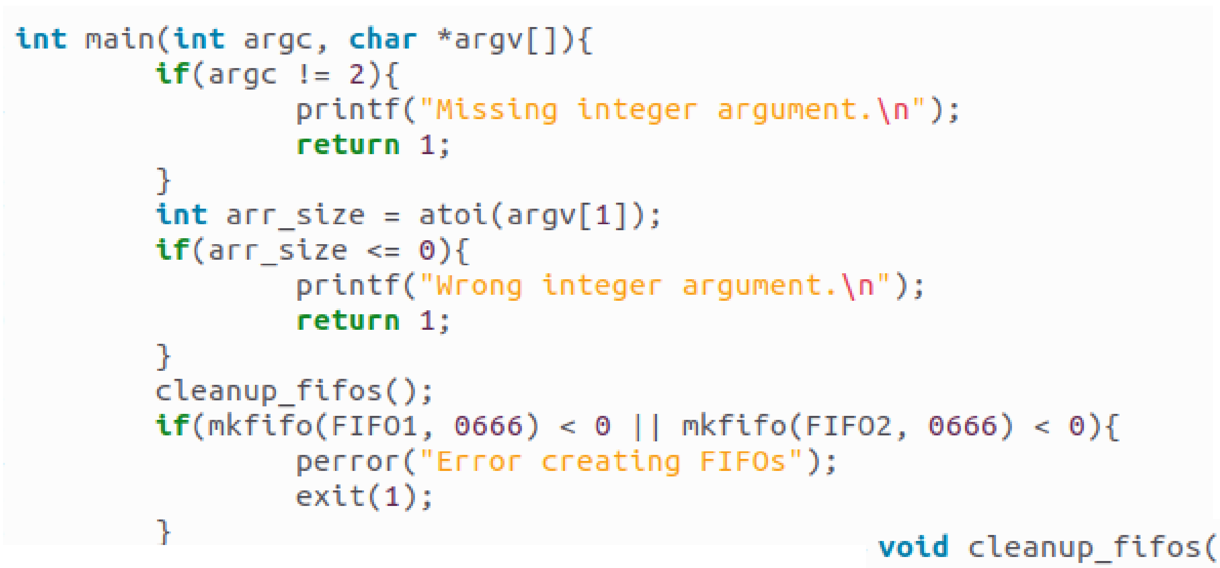
Alper Tavşanoğlu-210104004142-HW-2-Report

The purpose of this program is to implement inter-process communication (IPC) using FIFOs (named pipes). The program is creating two child processes which communicate through FIFOs. The first child is taking generated random numbers according to the user argument, sums them, and sends the result to the second child through a FIFO. The second child also take these numbers multiplies these numbers and adds the result from the first child, displaying the final result.

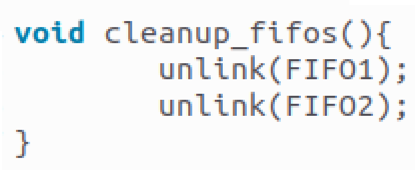
Main steps :

- Parent process must send random numbers to both FIFOs.
- Parent process should send the multiplication process to the second FIFO as the command "multiply".
- Addition should be done using random numbers in the first FIFO and this result should be written in the second FIFO.
- The second FIFO's command it received, must check with a string compare method and perform multiplication with same random numbers.
- The result values of both FIFOs should be added (the total result from the first FIFO and the multiplication result from the second FIFO) and the overall result should be printed on the screen.

First of all, I created a program that takes an integer argument which determines the size of the array of random numbers. And i am creating two FIFOs for inter-process communication. (Before creating the FIFOs using mkfifo, the cleanup_fifos function is called to ensure that any existing FIFOs with the same names are removed.)

