**Exercise 2 – Process Creation using fork() and usage of wait(), getpid(), getppid()**

**Notes from Internet – Series-1**

**Fork in C**

In this article, we’ll discuss the concept of fork system call. Some experience with Unix or other operating systems like it and experience in C/C++ will help one understand this article better. We’ll learn:

* What’s a Process?
* What Does fork() Mean?
* The fork() Function
* Examples
* FAQs on Fork in C

## What’s a Process?

A process refers to an actively executing program. That’s not to say that a process is just the code that’s running. A process also includes the process stack, counter, registers, etc., that’s associated with the process, which a process control block stores.

The creation of all the processes other than the startup process requires executing the fork system call in an operating system. The parent process is the process that uses the fork() function. And any process created by a parent using a fork system call is referred to as a child process.

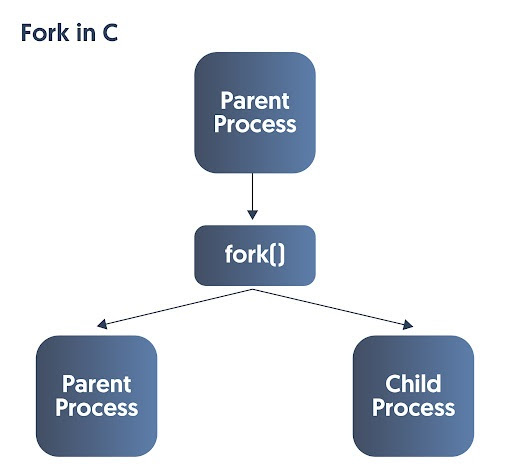
A parent process can have many child processes, but a child process can have only one parent process. If a child process does not have a parent process, the kernel was directly responsible for creating it. And if a parent process closes or crashes for any reason, it kills the child process.

## What Does fork() Mean?

Before we understand the fork function, it is worthwhile to define what fork means. In the context of routes, fork, as a verb, refers to dividing a path into two. Example: “The road forks here.” means the original road splits into two lanes from here. In the context of languages like C, C++, shell script, etc., fork refers to something similar in essence: Creating a child process by duplicating a parent process, and then they both run concurrently. Let’s now see how this works.

## The fork() Function

We use the fork() system call to create a new process from the calling process by duplicating it. The parent process does the fork() system call, and its child process is formed as a result of that call if it’s successful.



**The fork() function does not take any arguments. It just creates a child process and returns a process ID. If a fork call is successful:**

* The OS will make two identical copies of address spaces for parent and child processes. So the parent and child processes have different address spaces.
* A local variable is:  
  1. Declared inside the process2. Created when the process starts3. Lost when the process terminates
* A global variable is:  
  1. Declared outside the process2. Created as the process starts3. Lost when the program ends
* The process ID, i.e., PID of the child process created, is returned to the parent process. (In case of failure, -1 is returned to the parent process.)
* Zero is returned to the child process. (If it fails, the child process is not created.) If a child process exits at that instant or is interrupted, a signal SIGCHLD is sent to the parent process.
* Both parent and child processes independently execute the subsequent commands after the fork() system call.

Using the information above, we can use the value fork returns to distinguish between parent and child processes:

* **Negative value:** The fork call failed.
* **Zero value:** This value is returned to the child that has been newly created.
* **Positive value:** The parent received the PID of the child process as the return value.

## Examples

The best way to understand fork() calls further would be through some examples. Let’s jump right in!

### ***Example 1***

***In this example, we show that after fork(), if the call is successful, the parent and the child processes run concurrently.***

***#include <stdio.h>***

***int main()***

***{***

***fork();***

***printf("If the fork function is successful in creating a child process, I will print twice.\n");***

***return 0;***

***}***

**Output:**

If the fork function is successful in creating a child process, I will print twice.

If the fork function is successful in creating a child process, I will print twice.

