CSE 344 SYSTEM PROGRAMMING

HOMEWORK #2 REPORT

BUKET GENÇER

210104004298

18.04.2024

Table of Contents

Introduction and Project Description

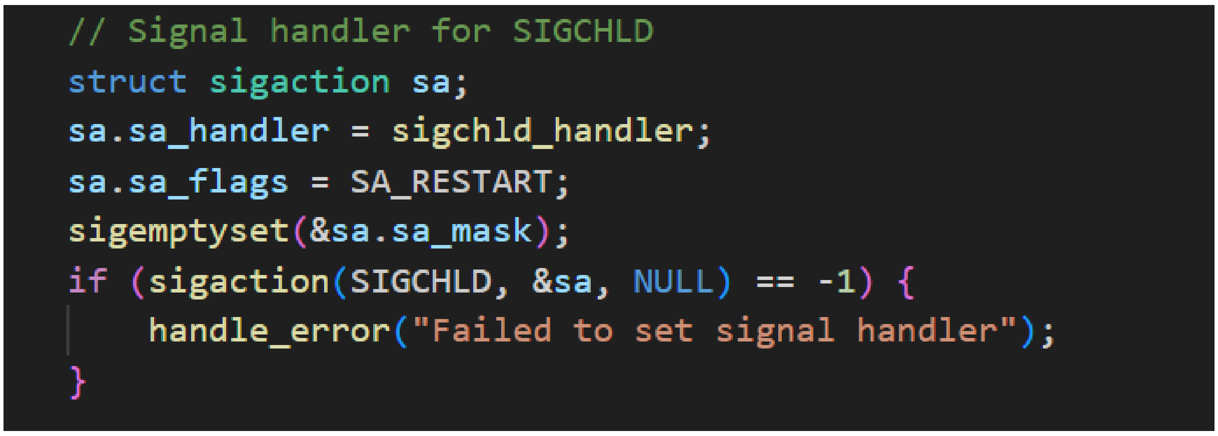
This project aims to establish inter-process communication (IPC) using FIFOs (named pipes) between two child processes under a parent process in a Linux environment. The primary focus is on using system-level programming capabilities in C to manage processes, handle errors, and synchronize tasks through signals and FIFOs. The project demonstrates how to efficiently perform and manage tasks like data writing, reading, and mathematical operations across processes, ensuring robust error handling and effective process synchronization.

Code Description and Analysis

This section analyzes the implementation details of the IPC communication project. It focuses on the key components of the code, explaining the functionalities and the reasons behind using specific functions and structures.

Signal Handler Setup

To manage child processes effectively and clean up resources after their termination, a signal handler for SIGCHLD is set up. This signal is sent to the parent process whenever a child process terminates.



```c
// Signal handler for SIGCHLD
struct sigaction sa;
sa.sa_handler = sigchld_handler;
sa.sa_flags = SA_RESTART;
sigemptyset(&sa.sa_mask);
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    handle_error("Failed to set signal handler");
}
```

The sigaction system call is used to define the action taken by a process on receipt of a specific signal (SIGCHLD in this case). The handler sigchld_handler clears up the zombie processes using waitpid, which non-blockingly reaps any child that has terminated. This setup ensures that no child processes remain as zombies, freeing up system resources.

```c
volatile sig_atomic_t child_counter = 0;

void sigchld_handler(int signum) {
    printf("SIGCHLD received\n");
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        printf("Child %d finished with status %d\n", pid, WEXITSTATUS(status));
        child_counter++;
    }
}

void handle_error(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}
```

A global variable child_counter is declared with volatile sig_atomic_t type to safely access it from within the signal handler and the main flow of the program. Its purpose is to count the number of child processes that have terminated.

The provided code sets up a signal handler to deal with terminated child processes. The sigchld_handler function is triggered by the SIGCHLD signal, incrementing the child_counter for each exited child process and printing its PID and exit status. This ensures that the system doesn't leave behind zombie processes. Additionally, a handle_error function is defined to output error messages and terminate the program when something goes wrong.