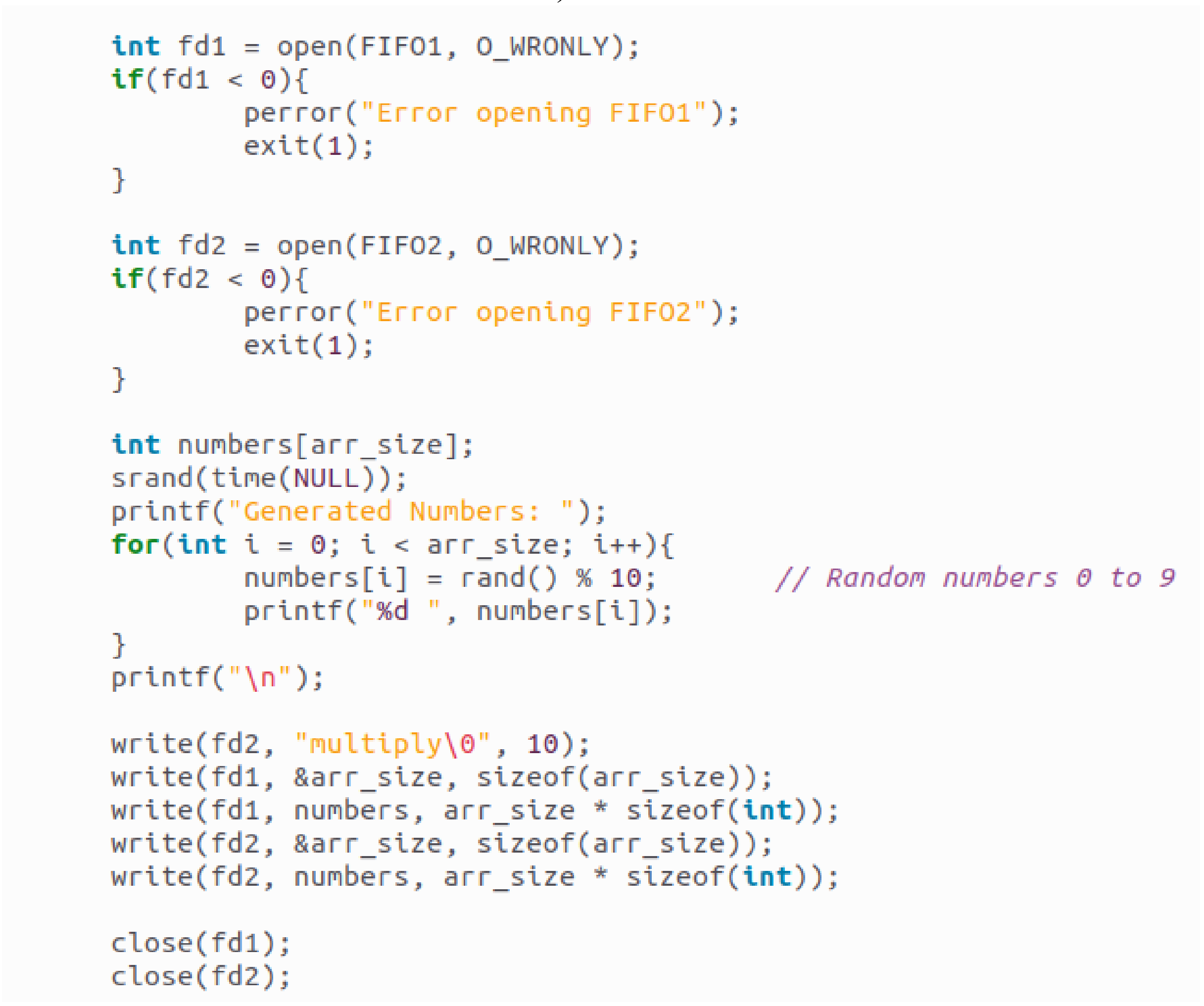


```c
int main(int argc, char *argv[]){
	if(argc != 2){
		printf("Missing integer argument.\n");
		return 1;
	}
	int arr_size = atoi(argv[1]);
	if(arr_size <= 0){
		printf("Wrong integer argument.\n");
		return 1;
	}
	cleanup_fifos();
	if(mkfifo(FIFO1, 0666) < 0 || mkfifo(FIFO2, 0666) < 0){
		perror("Error creating FIFOs");
		exit(1);
	}
```



