CSE344 Homework 2 Report: Inter-Process Communication with FIFOs

Tugay Talha İçen

April, 2024

1 Introduction

This report documents the implementation and testing of a C program designed to fulfill the requirements of Homework 2 for CSE344. The program utilizes FIFOs (named pipes) for inter-process communication (IPC) between a parent process and two child processes. The parent process generates random numbers, sends them to the child processes via FIFOs, and manages their execution. Child Process 1 calculates the sum of the received numbers, while Child Process 2 performs multiplication based on a command received from the parent.

2 Implementation

2.1 Code Structure and Functionality

The program is structured within a single C file (main.c) and utilizes various system libraries for process management, FIFO operations, and signal handling.

- **FIFOs and Error Handling**: The program creates two FIFOs (fifo1 and fifo2) using the mkfifo() function. Error handling is implemented to check for potential failures during FIFO creation and to provide informative messages in case of errors.
- **Process Management**: The fork() system call is used to create two child processes. Each child process is assigned to a specific FIFO for reading and writing data.
- **Parent Process**: * The parent process generates an array of random numbers based on the integer argument provided at runtime. * It writes these random numbers to both FIFOs. * Additionally, the "multiply" command is written to fifo2 for Child Process 2. * A signal handler for SIGCHLD is set up to monitor child process termination. * The parent process enters a loop, printing "Proceeding..." every 2 seconds and checking the child process counter. * When both child processes have exited (indicated by the counter), the program terminates.

- **Child Process 1**: * Child Process 1 opens fifo1 for reading and reads the random numbers. * It calculates the sum of the numbers and writes the result to fifo2.
- **Child Process 2**: * Child Process 2 opens fifo2 for reading and reads the random numbers and the command. * It validates the command ("multiply") and proceeds to calculate the product of the numbers. * It also reads the sum calculated by Child Process 1 from fifo2. * Finally, it prints the sum of both results (sum and product).

2.2 Error Handling and Exit Statuses

The code includes comprehensive error handling mechanisms to catch potential issues during FIFO operations, process forking, and signal handling. Descriptive error messages are printed to the console to aid in debugging. Additionally, the program captures and prints the exit statuses of child processes to provide further insight into their termination.

2.3 Zombie Process Prevention

The implemented signal handler effectively prevents zombie processes by calling waitpid() within the SIGCHLD handler. This ensures that terminated child processes are properly reaped and their resources are released.

3 Testing and Results

The program was compiled and tested on Ubuntu 22.04.3 LTS using GCC 11.4.0. The valgrind tool was used to check for memory leaks, confirming that the program manages memory efficiently. The provided test scenario was executed successfully, demonstrating the correct functionality of the program: FIFOs were created and utilized for communication without errors. Random numbers were generated, transmitted, and processed as expected by each child process. Child processes exited successfully, and their exit statuses were printed by the parent process. The final sum of results from both child processes was correctly calculated and displayed.

```
(base) root@DESKTOP-D58326S:/home/c# make clean
rm -f *.o *.out
rm -f fifo1 fifo2
(base) root@DESKTOP-D58326S:/home/c# make build
gcc -c -o main.o main.c -std=gnu99
gcc -o main.out main.o
(base) root@DESKTOP-D58326S:/home/c# valgrind ./main.out 10
==117105== Memcheck, a memory error detector
==117105== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et
    al.
```

```
==117105== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright
    info
==117105== Command: ./main.out 10
==117105==
Proceeding...
Proceeding...
Proceeding...
Proceeding...
Proceeding...
Proceeding...
Received numbers in Child Process 2: 2 7 2 7 5 1 5 1 7 9
Received numbers in Child Process 1: 2 7 2 7 5 1 5 1 7 9
Command: multiply
Sum of random numbers: 46
==117107==
==117107== HEAP SUMMARY:
==117107==    in use at exit: 0 bytes in 0 blocks
==117107==  total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==117107==
==117107== All heap blocks were freed -- no leaks are possible
==117107==
==117107== For lists of detected and suppressed errors, rerun with: -s
==117107== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Child process 117107 exited with status: 0
Proceeding...
Sum of results from all child processes: 308746
==117108==
==117108== HEAP SUMMARY:
==117108==    in use at exit: 0 bytes in 0 blocks
==117108==  total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==117108==
==117108== All heap blocks were freed -- no leaks are possible
==117108==
==117108== For lists of detected and suppressed errors, rerun with: -s
==117108== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Child process 117108 exited with status: 0
Program finished sucessfully, Parent process exiting...
==117105==
==117105== HEAP SUMMARY:
==117105==    in use at exit: 0 bytes in 0 blocks
==117105==  total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==117105==
==117105== All heap blocks were freed -- no leaks are possible
==117105==
==117105== For lists of detected and suppressed errors, rerun with: -s
==117105== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
(base) root@DESKTOP-D58326S:/home/c#
```

4 Conclusion

The implemented program successfully demonstrates IPC using FIFOs between a parent process and two child processes. It adheres to the requirements outlined in Homework 2 and includes robust error handling and zombie process prevention mechanisms.

5 Bonus Features

The program implements both bonus features: **Zombie Process Protection**: The SIGCHLD handler and waitpid() calls ensure that child processes are reaped promptly, preventing the creation of zombie processes. **Exit Status Reporting**: The exit status of each child process is printed within the SIGCHLD handler, providing information about their termination status.