FIFO Creation and Management

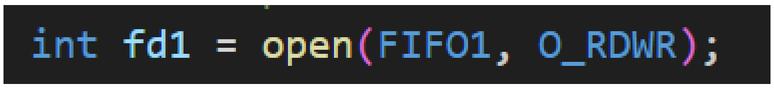
FIFOs, or named pipes, are crucial in facilitating IPC (Inter-Process Communication) between the parent and child processes in this project. They serve as conduits for transmitting data (an array of integers) and commands ("multiply") effectively and efficiently between these processes.

FIFO Creation:

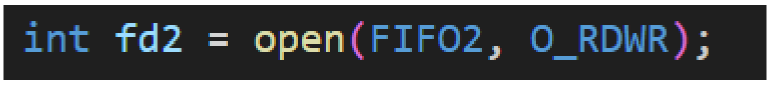
```c
// Unlink the FIFOs if they already exist
unlink(FIFO1);
unlink(FIFO2);

// Create the FIFOs
if (mkfifo(FIFO1, 0666) == -1 || mkfifo(FIFO2, 0666) == -1) {
    handle_error("Failed to create FIFOs"); // 0666 is the permission for the FIFOs
}
```

The code first attempts to unlink (remove) any pre-existing FIFOs with the same names to avoid conflicts. It then creates two new FIFOs using the mkfifo system call. The 0666 permissions ensure that any process can read from or write to these FIFOs. This setup is crucial for the proper initialization of communication paths before any read or write operations take place.

Managing FIFOs:



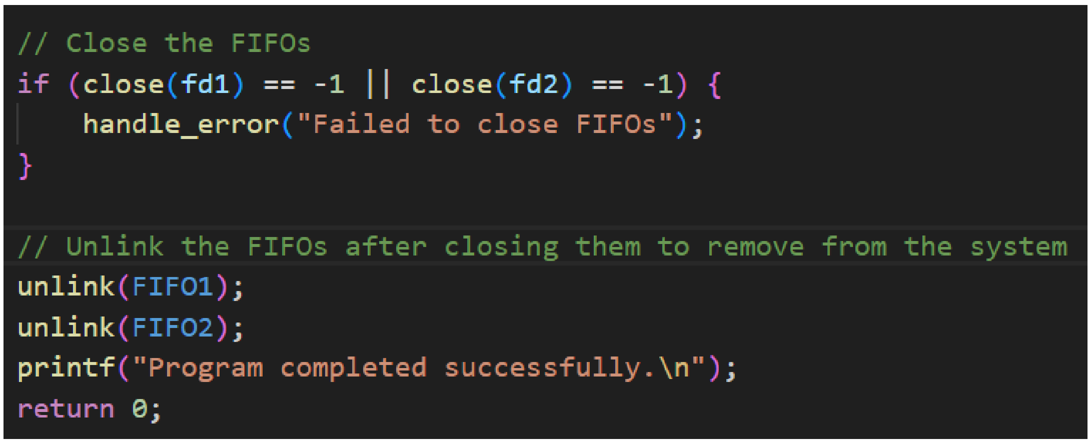
```c
int fd1 = open(FIFO1, O_RDWR);
```



```c
int fd2 = open(FIFO2, O_RDWR);
```

Here, the FIFOs are opened in read-write mode (O_RDWR). Opening the FIFOs in this mode is critical as it prevents the open call from blocking; normally, opening a FIFO for reading or writing only blocks until another process opens the file for writing or reading, respectively. This non-blocking feature is essential for the parent process, which needs to perform both operations.

Closing and Unlinking FIFOs:

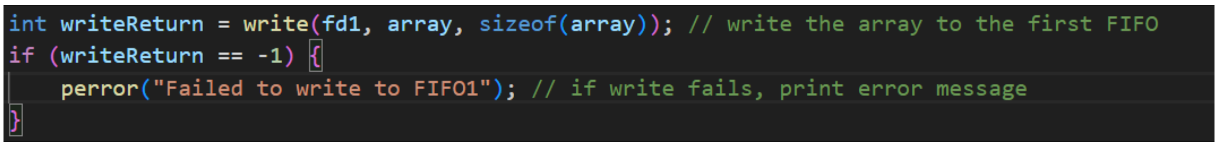


```c
// Close the FIFOs
if (close(fd1) == -1 || close(fd2) == -1) {
    handle_error("Failed to close FIFOs");
}

// Unlink the FIFOs after closing them to remove from the system
unlink(FIFO1);
unlink(FIFO2);
printf("Program completed successfully.\n");
return 0;
```