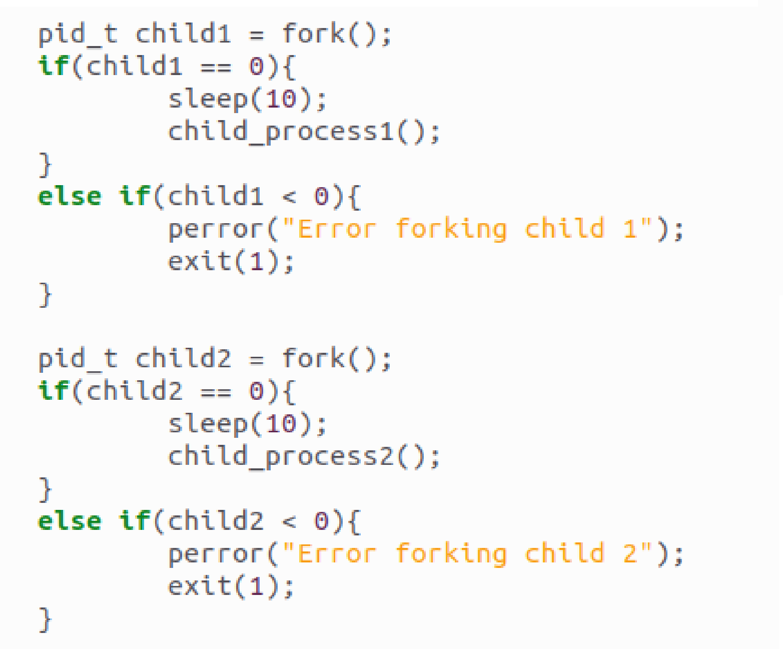
```c
void cleanup_fifos(){
	unlink(FIFO1);
	unlink(FIFO2);
}
```

Then, I sent an array of generated random numbers (which is genetared according to the user input) to the both FIFO and a command (“multiply”) to the second FIFO requesting a multiplication operation. (i am sending random number around 0 to 10)

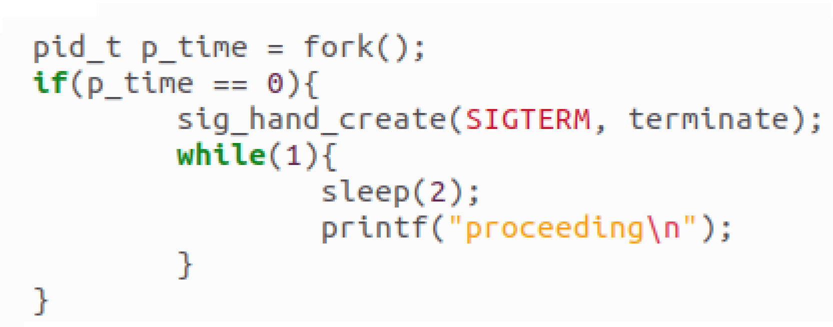


```c
int fd1 = open(FIFO1, O_WRONLY);
if(fd1 < 0){
        perror("Error opening FIFO1");
        exit(1);
}

int fd2 = open(FIFO2, O_WRONLY);
if(fd2 < 0){
        perror("Error opening FIFO2");
        exit(1);
}

int numbers[arr_size];
srand(time(NULL));
printf("Generated Numbers: ");
for(int i = 0; i < arr_size; i++){
        numbers[i] = rand() % 10;      // Random numbers 0 to 9
        printf("%d ", numbers[i]);
}
printf("\n");

write(fd2, "multiply\0", 10);
write(fd1, &arr_size, sizeof(arr_size));
write(fd1, numbers, arr_size * sizeof(int));
write(fd2, &arr_size, sizeof(arr_size));
write(fd2, numbers, arr_size * sizeof(int));

close(fd1);
close(fd2);
```

I used fork() to create two child processes, each handling a different FIFO. All child processes sleep for 10 seconds, execute their tasks, and then exit. In loop, printing a message containing "proceeding" every two seconds.

