

## **WEEK 3: Process Creation and Termination**

**Date: 11/09/2020**

### **OBJECTIVE:**

**Understanding Parent-Child Process relationship and the working of fork(), exec() and wait() system calls**

- STUDENTS ARE ADVISED TO READ THE TUTORIAL PROVIDED IN THE END OF THIS DOCUMENT BEFORE WORKING ON THE ACTUAL PROGRAMS.
- STUDENTS ARE REQUIRED TO PROVIDE PROOF OF CONDUCTION (SEE SUBMISSION BELOW) ONLY FOR THE ACTUAL PROGRAMS. SO THIS TIME THERE IS NO NEED TO SEND SCREENSHOTS FOR THE PRACTICE PROGRAMS INCLUDED IN THE TUTORIAL.
- ALSO, ANSWER 5 QUESTIONS ASKED AT THE END OF THIS DOCUMENT.

### **SUBMISSION:**

1. All the source code files for the actual programs should be uploaded to EDMODO separately in WORD or ZIP FORMAT.
2. All the screenshots clearly showing the directory name as SRN\_NAME\_WEEK3, all the output, results for the actual programs and the answers to 5 QUESTIONS should be uploaded to EDMODO in a SEPARATE FILE (Word or PDF format only, Do NOT zip this file). So, even the answers to the questions asked at the end of this document should go into the same file.

Students should keep these TWO deliverables (i.e. 1 & 2 above) separate and NOT zip all the files together in order to facilitate quick, timely and effective evaluation.

Contact your respective Lab faculty for any questions or clarifications needed.

**DUE DATE FOR SUBMISSION: 17/09/2020 11:59 PM**

## PROGRAMS FOR EXECUTION AND SUBMISSION:

1. The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Write a C program using the fork() system call that generates the Fibonacci sequence in the child process. The number of the sequence should be provided in the command line. For example, if 5 is provided as an input to the program, the first five numbers in the Fibonacci sequence should be output by the child process.
2. Write a C program that uses the child to compute partial sums and parent to compute the partial products of an array of integers. Both child and parent should print the respective total sum and product values. Use an array with a minimum of 5 elements.
3. Write a C program to demonstrate the use of fork(), exec() and wait() all in one program. Use any one of the family of exec system calls such as execl() or execvp().

Example: If the input/argument to the program is one of these

ls, ls -l, find, <executable\_program>

then your program should display the output of the same command (like the output of "ls -l" command) or the executable\_program that was passed as an argument.

## Answer the following questions and submit your answers as specified in Submission #2:

1. What is the role of the **init** process on UNIX and Linux systems in regard to process termination?
2. What is a **subreaper** process?
3. What causes a **defunct** process on the Linux system and how can you avoid it?
4. How can you identify zombie processes on the Linux system?
5. What does child process inherit from its parent?

**References:** <https://www.geeksforgeeks.org/exec-family-of-functions-in-c>

**'Man' pages of individual commands on Linux system**

# TUTORIAL

## The fork() System Call

---

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. 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. **Please note that Unix/Linux 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 to make the above points clear. This example does not distinguish parent and the child processes.