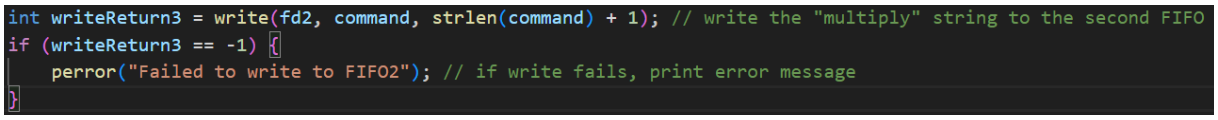
At the end of the process, it is crucial to close and unlink the FIFOs to release system resources and ensure that no data leakage or security holes remain. Properly managing FIFO closure and removal is as critical as their creation for maintaining system integrity.

Data Writing and Reading Operations

Data Writing Operations:



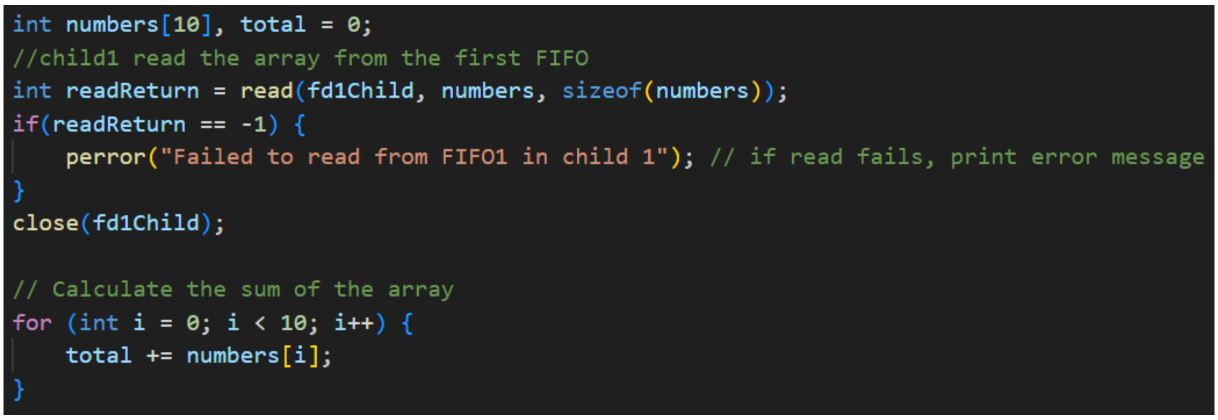
```c
int writeReturn = write(fd1, array, sizeof(array)); // write the array to the first FIFO
if (writeReturn == -1) {
    perror("Failed to write to FIFO1"); // if write fails, print error message
}
```



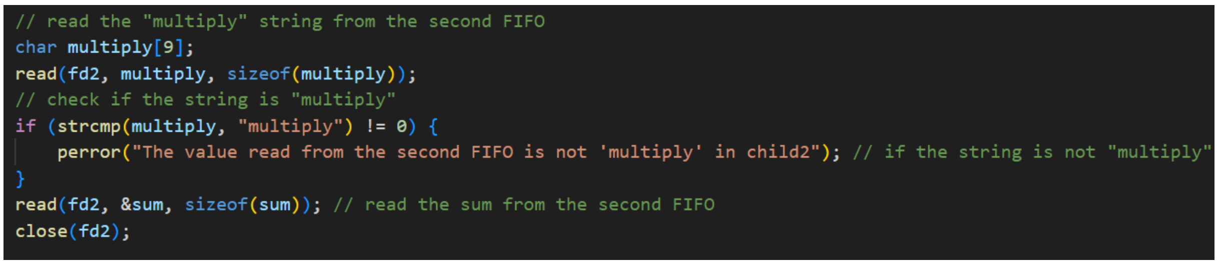
```c
int writeReturn3 = write(fd2, command, strlen(command) + 1); // write the "multiply" string to the second FIFO
if (writeReturn3 == -1) {
    perror("Failed to write to FIFO2"); // if write fails, print error message
}
```

