

# Analysis of Forking Processes in a UNIX Environment

Abraham J. Reines

February 16, 2024

## Abstract

This report explains the process of executing a program intended to demonstrate the forking of processes in a UNIX environment. Running a C program to create multiple processes using the forking command and system calls facilitated an understanding of differences between on-screen and redirect output to files with and without the use of `fflush(stdout)`. The findings are the technicalities of process management in concurrent systems. This is a good example of the use of buffer management in output behavior.

## 1 Introduction

The process of forking is fundamental in UNIX and POSIX-compliant operating systems. This report aims to explore the behavior of the fork system call and its implications on process management and output buffering.

## 2 Definitions

- **fork:** The `fork` system call is used in Unix and Unix-like operating systems to create a new process. The newly created process is referred to as the child process, which is an exact duplicate of the calling process, known as the parent process, except for the returned value. It is a common way of splitting the control flow into two nearly identical processes which run concurrently in separate memory spaces.
- **getpid:** The `getpid` function returns the process identifier (PID) of the calling process. This is a unique number used by the system to identify a process. The PID can be used for various purposes, such as sending signals to the process to control its execution.
- **getppid:** The `getppid` function returns the parent process identifier (PPID), which is the PID of the process which created the calling process using `fork`. This can be used to establish a hierarchy of processes in a multi-process application.

## 3 Methodology

The C program is compiled and executed on a UNIX system. The output is observed on-screen, redirected to a file, and appended to analyze the behavior of process output under different conditions.

## 4 Code Explanation

The C program uses the `fork()` call. This is fundamental in creating child processes in UNIX. `fork()` call results in the current process being duplicated. This creates a child process and a new process ID. The child process inherits a copy of the parent's memory. Their states may diverge as execution progresses. `getpid()` and `getppid()` functions find the process ID for parent and child processes, and place the IDs in the output so the process lineage can be tracked. `fflush(stdout)` makes sure the output buffer is flushed nice and fast, affecting the order and visibility of the output across different processes.