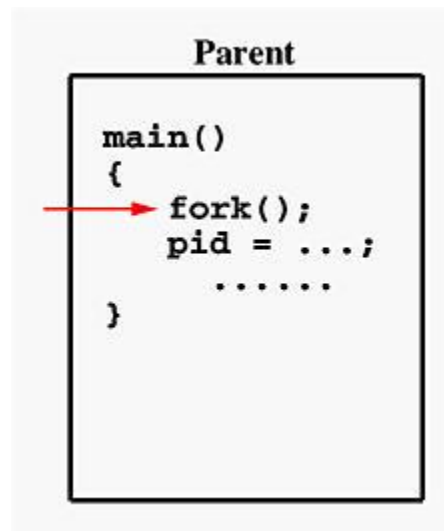
```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#define MAX_COUNT 200
#define BUF_SIZE 100

void main(void)
{
    pid_t pid;
    int i;
    char buf[BUF_SIZE];

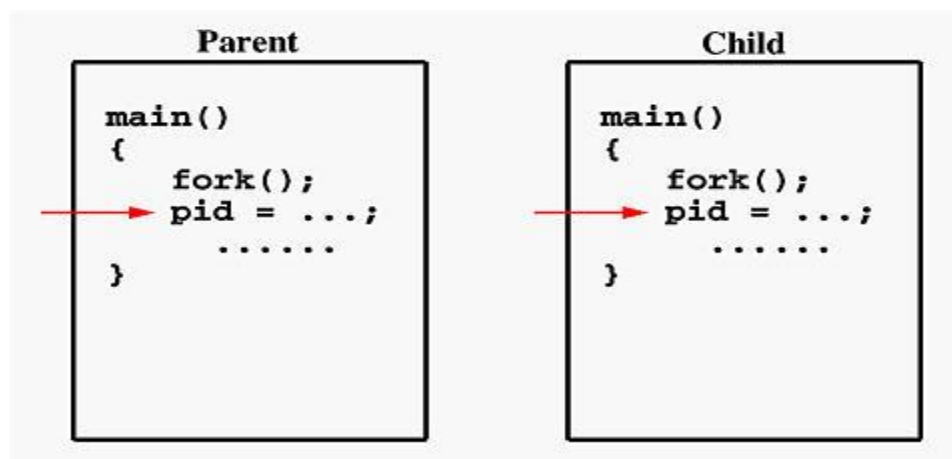
    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

Suppose the above program executes up to the point of the call to **fork()** (marked in red color):



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. In this case, both processes will start their execution at the assignment statement as shown below:



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.

What is the reason of using **write** rather than **printf**? It is because **printf()** is "buffered," meaning **printf()** will group the output of a process together. While buffering the output for the parent process, the child may also use **printf** to print out some information, which will also be buffered. As a result, since the output will not be sent to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed up in strange ways. To overcome this problem, you may consider using the "unbuffered" **write**.

If you run this program, you might see the following (but with different PIDs as per your system) on the screen:

```
.....
This line is from pid 3456, value 13
This line is from pid 3456, value 14
.....
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
.....
This line is from pid 3456, value 21
This line is from pid 3456, value 22
.....
```

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

Consider another simple example, which distinguishes the parent from the child.

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

#define MAX_COUNT 200

void ChildProcess(void);          /* child process prototype */
void ParentProcess(void);        /* parent process prototype */

void main(void)
{
    pid_t pid;

    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}
```

```

void ChildProcess(void)
{
    int i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("    This line is from child, value = %d\n", i);
    printf("    *** Child process is done ***\n");
}

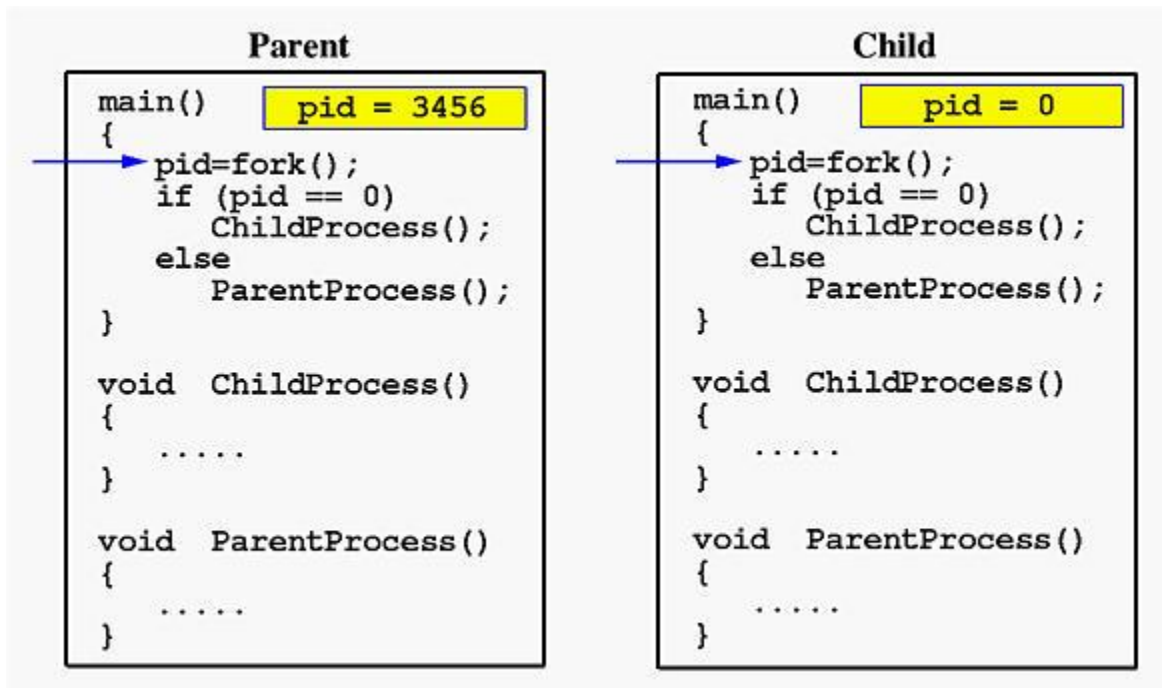
void ParentProcess(void)
{
    int i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
    printf("*** Parent is done ***\n");
}

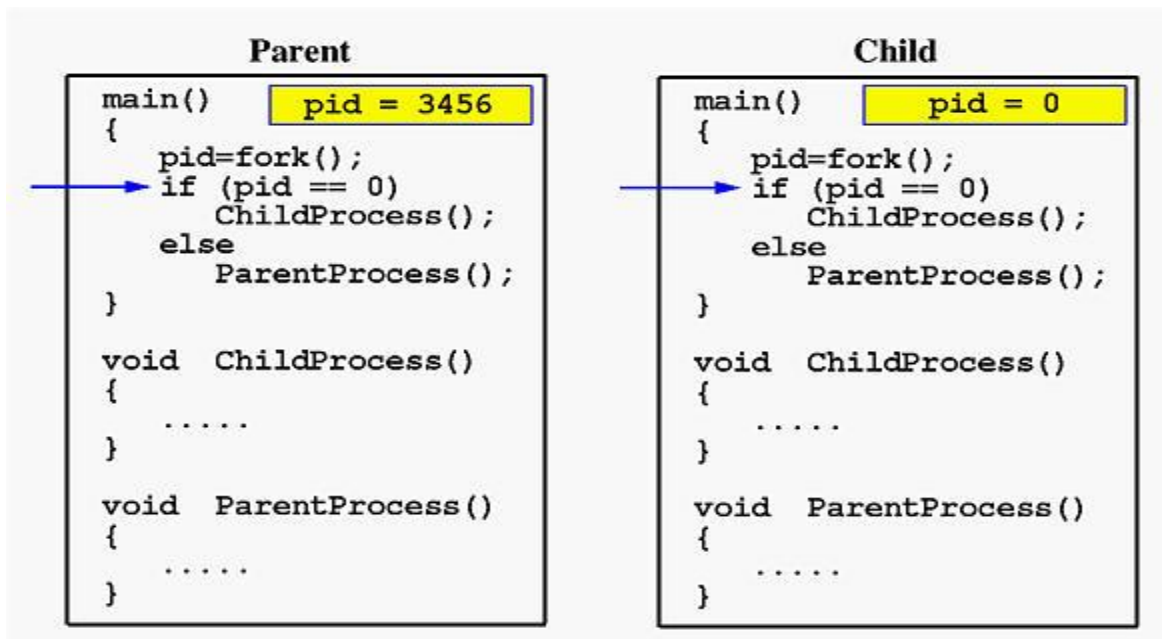
```

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**. For simplicity, **printf()** is used.

