# **Fork and Pipe Tutorial in Linux using C**

## **Part 1: Understanding fork()**

Note: the code included works in linux .

There is many online c,c++ compilers for linux

<https://onecompiler.com/cpp/43wyq24eu>

### **What is fork()?**

The Fork system call is used for creating a new process in Linux, and Unix systems, which is called the ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call.

The child process uses the same pc(program counter), same CPU registers, and same open files which use in the parent process. It takes no parameters and returns an integer value.

Below are different values returned by **fork().**

* **Negative Value**: The creation of a child process was unsuccessful, then it returns -1.
* **Zero**: Returned to the newly created child process.
* **Positive value**: Returned to parent or caller. The value contains the process ID of the newly created child process.

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <cstdlib>

int main() {

// make two process which run same

// program after this instruction

pid\_t p = fork();

if(p<0){

perror("fork fail");

exit(1);

}

printf("Hello world!, process\_id(pid) = %d \n",getpid());

return 0;

}

More readable code below....

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

#include <cstdlib>

int main() {  
 printf("Program starts - only one process exists here\n");  
   
 *// Create a new child process*  
 pid\_t pid = fork();  
   
 if (pid < 0) {  
 *// Fork failed*  
 printf("Fork creation failed!\n");  
 return 1;  
 }  
 else if (pid == 0) {  
 *// This code runs ONLY in the child process*  
 printf("Hello from child!, process\_id(pid) = %d \n",getpid());

}  
 else {  
 *// This code runs ONLY in the parent process*  
 printf("Hello from parent !, process\_id(pid) = %d \n",getpid());

}  
   
 *// This code runs in BOTH processes*  
 printf("This message appears in both parent and child\n");  
 return 0;  
}

### **Key Points:**

* fork() returns **0** to the child process
* fork() returns the **child's process ID** to the parent
* Both processes run the same code but can behave differently using the return value
* The child gets a copy of the parent's memory, file descriptors, etc.

### **Example : Calculate the number of times hello is printed.**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main(){

fork();

fork();

fork();

printf("hello\n");

return 0;

}

Moer about fork() https://www.geeksforgeeks.org/c/fork-system-call/

## **Part 2: Understanding pipe()**

### **What is a pipe?**

A pipe is like a temporary communication tunnel between processes. It has two ends:

* **Write end**: Where you put data in
* **Read end**: Where you take data out

Think of it like a water pipe - one process pours water in, another drinks from the other end.

Example code at <https://www.geeksforgeeks.org/c/c-program-demonstrate-fork-and-pipe/>

Below is more readable.

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

#include <sys/types.h>  
#include <string.h>  
  
*// Enum for pipe array indices - for readability*  
typedef enum {  
 PIPE\_READ\_END = 0,  
 PIPE\_WRITE\_END = 1  
} PipeEnd;  
  
int main() {  
 int pipe\_file\_descriptors[2];  
 char message\_to\_send[] = "Hello from the other process!";  
 char message\_buffer[100];  
   
 *// Create the pipe*  
 if (pipe(pipe\_file\_descriptors) == -1) {  
 printf("Pipe creation failed!\n");  
 return 1;  
 }  
   
 pid\_t pid = fork();  
   
 if (pid < 0) {  
 printf("Fork creation failed!\n");  
 return 1;  
 }  
   
 if (pid == 0) {  
 *// Child process - READ from pipe*  
 close(pipe\_file\_descriptors[PIPE\_WRITE\_END]); *// Close unused write end*  
   
 read(pipe\_file\_descriptors[PIPE\_READ\_END], message\_buffer, sizeof(message\_buffer)); // wait for something to be written  
 printf("Child received: %s\n", message\_buffer);  
   
 close(pipe\_file\_descriptors[PIPE\_READ\_END]); *// Close read end when done*  
 }  
 else {  
 *// Parent process - WRITE to pipe*  
 close(pipe\_file\_descriptors[PIPE\_READ\_END]); *// Close unused read end*  
   
 write(pipe\_file\_descriptors[PIPE\_WRITE\_END], message\_to\_send, strlen(message\_to\_send) + 1);  
 printf("Parent sent: %s\n", message\_to\_send);  
   
 close(pipe\_file\_descriptors[PIPE\_WRITE\_END]); *// Close write end when done*  
 }  
   
 return 0;  
}

## **Part 3: Two-Way Communication with Two Pipes**

Let's create a conversation between parent and child using two pipes:

#include <stdio.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <string.h>  
  
#define BUFFER\_SIZE 100  
  
*// Enum for pipe array indices*  
typedef enum {  
 PIPE\_READ\_END = 0,  
 PIPE\_WRITE\_END = 1

} PipeEnd;int main() {  
 *// Create two pipes for two-way communication*  
 int parent\_to\_child\_pipe[2]; *// Parent writes, child reads*  
 int child\_to\_parent\_pipe[2]; *// Child writes, parent reads*  
   
 char message\_from\_parent[] = "Hello child, how are you?";  
 char message\_from\_child[] = "I'm good parent, thanks for asking!";  
 char message\_buffer[BUFFER\_SIZE];  
   
 *// Create both pipes*  
 if (pipe(parent\_to\_child\_pipe) == -1 || pipe(child\_to\_parent\_pipe) == -1) {  
 printf("Pipe creation failed!\n");  
 return 1;  
 }  
   
 pid\_t pid = fork();  
   
 if (pid < 0) {  
 printf("Fork creation failed!\n");  
 return 1;  
 }  
   
 if (pid == 0) {  
 *// CHILD PROCESS*  
   
 *// Close unused ends - much more readable with enums!*  
 close(parent\_to\_child\_pipe[PIPE\_WRITE\_END]); *// Close write end of parent->child pipe*  
 close(child\_to\_parent\_pipe[PIPE\_READ\_END]); *// Close read end of child->parent pipe*  
   
 *// Read message from parent*  
 read(parent\_to\_child\_pipe[PIPE\_READ\_END], message\_buffer, BUFFER\_SIZE);  
 printf("Child received: %s\n", message\_buffer);  
   
 *// Send response to parent*  
 write(child\_to\_parent\_pipe[PIPE\_WRITE\_END], message\_from\_child, strlen(message\_from\_child) + 1);  
 printf("Child sent: %s\n", message\_from\_child);  
   
 *// Close used ends*  
 close(parent\_to\_child\_pipe[PIPE\_READ\_END]);  
 close(child\_to\_parent\_pipe[PIPE\_WRITE\_END]);  
 }  
 else {  
 *// PARENT PROCESS*  
   
 *// Close unused ends - very clear what we're closing*  
 close(parent\_to\_child\_pipe[PIPE\_READ\_END]); *// Close read end of parent->child pipe*  
 close(child\_to\_parent\_pipe[PIPE\_WRITE\_END]); *// Close write end of child->parent pipe*  
   
 *// Send message to child*  
 write(parent\_to\_child\_pipe[PIPE\_WRITE\_END], message\_from\_parent, strlen(message\_from\_parent) + 1);  
 printf("Parent sent: %s\n", message\_from\_parent);  
   
 *// Read response from child*  
 read(child\_to\_parent\_pipe[PIPE\_READ\_END], message\_buffer, BUFFER\_SIZE);  
 printf("Parent received: %s\n", message\_buffer);  
   
 *// Close used ends*  
 close(parent\_to\_child\_pipe[PIPE\_WRITE\_END]);  
 close(child\_to\_parent\_pipe[PIPE\_READ\_END]);  
 }  
   
 return 0;  
}

**fork()** is like making a photocopy of yourself. Both you and your copy can do different things, but you started from the same place.

**pipe()** is like setting up a walkie-talkie system between two people. You need:

* One pipe for talking (parent → child)
* Another pipe for the reply (child → parent)

**Why close unused ends?**

* If you leave both ends open, the reader might keep waiting forever thinking there might be more data
* It's like hanging up the phone after your conversation.