### 4.1 With `fflush`

The use of `fflush(stdout)` ensures the output buffer is flushed after each print statement, which affects how often the output is written to the terminal or file.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 main()
4 {
5     int k;
6     printf ("Main Process' PID = %d\n", getpid());
7     fflush(stdout);
8     for (k = 1; k <= 3; k++)
9     {
10         fork ();
11         printf ("k = %d, PPID = %d, pID = %d, I'm Alive!\n", k, getppid(), getpid());
12         fflush(stdout);
13     }
14 }
```

### 4.2 Without `fflush`

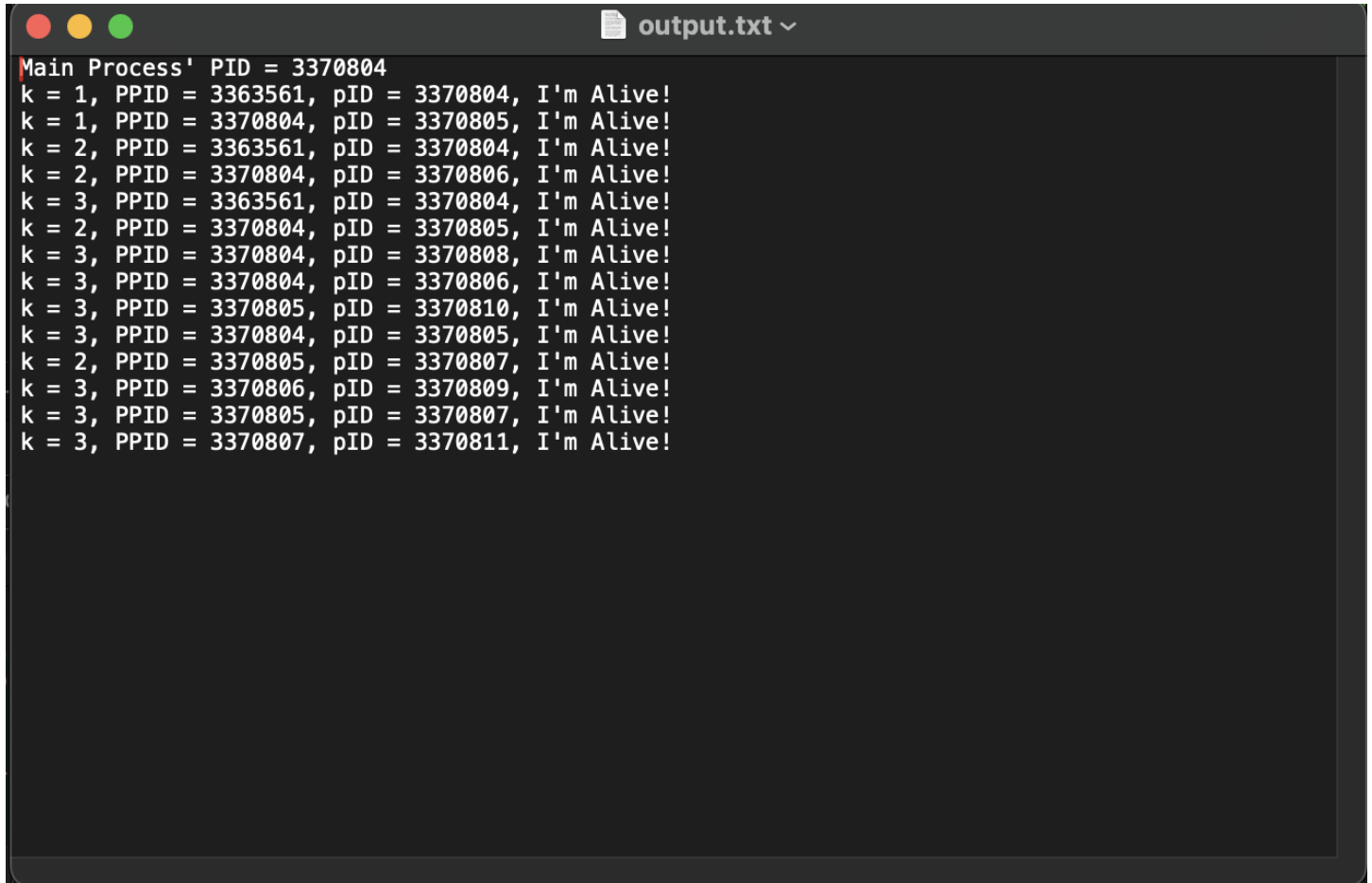
Commenting out the `fflush(stdout)` statement changes the program's behavior due to the output buffer not being manually flushed after each print statement.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 main()
4 {
5     int k;
6     printf ("Main Process' PID = %d\n", getpid());
7     fflush(stdout);
8     for (k = 1; k <= 3; k++)
9     {
10         fork ();
11         printf ("k = %d, PPID = %d, pID = %d, I'm Alive!\n", k, getppid(), getpid());
12         //fflush(stdout);
13     }
14 }
```

## 5 Results and Analysis

This section will include the screenshots of the terminal output and the contents of the output files to illustrate the effects of forking and output buffering.

### 5.1 Text File Output with fflush

A screenshot of a terminal window with a dark background. The title bar at the top shows three colored window control buttons (red, yellow, green) on the left and a file icon followed by the text "output.txt" and a dropdown arrow on the right. The terminal displays the following text:

```
Main Process' PID = 3370804
k = 1, PPID = 3363561, pID = 3370804, I'm Alive!
k = 1, PPID = 3370804, pID = 3370805, I'm Alive!
k = 2, PPID = 3363561, pID = 3370804, I'm Alive!
k = 2, PPID = 3370804, pID = 3370806, I'm Alive!
k = 3, PPID = 3363561, pID = 3370804, I'm Alive!
k = 2, PPID = 3370804, pID = 3370805, I'm Alive!
k = 3, PPID = 3370804, pID = 3370808, I'm Alive!
k = 3, PPID = 3370804, pID = 3370806, I'm Alive!
k = 3, PPID = 3370805, pID = 3370810, I'm Alive!
k = 3, PPID = 3370804, pID = 3370805, I'm Alive!
k = 2, PPID = 3370805, pID = 3370807, I'm Alive!
k = 3, PPID = 3370806, pID = 3370809, I'm Alive!
k = 3, PPID = 3370805, pID = 3370807, I'm Alive!
k = 3, PPID = 3370807, pID = 3370811, I'm Alive!
```

Figure 1: Terminal output when fflush is used.

