Analysis of Forking Processes in a UNIX Environment

Abraham J. Reines

February 15, 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 undertanding 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 managemetn 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 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 another file to analyze the behavior of process output under different conditions.

3 Code Explanation

The C program [1] uses the `fork()` call. This is fundamental in craeting 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 inherents a copy of the parent's memory. Their states may diverage 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.

[1] See next page for the full code listing.

3.1 With fflush

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
  int k;
  printf ("Main Process' PID = %d\n", getpid());
  fflush(stdout);
  for (k = 1; k <= 3; k++)
  {
    fork ();
    printf ("k = %d, PPID = %d, pID = %d, I'm Alive!\n", k, getppid(), getpid
   ());
    fflush(stdout);
  }
}
```

3.2 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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
  int k;
  printf ("Main Process' PID = %d\n", getpid());
  fflush(stdout);
  for (k = 1; k <= 3; k++)
  {
    fork ();
    printf ("k = %d, PPID = %d, pID = %d, I'm Alive!\n", k, getppid(), getpid
   ());
    //fflush(stdout);
  }
}
```

4 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.

4.1 Terminal Output with fflush

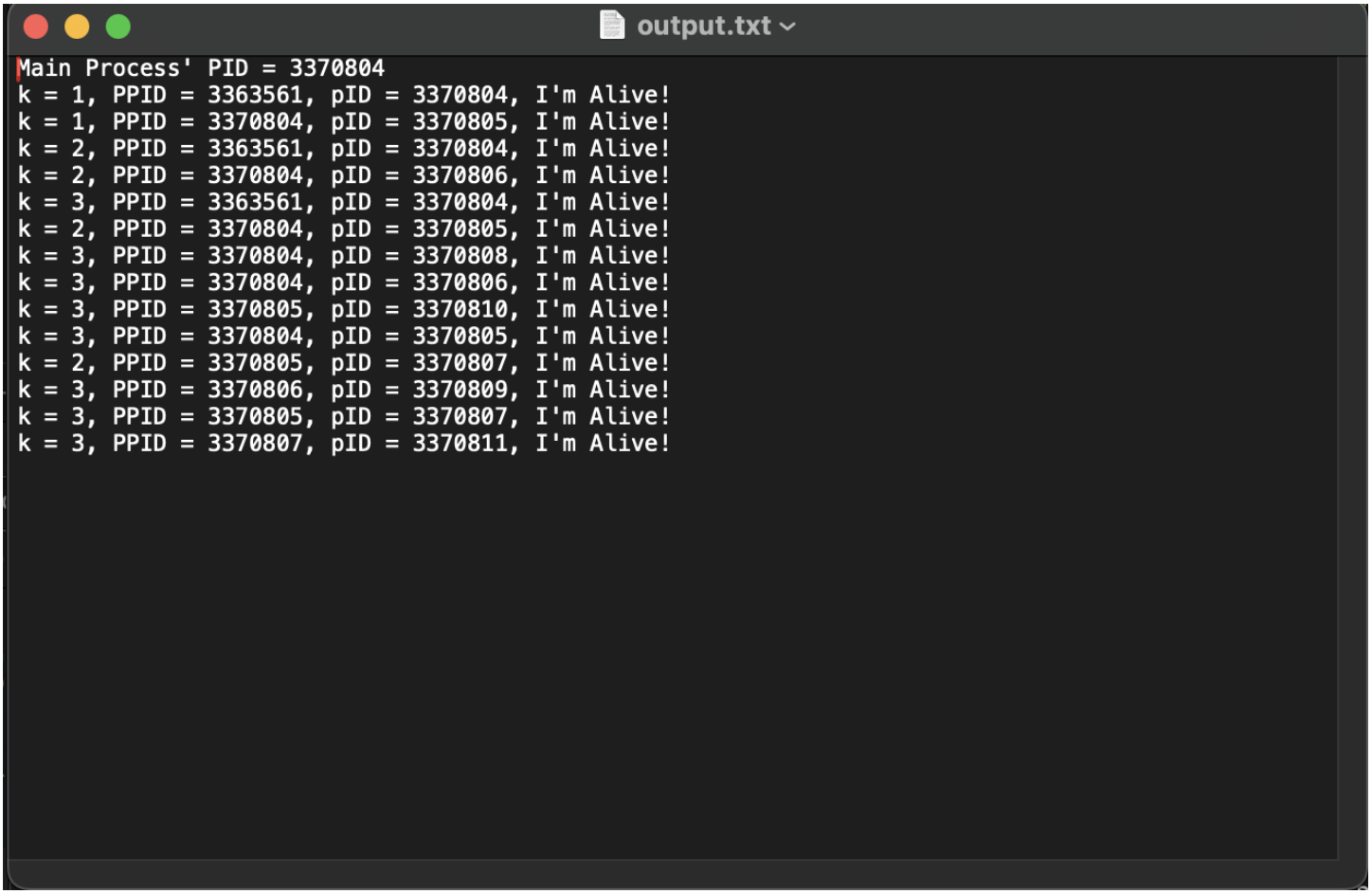

Figure 1: Terminal output when fflush is used.