### ***Example 2***

***In this example, we focus on getting process IDs and again show that both parent and child run concurrently after a successful fork system call.***

***#include <stdio.h>***

***int main()***

***{***

***int processID= fork();***

***if(processID>0)***

***{***

***printf("fork() returned a +ve value. This is the parent process, with ID: %d \n",getpid());***

***}***

***else if(processID==0)***

***{***

***printf("fork() returned a 0 value. This is a newly created child process with ID: %d \n",getpid());***

***printf("The parent process of this child process has the ID: %d\n",getppid());***

***}***

***else***

***{***

***printf("fork() returned a -ve value, so the fork system called failed and the child process could not be created\n");***

***}***

***printf("This is a single print statement. If the fork() system call was successful, both the parent and child process will run concurrently, and this statement will print twice.\n");***

***return 0;***

***}***

**Output:**

fork() returned a +ve value. This is the parent process, with ID: 25285

This is a single print statement. If the fork() system call was successful, both the parent and child process will run concurrently, and this statement will print twice.

fork() returned a 0 value. This is a newly created child process with ID: 25505

The parent process of this child process has the ID: 1

This is a single print statement. If the fork() system call was successful, both the parent and child process will run concurrently, and this statement will print twice.

If the PPID is 1, it means the parent process terminated before the child process. And the PPID of the child got updated to 1 as init became the new parent. The output of the examples we see here may vary depending on if the parent process gets terminated before or after the child process. If needed, we can use more advanced concepts like “wait” and “status” to ensure the parent doesn’t end before the child.

### ***Example 3***

***In this example, we do multiple fork calls and show that the number of times a statement is run after the calls is equal to  2^N, where N is the number of fork() system calls we’ve made. Here, the number of child processes created is equal to 2^N-1.***

***#include <stdio.h>***

***int main()***

***{***

***fork();***

***fork();***

***fork();***

***printf("The fork function was called three times. I should print eight times.\n");***

***return 0;***

***}***

**Output:**

The fork function was called three times. I should print eight times.

The fork function was called three times. I should print eight times.

The fork function was called three times. I should print eight times.

The fork function was called three times. I should print eight times.

The fork function was called three times. I should print eight times.

The fork function was called three times. I should print eight times.

The fork function was called three times. I should print eight times.

The fork function was called three times. I should print eight times.

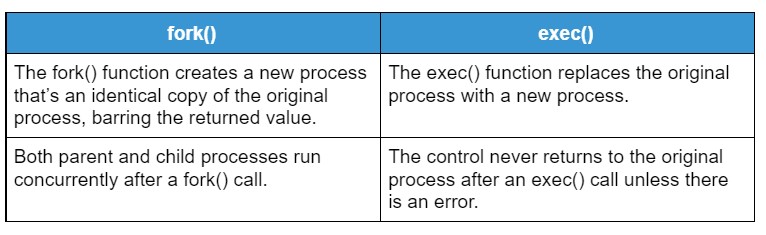
We get this output when no parent terminates before its children. Here, the number of times the line gets printed may change if a parent terminates before its children. We may, as mentioned earlier, use concepts like “wait” and “status” to ensure no parent terminates before its children.

## FAQs on Fork in C

**Question 1:** **Will the parent or the child process execute the statement after the fork() call first?**

The parent and the child process are running concurrently, and the OS could grant control to either of them first. Therefore, either of them could run the statement after the fork() call first.

**Question 2:** **What’s the difference between fork() and exec() function in C?**



**Sample Programs and output**

***#include <stdio.h>***

***#include <unistd.h>***

***int main()***

***{***

***int id;***

***printf("Hello, World!\n");***

***id = fork();***

***if (id > 0) {***

***/\*parent process\*/***

***printf("This is parent section [Process id: %d].\n", getpid());***

***}***

***else if (id == 0) {***

***/\*child process\*/***

***printf("fork created [Process id: %d].\n", getpid());***

***printf("fork parent process id: %d.\n", getppid());***

***}***

***else {***

***/\*fork creation faile\*/***

***printf("fork creation failed!!!\n");***

***}***

***return 0;***

***}***

**Output:**

Hello, World!

This is parent section [Process id: 1252].

fork created [Process id: 1253].

fork parent process id: 1252.

**Explanation:**

**This is parent section [Process id: 1252].**  
1252 is the parent process id which is the process id of the main function too; this is the parent section of new created fork.