The parent process writes an array of integers to FIFO1 and a command to FIFO2. These operations are encapsulated in write calls, which handle the transfer of data to the child processes. Proper error checking follows each write operation to ensure data integrity and handle any issues immediately.,

Data Reading and Handling by Child Processes:



```c
int numbers[10], total = 0;
//child1 read the array from the first FIFO
int readReturn = read(fd1Child, numbers, sizeof(numbers));
if(readReturn == -1) {
    perror("Failed to read from FIFO1 in child 1"); // if read fails, print error message
}
close(fd1Child);

// Calculate the sum of the array
for (int i = 0; i < 10; i++) {
    total += numbers[i];
}
```



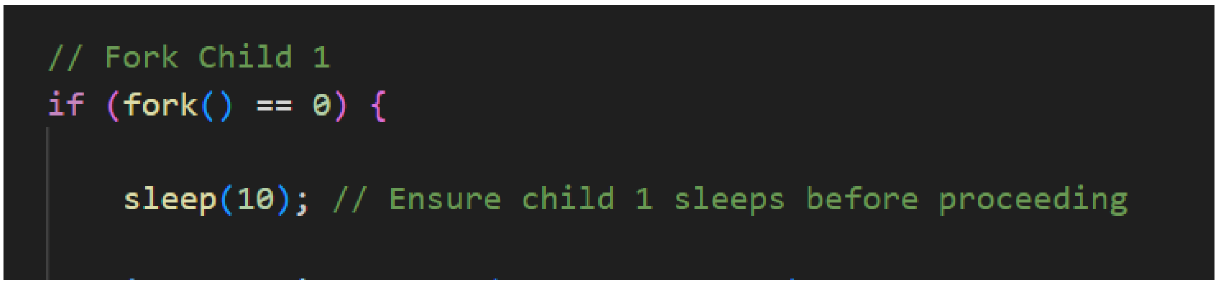
```c
// read the "multiply" string from the second FIFO
char multiply[9];
read(fd2, multiply, sizeof(multiply));
// check if the string is "multiply"
if (strcmp(multiply, "multiply") != 0) {
    perror("The value read from the second FIFO is not 'multiply' in child2"); // if the string is not "multiply"
}
read(fd2, &sum, sizeof(sum)); // read the sum from the second FIFO
close(fd2);
```

Child Process 1 opens FIFO1 for reading and retrieves an array of integers for processing. Child Process 2 reads the command from FIFO2 and checks it against expected commands. This structured approach ensures that each child process performs its designated task based on the data received through the FIFOs.

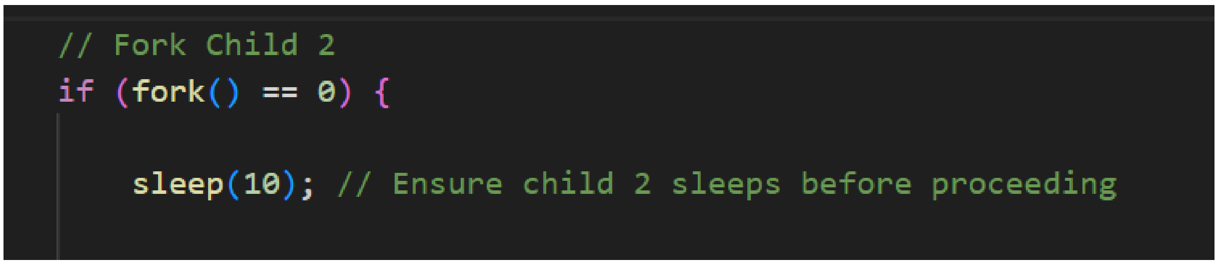
Process Management

Process management in this project involves creating child processes, assigning them tasks, and ensuring their proper termination.