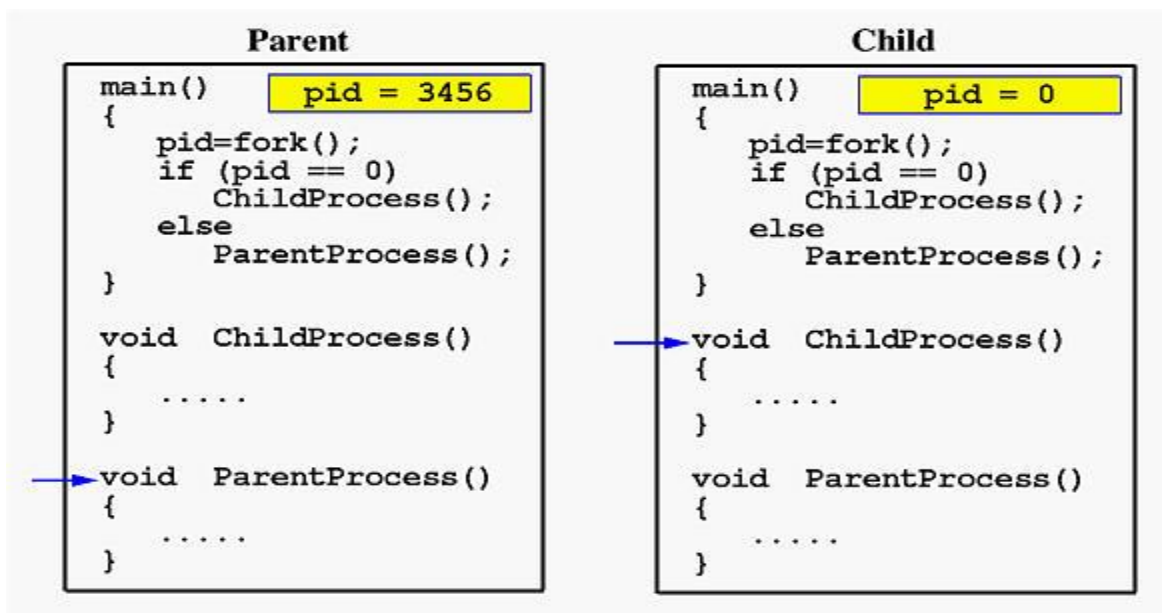
When the main program executes **fork()**, an identical copy of its address space, including the program and all data, is created. System call **fork()** returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable **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. Therefore, the value of **MAX\_COUNT** should be large enough so that both processes will run for at least two or more time quanta. If the value of **MAX\_COUNT** is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.