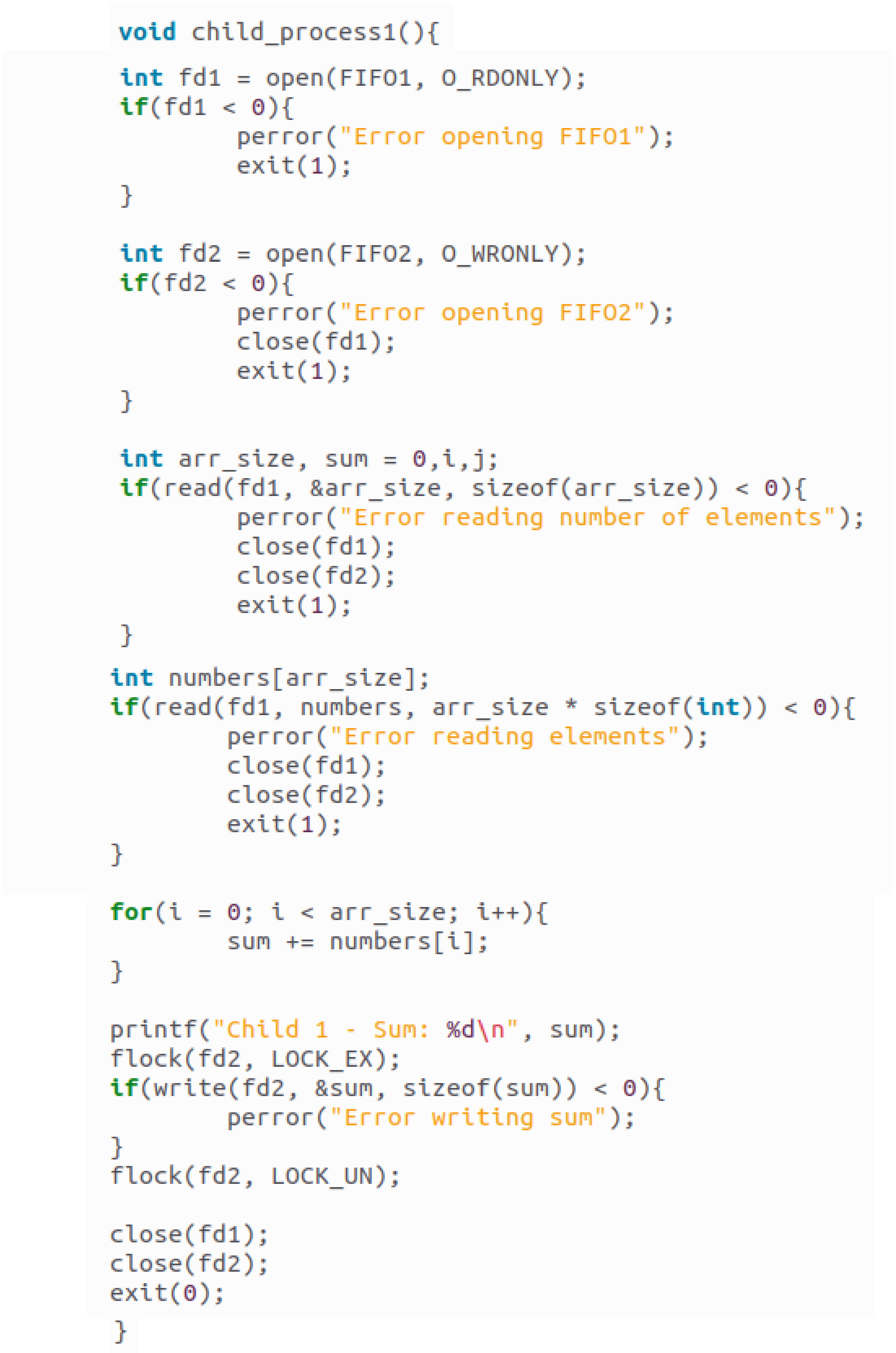


```c
pid_t child1 = fork();
if(child1 == 0){
        sleep(10);
        child_process1();
}
else if(child1 < 0){
        perror("Error forking child 1");
        exit(1);
}

pid_t child2 = fork();
if(child2 == 0){
        sleep(10);
        child_process2();
}
else if(child2 < 0){
        perror("Error forking child 2");
        exit(1);
}
```



```c
pid_t p_time = fork();
if(p_time == 0){
        sig_hand_create(SIGTERM, terminate);
        while(1){
                sleep(2);
                printf("proceeding\n");
        }
}
```