Process Management:



```
// Fork Child 1
if (fork() == 0) {

    sleep(10); // Ensure child 1 sleeps before proceeding
```



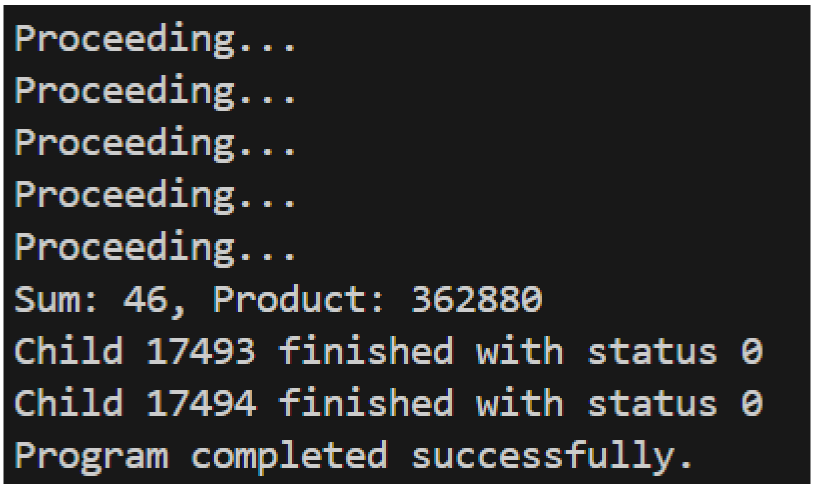
```
// Fork Child 2
if (fork() == 0) {

    sleep(10); // Ensure child 2 sleeps before proceeding
```

The code uses the fork() system call to create two separate child processes. Upon each call to fork(), the current process is duplicated. The return value of fork() determines the process context: it returns 0 in the child process and the child's PID in the parent process.

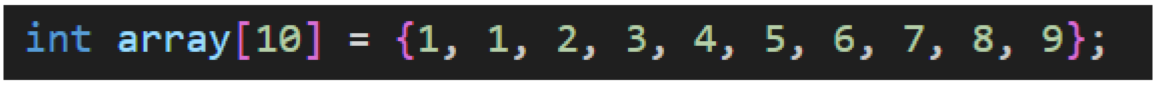
- Child 1 Creation: The first fork() creates the first child process. Inside the if statement that checks for a 0 return value, we place the tasks intended for Child 1, such as opening the FIFO to read data, processing that data, and then writing back the results. After performing its tasks, Child 1 calls exit(0) to terminate, ensuring a clean exit with a status code of 0, indicating success.
- Child 2 Creation: Similarly, the second fork() creates the second child process. Its tasks are specified within its context, which includes reading the command and sum from the FIFOs, performing the multiplication operation, and handling the output.

The parent process, after forking the child processes, enters a loop where it periodically prints "Proceeding..." to indicate it is active and waiting for the child processes to terminate. The signal handler set up previously catches the SIGCHLD signal to handle the termination of the child processes, thus preventing any zombies.