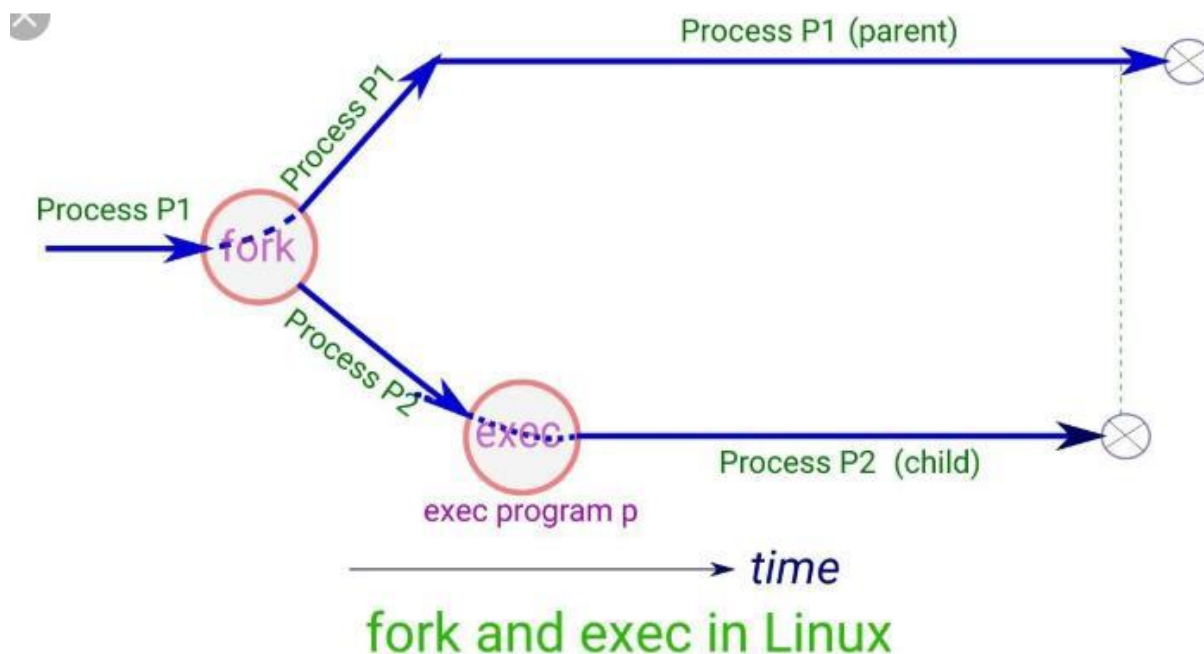


## The exec() system call

The **exec system call** is used to execute a file which is residing in an active process. When **exec** is called the previous executable file is replaced and new file is executed. More precisely, we can say that using **exec system call** will replace the old file or program from the process with a new file or program.

### ▶ The prototypes are:

- `int execl(const char *path, const char *arg, ...);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execl(const char *path, const char *arg, char *const envp[]);`
- `int execv(const char *path, char *const argv[]);`
- `int execve(const char *path, const char *argv[], char *const envp[]);`
- `int execvp(const char *file, char *const argv[]);`

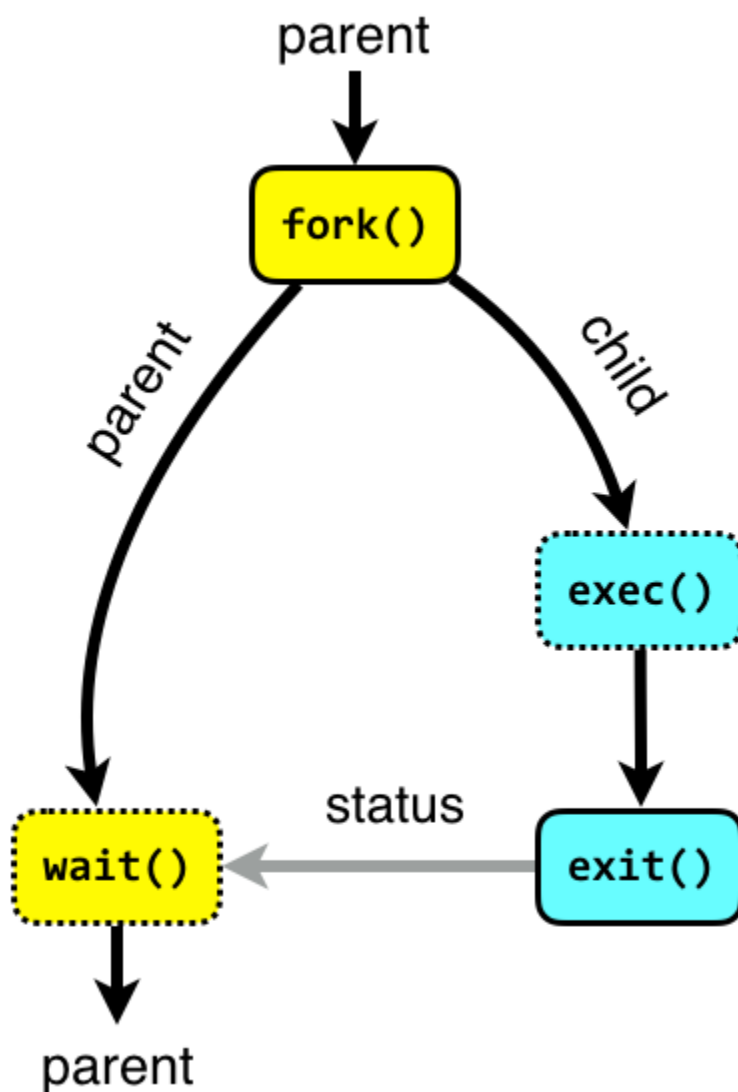




## Wait System Call in C

1. It **calls** `exit()`;
2. It returns (an int) from main.
3. It receives a signal (from the OS or another process) whose default action is to terminate.

