Lab 5: Fork System Call

A system call is a way for a user program to interface with the operating system. The program requests several services, and the OS responds by invoking a series of system calls to satisfy the request. A system call can be written in assembly language or a high-level language like C or Pascal. System calls are predefined functions that the operating system may directly invoke if a high-level language is used.

The Application Program Interface (API) connects the operating system's functions to user programs. It acts as a link between the operating system and a process, allowing user-level programs to request operating system services. The kernel system can only be accessed using system calls. System calls are required for any programs that use resources. One of the most frequently used system call is the fork system call. The fork system call is used to create a new process. The newly created process is the child process. The process which calls fork and creates a new process is the parent process. The child and parent processes are executed concurrently.

But the child and parent processes reside on different memory spaces. These memory spaces have same content and whatever operation is performed by one process will not affect the other process. Once the child processes are created, now both the processes will have the same Program Counter (PC), so both processes will point to the same next instruction. The files opened by the parent process will be the same for child process.

Difference between Process Id's of Child & Parent process

1. The process ID of the child process is a unique process ID which is different from the IDs of all other existing processes.
2. The Parent process ID will be the same as that of the process ID of child's parent.

Although the child process is created from the parent process, they differ from each other in several ways. Some of the key properties of the parent and child processes are listed below:

Properties of child Process:

1. The CPU counters and the resource utilizations are initialized to reset to zero.
2. When the parent process is terminated, child processes do not receive any signal.
3. The thread used to call fork() creates the child process. So, the address of the child process will be the same as that of parent.
4. The file descriptor of parent process is inherited by the child process. For example, the offset of the file or status of flags and the I/O attributes will be shared among the file descriptors of child and parent processes. So, file descriptor of parent class will refer to same file descriptor of child class.
5. The open message queue descriptors of parent process are inherited by the child process. For example, if a file descriptor contains a message in parent process the same message will be present in the corresponding file descriptor of child process. So, we can say that the flag values of these file descriptors are same.
6. Similarly open directory streams will be inherited by the child processes.
7. The default Timer slack value of the child class is same as the current timer slack value of parent class.

Properties which are not inherited by child process:

1. Memory locks
2. The pending signal of a child class is empty.
3. Process associated record locks.
4. Asynchronous I/O operations and I/O contents.
5. Directory change notifications.
6. Timers such as alarm(), setTimer() are not inherited by the child class

Fork() call:

The fork() system call is used to create a new process by duplicating the calling process. The fork() system call is made by the parent process, and if it is successful, a child process is created. It does not accept any parameters and it returns an integer value. After the creation of a new child process, both processes then execute the next command following the fork system call. Therefore, it is a must to separate the parent process from the child by checking the returned value of the fork():

- **Negative:** A child process could not be successfully created if the fork() returns a negative value.
- **Zero:** A new child process is successfully created if the fork() returns a zero
- **Positive:** The positive value is the process ID of a child's process to the parent. The process ID is the type of **pid_t**.

Syntax:

pid_t fork(void);

Implementation

However, it can also be used in programming languages such as C. Consider the following basic implementation to understand the syntax and working of fork() call.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
        fork();
        printf("Hello World \n");
        return 0;
}
```

```
harsh@Ubuntu:~/Desktop/OS Lab Course/lab5$ gcc fork.c -o fork1
harsh@Ubuntu:~/Desktop/OS Lab Course/lab5$ ./fork1
Hello World
Hello World
```

In the above program, the fork() function is used, which will create a new child process. Once the child process has been created, the child process and the parent process will both point to the next command. In this manner, the remaining commands will be executed 2^n times, where n represents the number of fork() system calls. Consider another implementation below:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
        fork();
        fork();
        fork();
        printf("Hello World \n");
        return 0;
}
```

```
harsh@Ubuntu:~/Desktop/OS Lab Course/lab5$ gcc fork.c -o fork1
harsh@Ubuntu:~/Desktop/OS Lab Course/lab5$ ./fork1
Hello World
harsh@Ubuntu:~/Desktop/OS Lab Course/lab5$ Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```