

Experiment Number: 7

TITLE: UNIX Process Control- Introductory Program.

OBJECTIVE: Study how to create a process in UNIX using fork() system call.

THEORY:

fork():

It is a system call that creates a new process under the UNIX operating system. It takes no arguments. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

- 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.

Therefore, after the system call to fork(), a simple test can tell which process is the child. Note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

Let us take an example:

```
int main()  
{  
    printf("Before Forking");  
    fork();  
    printf("After Forking");  
    return 0;  
}
```

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.
- Both processes will start their execution at the next statement following the fork() call.

If we run this program, we might see the following on the screen:

Before Forking
After Forking
After Forking

Here printf() statement after fork() system call executed by parent as well as child process.

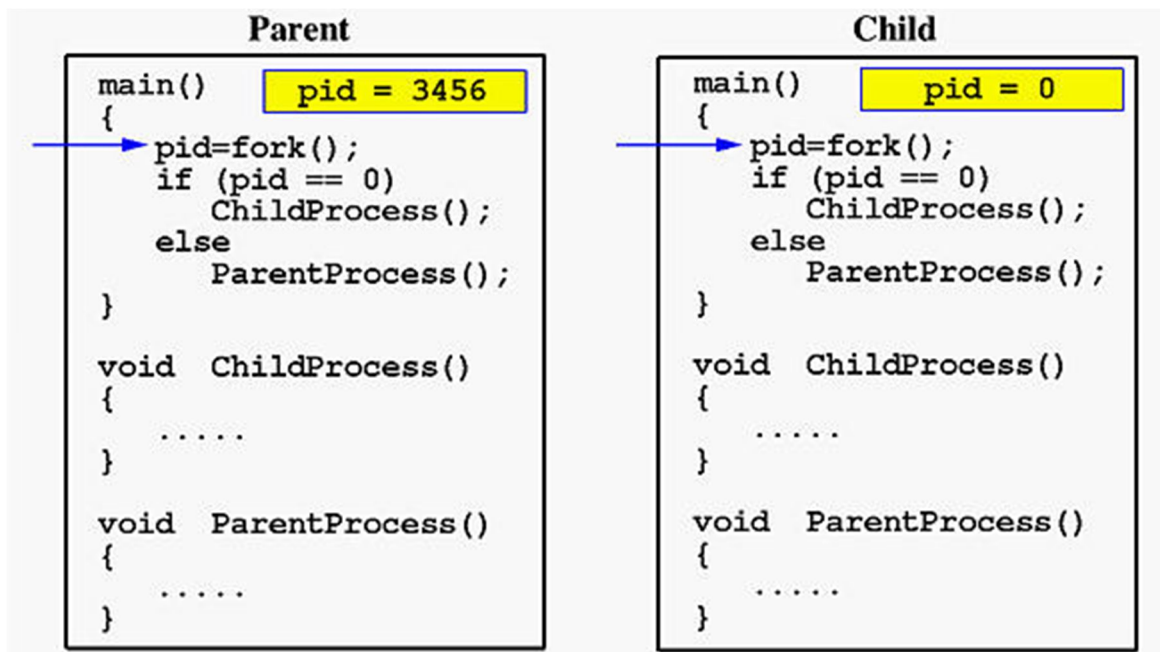
Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

Consider one simpler example, which distinguishes the parent from the child.

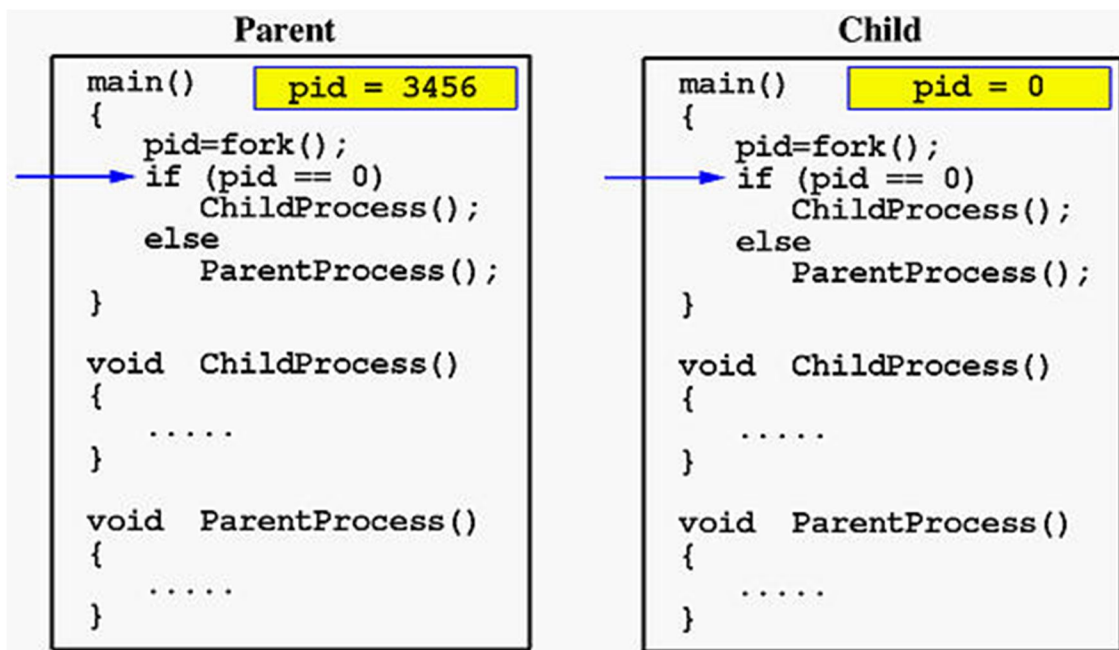
```
#include <stdio.h>
#include <sys/types.h>
void ChildProcess();          /* child process prototype */
void ParentProcess();        /* parent process prototype */
int main()
{
    pid_t  pid;
    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
    return 0;
}
void ChildProcess()
{
}
void ParentProcess()
{
}
```

