| 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);
    }

    return 0;
}
```

**Output:**

```
E:/Testing_Lang/test.c:28: undefined reference to `wait'
collect2.exe: error: ld returned 1 exit status

Build finished with error(s).

*  The terminal process failed to launch (exit code: -1).
*  Terminal will be reused by tasks, press any key to close it.
```

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.

Code :

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

int main()
{
   fork();
   fork();
   fork();
   fork();
   printf("Using fork() system call\n");
   return 0;
}
```

**Output:**

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

Conclution :

In conclusion, the fork system call in Linux is a powerful mechanism for creating child processes within a parent process. By calling fork, the parent process creates an identical copy of itself, known as the child process. This allows for parallel execution of tasks, efficient resource management, and enhanced program functionality. Understanding how to utilize fork effectively is fundamental for developing robust and scalable Linux applications. Additionally, proper handling of error conditions and resource management is crucial to ensure the stability and reliability of the system. Through the use of fork, developers can harness the full potential of multitasking and concurrency in Linux environments.