Experiment No. 4
Create a child process in Linux using the fork system call.
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: Create a child process in Linux using the fork systemcall.

Objective:

Create a child process using fork system call.

Theory:

A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. This may include hardware-related

From the child process obtain the process ID of both child and parent by using getpid and getppid system calls. services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a **new** process, which becomes the child process of the caller.

- If **fork()** returns a negative value, the creation of a child process was unsuccessful. ● **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the **process ID** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

If the call to **fork()** is executed successfully, Unix will make two identical copies of address spaces, one for the parent and the other for the child.**getpid, getppid - get process identification**

- **getpid()** returns the process ID (PID) of the calling process. This is often used by routines that generate unique temporary filenames.
- **getppid()** returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork().

Code:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
   pid_t pid;

   // Create a child process
   pid = fork();

   if (pid < 0) {
      // Fork failed
      fprintf(stderr, "Fork failed\n");
      return 1;
   } else if (pid == 0) {
```

```
    // Child process
    printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());

    // Simulate some work in the child process
    sleep(2);

    printf("Child process exiting\n");
  } else {
    // Parent process
    printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);

    // Wait for the child process to terminate using wait
    int status;
    wait(&status);
    printf("Child process terminated with status: %d\n", status);
  }
}
```

Output:

```
Parent process: PID = 30781, Child PID = 30782
Child process: PID = 30782, Parent PID = 30781
Child process exiting
Child process terminated with status: 0
```

```
sysads@linuxhint $ gcc fork.c -o fork
sysads@linuxhint $ ./fork
Using fork() system call
Using fork() system call
```

This program first forks a child process. In the child process, it prints its PID and the parent's PID, simulates some work by sleeping for 2 seconds, and then exits. In the parent process, it prints its PID and the child's PID, and then waits for the child process to terminate using the wait system call. After the child process terminates, it prints the status of the child process.

Conclusion:
In conclusion, through the utilization of the fork() system call in Linux, we have successfully implemented the creation of a child process within a parent process. This method enables the concurrent execution of multiple processes, facilitating parallelism and multiprocessing in Linux-based environments. By leveraging the capabilities provided by fork(), developers can effectively manage and control the execution of tasks, leading to efficient utilization of system resources and improved performance in various computing scenarios.

What do you mean by system call?

A system call is a mechanism provided by the operating system that allows user-level processes (applications) to request services from the kernel, which is the core of the operating system. These services typically involve tasks that require privileged access or interactions with hardware. System calls provide a way for user-space programs to perform actions such as reading from or writing to files, creating new processes, allocating memory, and interacting with devices.

When a user-level program needs to perform a task that requires kernel-level privileges or resources, it initiates a system call by executing a special instruction (e.g., int 0x80 in x86 architecture). This instruction triggers a switch from user mode

to kernel mode, transferring control to a specific location in the kernel designated for handling system calls.

Once in kernel mode, the kernel executes the requested operation on behalf of the calling process. After completing the requested task, the kernel returns control to the calling process, and it resumes execution in user mode.

System calls provide an essential interface between user-space applications and the kernel, allowing programs to access the underlying resources of the operating system in a controlled and secure manner. Examples of common system calls include:

open(): Opens a file for reading, writing, or both.

read(): Reads data from a file descriptor.

write(): Writes data to a file descriptor.

fork(): Creates a new process (child) identical to the current process (parent).

exec(): Replaces the current process's memory image with a new program.

exit(): Terminates the current process.

ioctl(): Performs I/O control operations on devices.

socket(): Creates a new communication endpoint (socket) for networking.