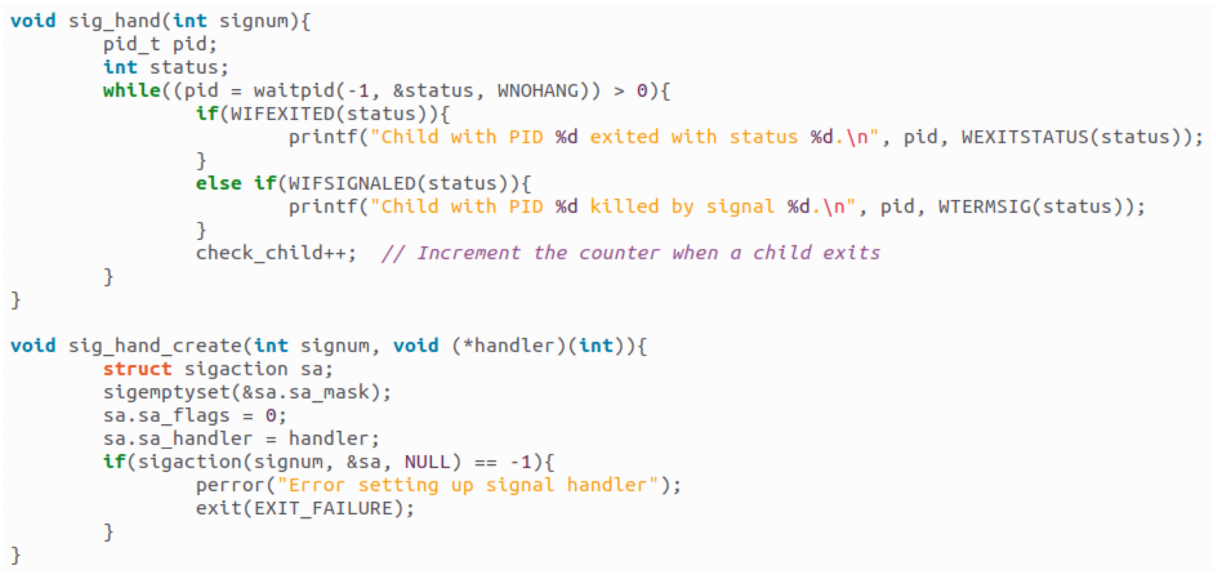
In child1, I opened FIFOs (opened FIFO1 for reading random numbers, opened FIFO2 for writing sum result) and read random numbers from the first FIFO. I calculated their sum and sended the sum to the second FIFO.



```
void child_process1(){
 int fd1 = open(FIFO1, O_RDONLY);
 if(fd1 < 0){
         perror("Error opening FIFO1");
         exit(1);
 }

 int fd2 = open(FIFO2, O_WRONLY);
 if(fd2 < 0){
         perror("Error opening FIFO2");
         close(fd1);
         exit(1);
 }

 int arr_size, sum = 0,i,j;
 if(read(fd1, &arr_size, sizeof(arr_size)) < 0){
         perror("Error reading number of elements");
         close(fd1);
         close(fd2);
         exit(1);
 }
int numbers[arr_size];
if(read(fd1, numbers, arr_size * sizeof(int)) < 0){
        perror("Error reading elements");
        close(fd1);
        close(fd2);
        exit(1);
}

for(i = 0; i < arr_size; i++){
        sum += numbers[i];
}

printf("Child 1 - Sum: %d\n", sum);
flock(fd2, LOCK_EX);
if(write(fd2, &sum, sizeof(sum)) < 0){
        perror("Error writing sum");
}
flock(fd2, LOCK_UN);

close(fd1);
close(fd2);
exit(0);
 }
```