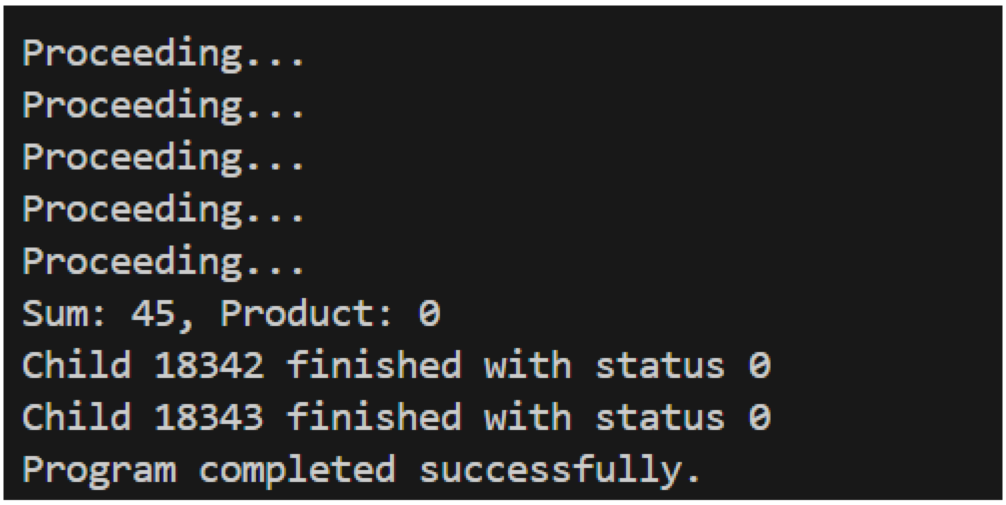
Testing



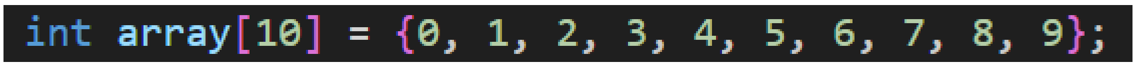
```
Proceeding...
Proceeding...
Proceeding...
Proceeding...
Proceeding...
Sum: 46, Product: 362880
Child 17493 finished with status 0
Child 17494 finished with status 0
Program completed successfully.
```



```c
int array[10] = {1, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

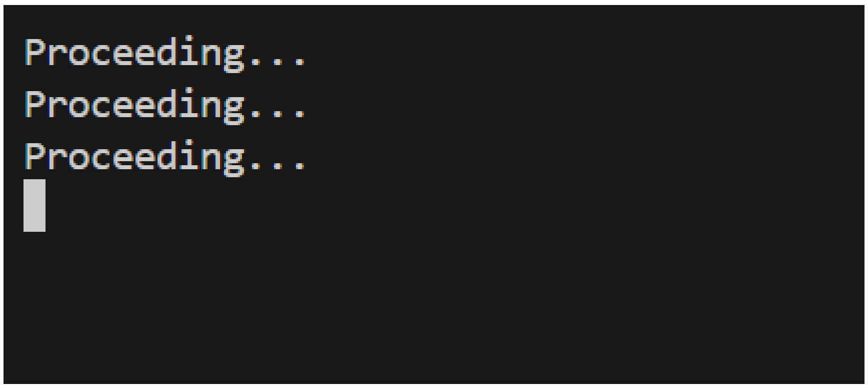


```
Proceeding...
Proceeding...
Proceeding...
Proceeding...
Proceeding...
Sum: 45, Product: 0
Child 18342 finished with status 0
Child 18343 finished with status 0
Program completed successfully.
```



```c
int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

The output "Proceeding..." repeatedly indicates that the parent process is operational and waiting for the child processes to complete their execution.



```
Proceeding...
Proceeding...
Proceeding...
```