**fork created [Process id: 1253].**  
This message will print under the fork section and assigns a new id to the newly created child process.

**fork parent process id: 1252.**  
This message will also print under the fork section, it shows parent process id of the newly created child process which is equivalent to parent process’s id (main process id). Hence we can say newly created child process is the child process of main which is known as parent process.

**Another Example using**fork()

In this example, we will print natural numbers from 1 to 10 using for loop and we will create a fork() - you will see numbers will be printed twice, because of for(), it will create duplicate copy of the process.

#include <stdio.h>

#include <unistd.h>

**int** **main**()

{

**int** id, i;

printf("Start of main...**\n**");

id = fork();

**if** (id > **0**) {

/\*parent process\*/

printf("Parent section...**\n**");

}

**else** **if** (id == **0**) {

/\*child process\*/

printf("**\n**fork created...**\n**");

}

**else** {

/\*fork creation faile\*/

printf("**\n**fork creation failed!!!**\n**");

}

printf("Printing the numbers from 1 to 10**\n**");

**for** (i = **1**; i <= **10**; i++)

printf("%d ", i);

printf("**\n**");

printf("End of the main function...**\n**");

**return** **0**;

}

**Output:**

Start of main...

Parent section...

Printing the numbers from 1 to 10

1 2 3 4 5 6 7 8 9 10

End of the main function...

fork created...

Printing the numbers from 1 to 10

1 2 3 4 5 6 7 8 9 10

End of the main function...

# C program to demonstrate zombie process

## **Zombie process**

**A process which has finished its execution but still has an entry in the process table to report to its parent process is known as a zombie process**.

In the following code, you can see that the parent will sleep for 20 sec, so it will complete its execution after 20 sec. But, Child will finish its execution using exit() system call while its parent process has gone for sleep.

After execution the child must report to its parent, So the child process entry has to be in the process table to report to its parent even after it has finished execution.

Note: [fork()](https://www.includehelp.com/c-programs/c-fork-function-linux-example.aspx) is a UNIX system call so following program will work only on UNIX based operating systems.

The following code will not produce any output. It is just for demonstration purpose.

## **Program for zombie process in C**

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

**int** **main**()

{

// fork() creates child process identical to parent

**int** pid = fork();

// if pid is greater than 0 than it is parent process

// if pid is 0 then it is child process

// if pid is -ve , it means fork() failed to create child process

// Parent process

**if** (pid > **0**)

sleep(**20**);

// Child process

**else** {

exit(**0**);

}

**return** **0**;

}

## **Orphan process**

The **process whose parent process has finished (Completed execution) or terminated and do not exists in the process table are called orphan process**. Usually, a parent process waits for its child to terminate or finish their job and report to it after execution but if he fails to do so it results in the Orphan process.

In most cases, the **Orphan process** is immediately adopted by the init process (a very first process of the system).

Note: [fork()](https://www.includehelp.com/c-programs/c-fork-function-linux-example.aspx) is a UNIX system call so following program will work only on UNIX based operating systems.

## **Program for orphan process in C**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

**int** **main**()

{

// fork() Create a child process

**int** pid = fork();

**if** (pid > **0**) {

//getpid() returns process id

// while getppid() will return parent process id

printf("Parent process**\n**");

printf("ID : %d**\n\n**", getpid());

}

**else** **if** (pid == **0**) {

printf("Child process**\n**");

// getpid() will return process id of child process

printf("ID: %d**\n**", getpid());

// getppid() will return parent process id of child process

printf("Parent -ID: %d**\n\n**", getppid());

sleep(**10**);

// At this time parent process has finished.

// So if u will check parent process id

// it will show different process id

printf("**\n**Child process **\n**");

printf("ID: %d**\n**", getpid());

printf("Parent -ID: %d**\n**", getppid());

}

**else** {

printf("Failed to create child process");

}

**return** **0**;

}

**Output**

Parent process

ID : 2274

Child process

ID: 2275

Parent -ID: 2274

Child process

ID: 2275

Parent -ID: 1103