In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable i.

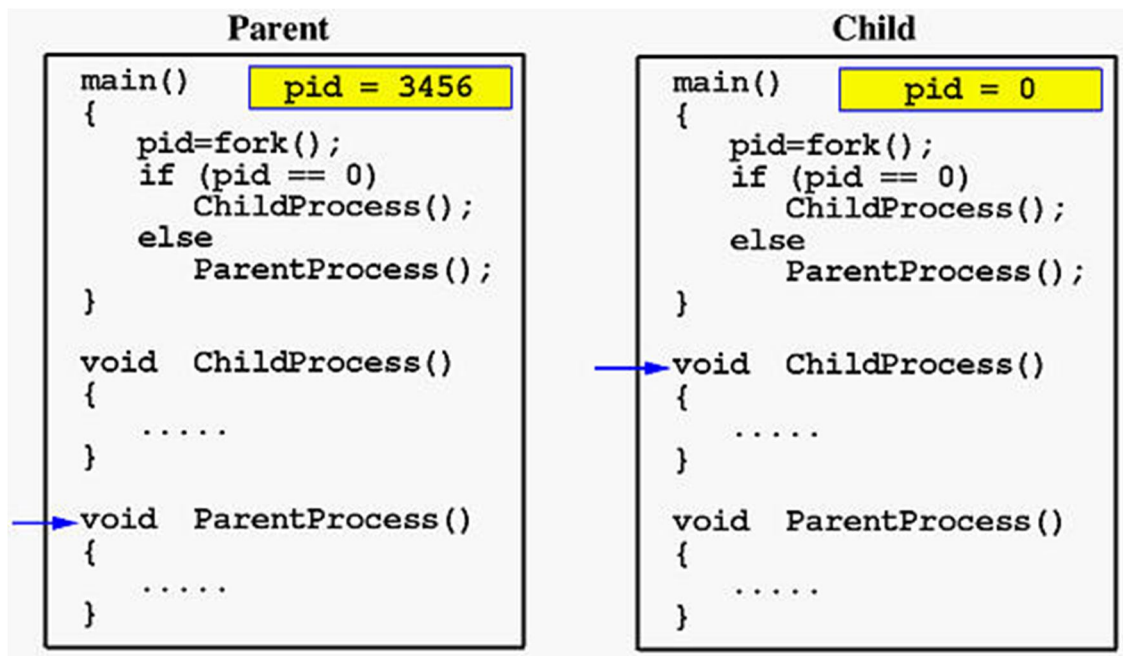
When the main program executes fork(), an identical copy of its address space, including the program and all data, is created. System call fork() returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable pid. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



Now both programs (i.e., the parent and child) will execute independent of each other starting at the next statement:



In the parent, since `pid` is non-zero, it calls function `parentprocess()`. On the other hand, the child has a zero `pid` and calls `childprocess()` as shown below:



Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process.

ps command:

The **ps** command shows the processes we're running, the process another user is running, or all the processes on the system. E.g.

```
$ ps -ef
```

By default, the **ps** program shows only processes that maintain a connection with a terminal, a console, a serial line, or a pseudo terminal. Other processes run without needing to communicate with a user on a terminal. These are typically system processes that Linux uses to manage shared resources. We can use **ps** to see all such processes using the **-e** option and to get "full" information with **-f**.

References:

1. Beginning Linux Programming by Neil Mathew and Richard Stones, Wrox Publications.
2. Unix Concepts and Applications By Sumitabha Das, Tata McGraw Hill

Tushar B Kute
(Subject Teacher)