Suspends the **calling** process until a child process ends or is stopped. More precisely, `waitpid()` suspends the **calling** process until the **system** gets status information on the child. If the **system** already has status information on an appropriate child when `waitpid()` is called, `waitpid()` returns immediately.



## EXAMPLE 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t forkStatus;

    forkStatus = fork();

    /* Child... */
    if (forkStatus == 0) {
        printf("Child is running, processing.\n");
        sleep(5);
        printf("Child is done, exiting.\n");

        /* Parent... */
    } else if (forkStatus != -1) {
        printf("Parent is waiting...\n");

        wait(NULL);
        printf("Parent is exiting...\n");

    } else {
        perror("Error while calling the fork function");
    }

    return 0;
}
```

You can compile “gcc forksleep.c” without using any arguments shown below and notice the differences

```
$ gcc -std=c89 -Wpedantic -Wall forksleep.c -o forksleep -O2
$ ./forksleep
Parent is waiting...
Child is running, processing.
Child is done, exiting.
Parent is exiting...
```

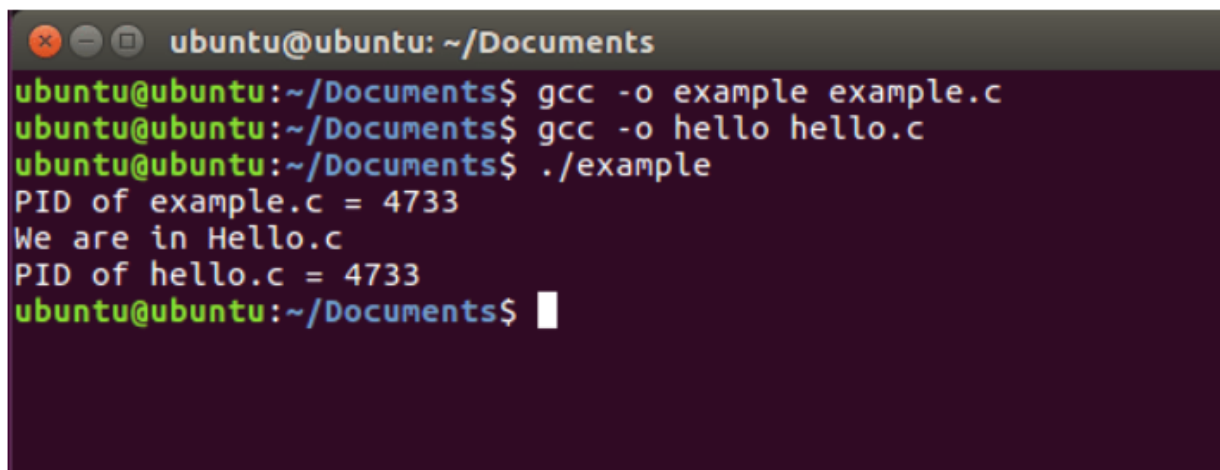
## EXAMPLE 2:

### Example.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    printf("Back to example.c");
    return 0;
}
```

### Hello.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}
```

A terminal window titled 'ubuntu@ubuntu: ~/Documents' showing the compilation and execution of two C programs. The user runs 'gcc -o example example.c' and 'gcc -o hello hello.c'. Then, they run './example', which prints 'PID of example.c = 4733' and then 'We are in Hello.c'. Finally, it prints 'PID of hello.c = 4733' and returns to the prompt.

```
ubuntu@ubuntu: ~/Documents
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
ubuntu@ubuntu:~/Documents$
```