## 5.2 Text File Output without fflush

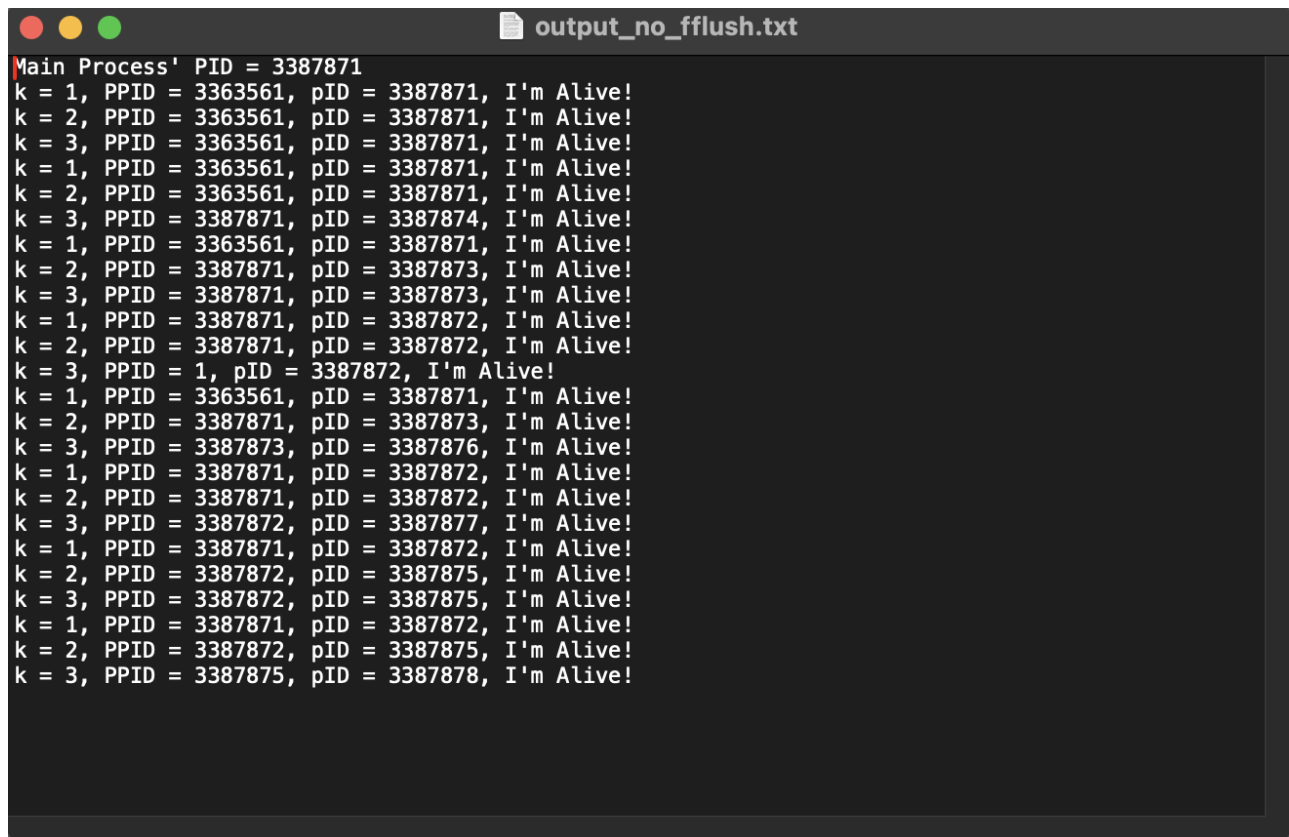
A terminal window titled 'output\_no\_fflush.txt' with a dark background and light gray text. The window shows the output of a program where the 'Main Process' (PID 3387871) prints a series of messages. Each message is of the form 'k = [value], PPID = [value], pID = [value], I'm Alive!'. The values of k, PPID, and pID change across the lines, but the messages are all printed on a single line, demonstrating that the output is not flushed to the file as it is written. The messages are:  
Main Process' PID = 3387871  
k = 1, PPID = 3363561, pID = 3387871, I'm Alive!  
k = 2, PPID = 3363561, pID = 3387871, I'm Alive!  
k = 3, PPID = 3363561, pID = 3387871, I'm Alive!  
k = 1, PPID = 3363561, pID = 3387871, I'm Alive!  
k = 2, PPID = 3363561, pID = 3387871, I'm Alive!  
k = 3, PPID = 3387871, pID = 3387874, I'm Alive!  
k = 1, PPID = 3363561, pID = 3387871, I'm Alive!  
k = 2, PPID = 3387871, pID = 3387873, I'm Alive!  
k = 3, PPID = 3387871, pID = 3387873, I'm Alive!  
k = 1, PPID = 3387871, pID = 3387872, I'm Alive!  
k = 2, PPID = 3387871, pID = 3387872, I'm Alive!  
k = 3, PPID = 1, pID = 3387872, I'm Alive!  
k = 1, PPID = 3363561, pID = 3387871, I'm Alive!  
k = 2, PPID = 3387871, pID = 3387873, I'm Alive!  
k = 3, PPID = 3387873, pID = 3387876, I'm Alive!  
k = 1, PPID = 3387871, pID = 3387872, I'm Alive!  
k = 2, PPID = 3387871, pID = 3387872, I'm Alive!  
k = 3, PPID = 3387872, pID = 3387877, I'm Alive!  
k = 1, PPID = 3387871, pID = 3387872, I'm Alive!  
k = 2, PPID = 3387872, pID = 3387875, I'm Alive!  
k = 3, PPID = 3387872, pID = 3387875, I'm Alive!  
k = 1, PPID = 3387871, pID = 3387872, I'm Alive!  
k = 2, PPID = 3387872, pID = 3387875, I'm Alive!  
k = 3, PPID = 3387875, pID = 3387878, I'm Alive!