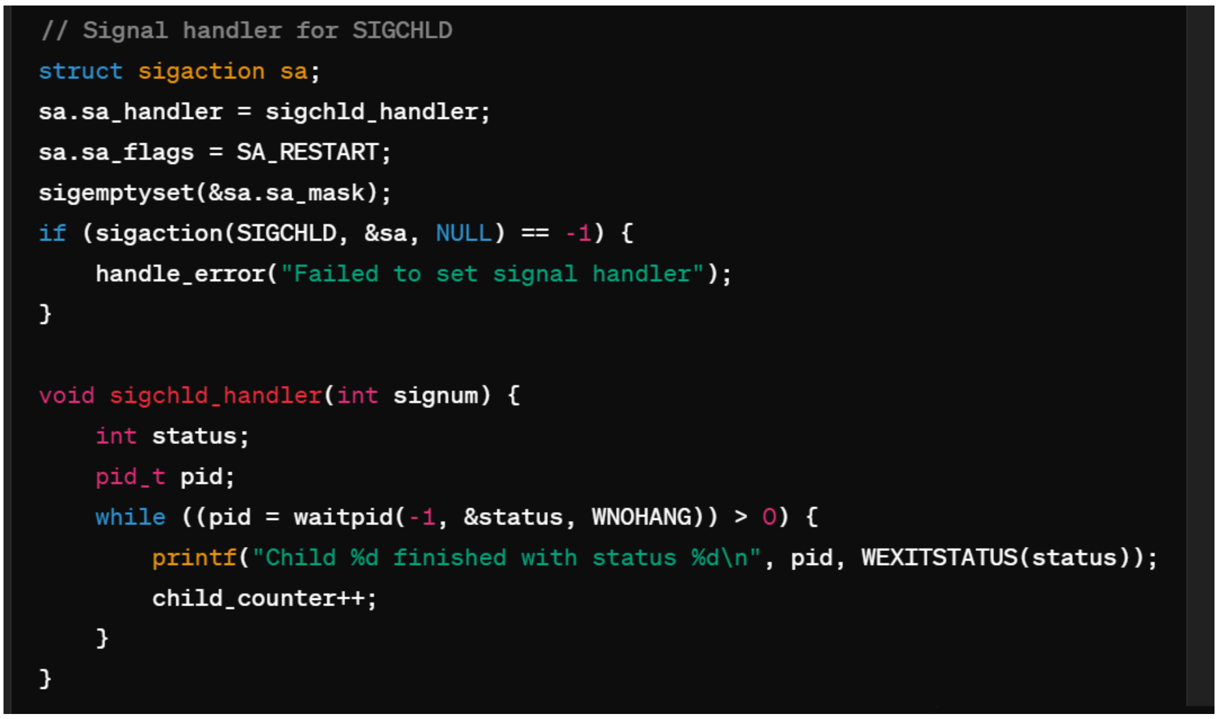
Messages indicating that children with PIDs 17493 and 17494 finished with status 0 confirm the successful execution and termination of the child processes without any errors.

"Program completed successfully." validates the overall success of the program execution, including the correct reception and handling of termination signals by the parent process.

Performance Evaluation

Zombie Process Control

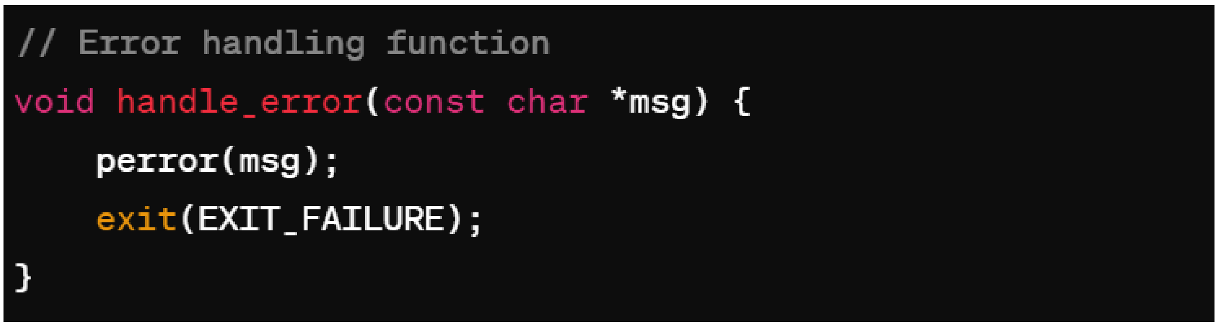
The program effectively manages child processes to prevent the occurrence of zombie processes. The signal handling mechanism is set up with sigaction() to catch SIGCHLD, which is triggered when a child process terminates. The signal handler sigchld_handler() uses waitpid() with the WNOHANG option, which collects the termination status of child processes that have finished execution without blocking. This ensures that no child process remains in a zombie state. The output indicates that all child processes finished with status 0, confirming their successful reaping.