In child2, I opened second FIFO (to read command and random numbers). Then i read the command ,to perform the multiplication operation, and random numbers. The “multiply” command which received from the parent process checked with using string comparison method. After that, child2 performed the multiplication operation and prints the sum of the results of child1 and child2 processes on the screen.

```c
void child_process2(){

int fd2 = open(FIFO2, O_RDONLY);
if(fd2 < 0){
        perror("Error opening FIFO2");
        exit(1);
}

char command[10] = {0};
int arr_size, product = 1, rec_sum,i,j;

if(read(fd2, command, sizeof(command)) < 0){
        perror("Error reading command");
        close(fd2);
        exit(1);
}
if(strcmp(command, "multiply") != 0){
        printf("Child 2 - Command not recognized.\n");
        exit(1);
}
printf("Child 2 - Command: %s\n", command);
read(fd2, &arr_size, sizeof(arr_size));
int numbers[arr_size];
read(fd2, numbers, arr_size * sizeof(int));

for(i = 0; i < arr_size; i++){
        product *= numbers[i];
}

printf("Child 2 - Product: %d\n", product);
read(fd2, &rec_sum, sizeof(rec_sum));
printf("Child 2 - Sum received: %d\n", rec_sum);

int res_all = product + rec_sum;
printf("Final Result: %d\n", res_all);

close(fd2);
exit(0);
}
```



```c
void sig_hand(int signum){
        pid_t pid;
        int status;
        while((pid = waitpid(-1, &status, WNOHANG)) > 0){
                if(WIFEXITED(status)){
                        printf("Child with PID %d exited with status %d.\n", pid, WEXITSTATUS(status));
                }
                else if(WIFSIGNALED(status)){
                        printf("Child with PID %d killed by signal %d.\n", pid, WTERMSIG(status));
                }
                check_child++;  // Increment the counter when a child exits
        }
}

void sig_hand_create(int signum, void (*handler)(int)){
        struct sigaction sa;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        sa.sa_handler = handler;
        if(sigaction(signum, &sa, NULL) == -1){
                perror("Error setting up signal handler");
                exit(EXIT_FAILURE);
        }
}
```

I implemented a signal handler to manage child process terminations and ensured no zombie processes are left. I entered a loop that prints "proceeding" every two seconds until all child processes have terminated.

Signal Handling: I used **sigaction** to handle **SIGCHLD** signals, ensuring that the exit status of child processes is captured and that no zombie processes remain.

Error Handling: I handled potential errors such as failed FIFO creation, file opening failures, and read/write errors robustly. All errors are reported using **perror().**

In the **sigchld_handler** function , program prints the exit status of each child process as it terminates. The exit status is captured using the **waitpid()** function, and different scenarios (normal exit vs. exit by signal) are handled distinctly.

waitpid(-1, &status, WNOHANG): I used this call to reaps **zombie children** (if any). The -1 argument makes **waitpid()** wait for any child process. The **WNOHANG** option specifies that **waitpid()** should return immediately instead of waiting, if no child has exited.

WIFEXITED and WIFSIGNALED: These macros check how the child exited normally or due to a signal. This helps in determining the exact nature of the termination.

Print Statements: These provide feedback about each child's exit status, whether they exited normally or were stopped by a signal, and are essential for debugging and ensuring the program runs as expected.

WEXITSTATUS(status): This macro retrieves the exit status of the child process if **WIFEXITED** is non-zero, showing that the child exited normally.

WTERMSIG(status): This macro retrieves the terminating signal of the child process if **WIFSIGNALED** is non-zero, indicating the process was terminated by a signal.