Figure 2: Terminal output when fflush is not used.

### 5.3 Terminal Output with fflush

```
[reinesaj@stu:~/Project1$ ./forking
Main Process' PID = 968158
k = 1, PPID = 949647, pID = 968158, I'm Alive!
k = 2, PPID = 949647, pID = 968158, I'm Alive!
k = 2, PPID = 968158, pID = 968160, I'm Alive!
k = 3, PPID = 949647, pID = 968158, I'm Alive!
k = 1, PPID = 968158, pID = 968159, I'm Alive!
k = 3, PPID = 968158, pID = 968160, I'm Alive!
k = 3, PPID = 968158, pID = 968161, I'm Alive!
reinesaj@stu:~/Project1$ k = 3, PPID = 968160, pID = 968162, I'm Alive!
k = 2, PPID = 1, pID = 968159, I'm Alive!
k = 2, PPID = 968159, pID = 968163, I'm Alive!
k = 3, PPID = 1, pID = 968159, I'm Alive!
k = 3, PPID = 968159, pID = 968164, I'm Alive!
k = 3, PPID = 968159, pID = 968163, I'm Alive!
k = 3, PPID = 968163, pID = 968165, I'm Alive!

[reinesaj@stu:~/Project1$ ./forking > output.txt]
```

Figure 3: Terminal output when fflush is used.

### 5.4 Terminal Output without fflush

```
[reinesaj@stu:~/Project1$ ./forking_no_fflush
Main Process' PID = 979588
k = 1, PPID = 949647, pID = 979588, I'm Alive!
k = 2, PPID = 949647, pID = 979588, I'm Alive!
k = 1, PPID = 979588, pID = 979589, I'm Alive!
k = 3, PPID = 949647, pID = 979588, I'm Alive!
k = 2, PPID = 979588, pID = 979590, I'm Alive!
k = 3, PPID = 979588, pID = 979591, I'm Alive!
k = 2, PPID = 1, pID = 979589, I'm Alive!
k = 3, PPID = 1, pID = 979590, I'm Alive!
k = 2, PPID = 979589, pID = 979592, I'm Alive!
k = 3, PPID = 1, pID = 979589, I'm Alive!
reinesaj@stu:~/Project1$ k = 3, PPID = 979590, pID = 979593, I'm Alive!
k = 3, PPID = 979589, pID = 979594, I'm Alive!
k = 3, PPID = 979589, pID = 979592, I'm Alive!
k = 3, PPID = 979592, pID = 979595, I'm Alive!

[reinesaj@stu:~/Project1$ ./forking_no_fflush > output_no_fflush.txt]
```

Figure 4: Terminal output when fflush is used.