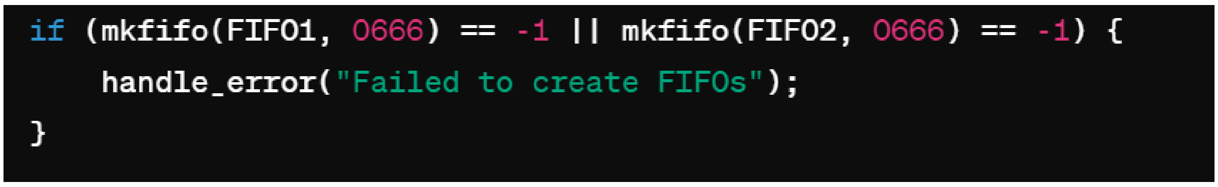
```c
// Signal handler for SIGCHLD
struct sigaction sa;
sa.sa_handler = sigchld_handler;
sa.sa_flags = SA_RESTART;
sigemptyset(&sa.sa_mask);
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    handle_error("Failed to set signal handler");
}

void sigchld_handler(int signum) {
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        printf("Child %d finished with status %d\n", pid, WEXITSTATUS(status));
        child_counter++;
    }
}
```

Error Management

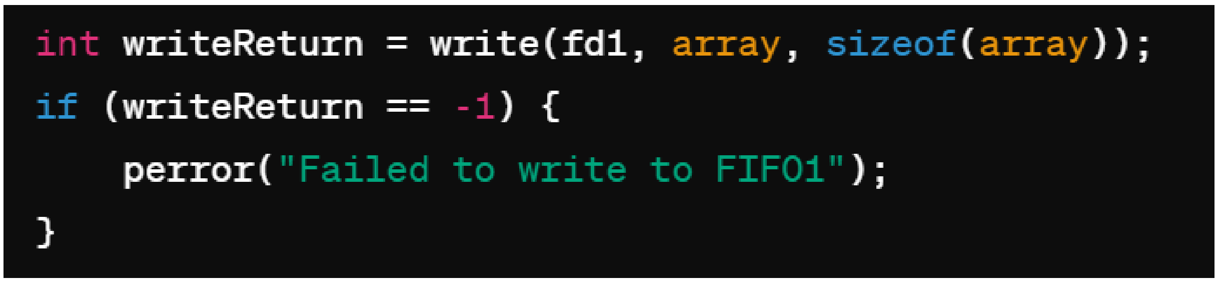
Error management in the code is conducted through meticulous checking of system call return values, and appropriate error responses are provided. Here's the part of the code that showcases this approach:



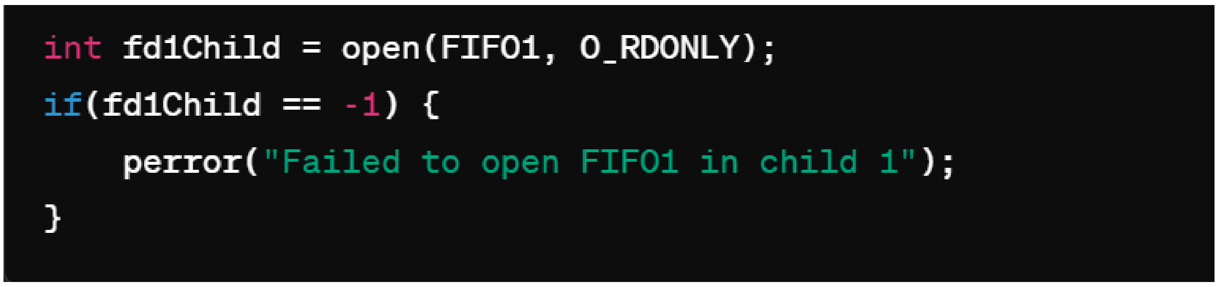
```c
// Error handling function
void handle_error(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}
```



```c
if (mkfifo(FIFO1, 0666) == -1 || mkfifo(FIFO2, 0666) == -1) {
    handle_error("Failed to create FIFOs");
}
```



```c
int writeReturn = write(fd1, array, sizeof(array));
if (writeReturn == -1) {
    perror("Failed to write to FIFO1");
}
```



```c
int fd1Child = open(FIFO1, O_RDONLY);
if(fd1Child == -1) {
    perror("Failed to open FIFO1 in child 1");
}
```

In each of the above examples, the code checks if the system call succeeded. If not, it uses perror() to print the error reason, or calls handle_error() to print the error and exit, ensuring that the program does not continue in an undefined state.

The clear structure of error checking after each critical system call and the use of a designated error handling function ensure that errors are managed effectively and consistently throughout the program. This practice not only helps in debugging during development but also aids in robustness during production runs.