Compile and Examples

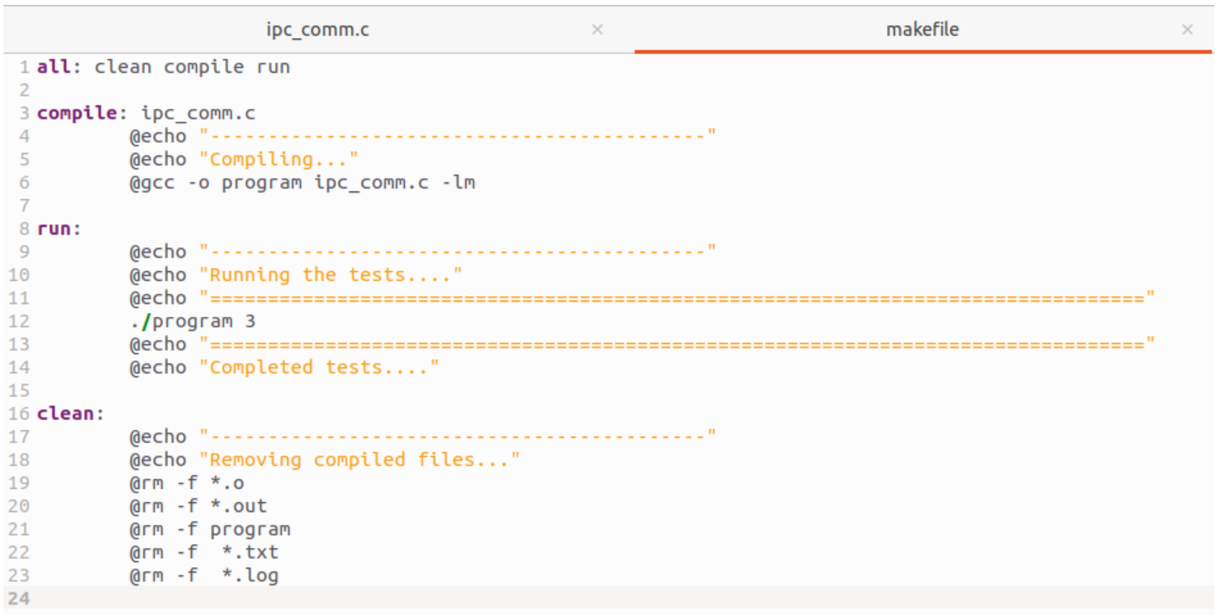
First Option

For running the code we can use makefile directly. With **make** command program runs automaticly with 3 as a user integer argument. If we want to try with another number we can change 3 (**in makefile line 12**) with another number what we want. (If we used **make** command at least one time, we can also use **./program “number”** command also).

Second Option

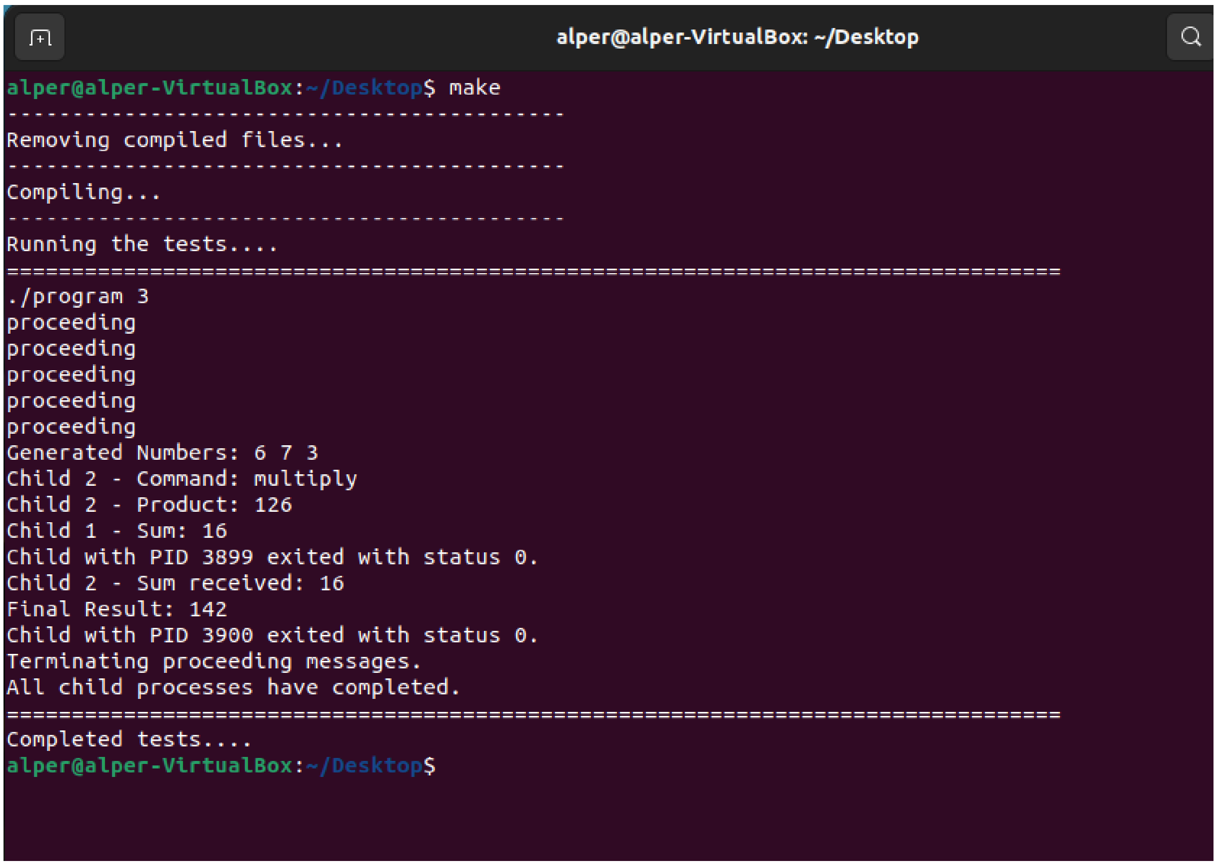
Or without using makefile we can compile with these commads first **gcc ipc_comm.c** and **./a.out 4** (4 as example we can use any number).