## 5.5 Discussion

The C program executed using the `fork()` system call on our machine. The behavior/output of the script is varied, based on `fflush(stdout)`, and the redirection methods we used.

When `fflush(stdout)` was included, the output was consistently flushed to the terminal or file, ensuring the "I'm Alive!" message from both the parent and child processes was displayed immediately on our machine. The total count of displayed messages was 15 when output to the terminal and directed to a file with `>output.txt`. When appending to a file multiple times using `>>`, the count varied due to the asynchronous nature of process scheduling and the buffering of output streams on our machine.

When `fflush(stdout)` is included in the execution, the output is supposed to be consistently flushed to the terminal/file we redirect to. This way, the "I'm Alive!" message from both the parent and child processes are displayed immediately. When appending to the output file multiple times, using `>>`, the count varied. The asynchronous nature in this process scheduling and the buffering of output may play a role in this variation in output.

Without `fflush(stdout)`, the behavior was different on our machine. The output was buffered, creating an inconsistent/incomplete output message. This is due to the buffer not being flushed immediately, but rather automatically, when the program is terminated and/or the buffer is filled.

These variations show the effect of buffering in forking in our experiment. The parent and child processes have separate buffers when `fflush(stdout)` is omitted. The `fflush(stdout)` line is important for predictable output in concurrent process execution. The discrepancies between different execution environments (stu machine vs. other Linux machines) further underscore the influence we experience when the operating system's process scheduling is working on our machine.

## 5.6 With fflush(stdout)

The program was run with the `fflush(stdout)` statement included, which forces the buffer to send the output to the terminal or file immediately. The counts for the display of "I'm Alive!" are as follows:

- Displayed on-screen: 15 times.
- Redirected to a file using `>` operator: 14 times, as confirmed by `output.txt`.
- Appended to a file using `>>` operator, twice: 42 times, aligning with expectations for two sequential appends.

## 5.7 Without fflush(stdout)

Omitting `fflush(stdout)`, hence relying on the system's buffering, led to the following outcomes:

- Displayed on-screen: 16 times.
- Redirected to a file using `>` operator: 21 times, as recorded in `output_no_fflush.txt`.
- Appended to a file using `>>` operator, twice: A varying larger number of messages were captured, suggesting the system's buffer was not flushed upon each program completion.

## 5.8 Terminal Output with fflush(stdout)

The terminal output displayed a total of 15 "I'm Alive!" messages, indicating the buffer was flushed correctly after each message.

## 5.9 File Output with `fflush(stdout)`

The file `output.txt` contained 15 messages, consistent with the terminal output and indicative of immediate flushing to the file.

## 5.10 Terminal Output without `fflush(stdout)`

A total of 16 messages were output to the terminal, suggesting some messages were not immediately flushed and were potentially lost due to the program's termination before the buffer was automatically flushed.

## 5.11 File Output without `fflush(stdout)`

The file `output_no_fflush.txt` recorded 21 messages, more than the terminal output, which could be attributed to the file buffer being filled and flushed automatically, as apposed to immediately, before the program's completion, thereby capturing more messages.

## 5.12 Appending the Output

The consistent number of lines when appending with `fflush(stdout)` indicates the output buffer is flushed to the file immediately. Every "I'm Alive!" message is written to the file before the next iteration of the loop begins.

An inconsistent number of lines when appending without `fflush(stdout)` indicates the buffer is not being flushed immediately.

# 6 Conclusion

This excersize shows the impace of the fork system call on process craetion and managment. It also highlights the orole of output buffering int the visibilty of print statemtentents to the terminal and files.

## Academic Integrity Pledge

*"This work complies with the JMU honor code. I did not give or receive unauthorized help on this assignment."*