4.2 Terminal Output without fflush

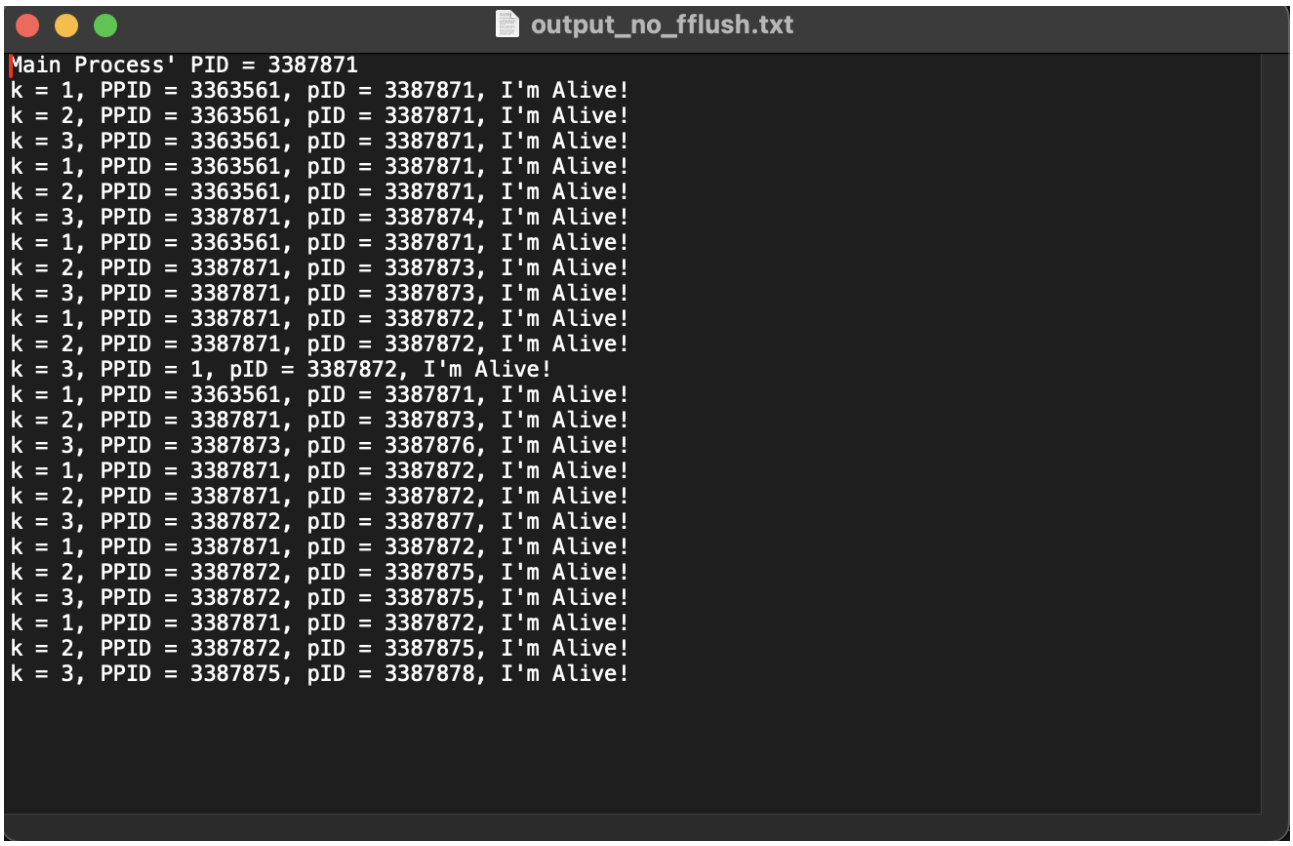

Figure 2: Terminal output when fflush is not used.

4.3 File Output Analysis

In output buffering, this compares terminal and file outputs with and without the use of `fflush`. `fflush` results in immediate output to the terminal, while without fflush, the output is buffered, leading to a non-deterministci order in terminal output. Buffered I/O may lead to race conditions.

5 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 statemtents 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."