).
   * Observe the replacement of the child process image.
5. Write a basic C Program to print out a constant value. Use gcc compiler to generate a LINUX executable file.
   1. Write a C language source file.



* 1. Write the Constant value code in the file.



* 1. Compile the C file using ‘GCC’



* 1. Execute the file.

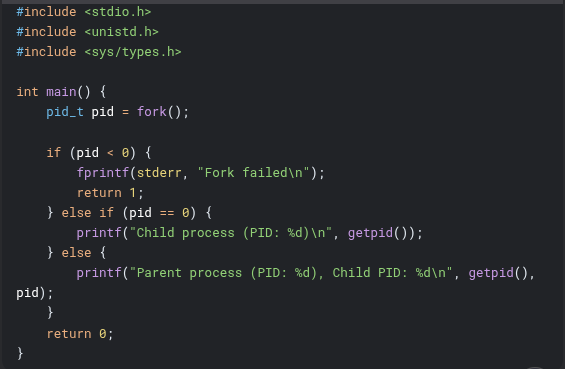


1. Creating Child process from a parent process using function fork ().

1. Write a C language Source File.



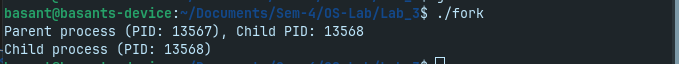
1. Write the Fork function Code in the file.



1. Compile the C File using ‘GCC’.



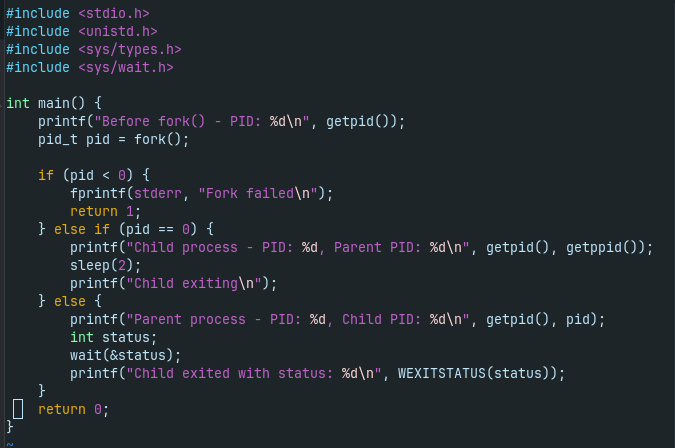
1. Execute the File.



1. Demonstration of fork () function which is used to create a child process from parent process.
2. Write the C Language Source File.



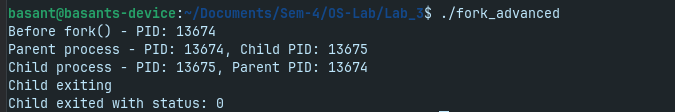
1. Write the Child-Parent Fork Function Code.



1. Compile the C File using ‘GCC’.



1. Execute the File.

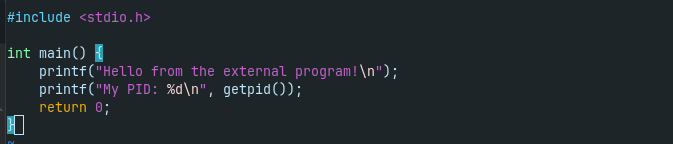


1. Execution of program from another program

1. Write a C Language Source File2.



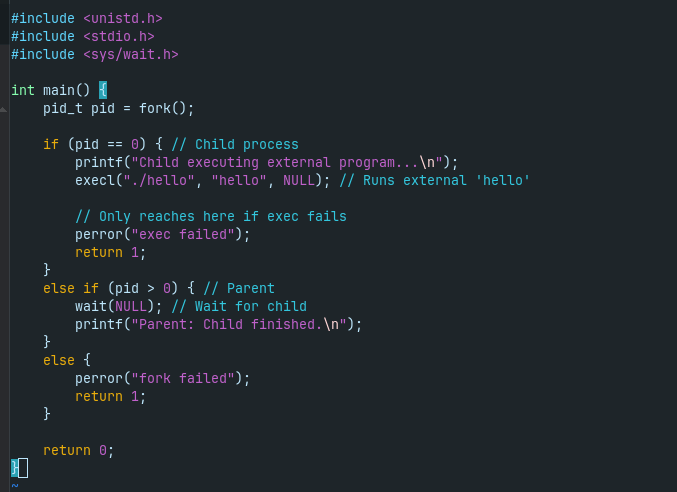
1. Write an executable C code for File2.



1. Write another C Language Source File1.



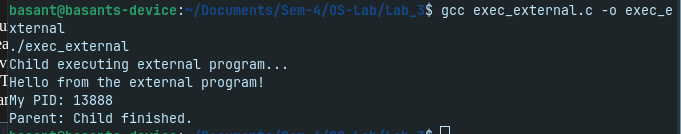
1. Write an executable C code for File1 with exec and fork function for File2.



1. Compile both the C File1 and File2 using ‘GCC’.



1. Execute the File2.



**5. Conclusion**

This lab provided hands-on experience with fundamental system calls for process management in Linux. We learned how to create child processes using fork(), understood how parent and child processes differ in behavior and execution, and practiced replacing a process with another using exec() functions. These concepts are critical for understanding multitasking, process control, and system-level programming in Unix-like environments.