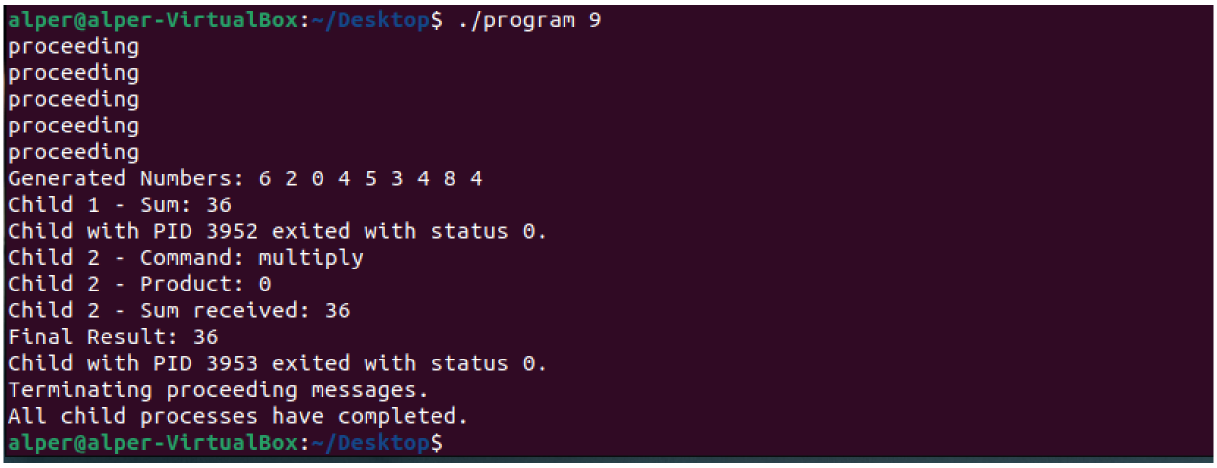
Makefile also have **makeclean** command which removes everything except **c code** and **makefile**.

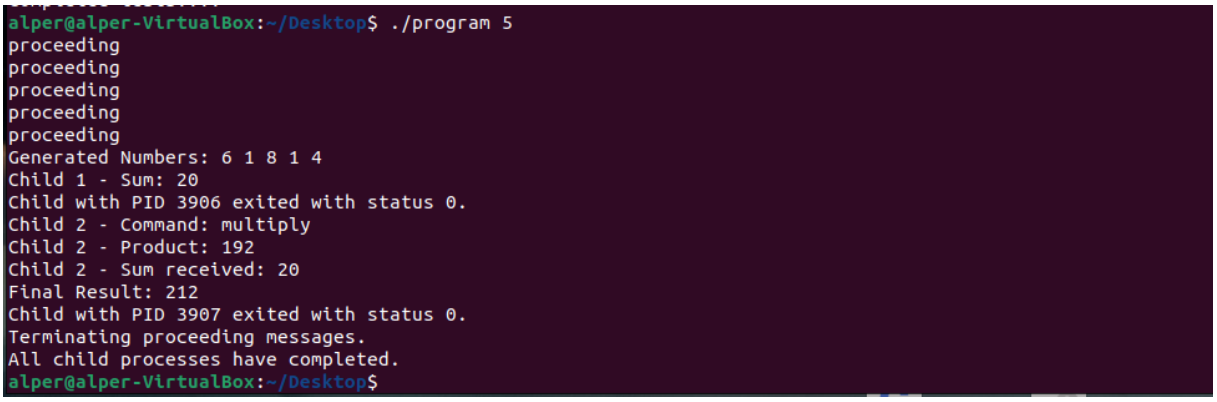


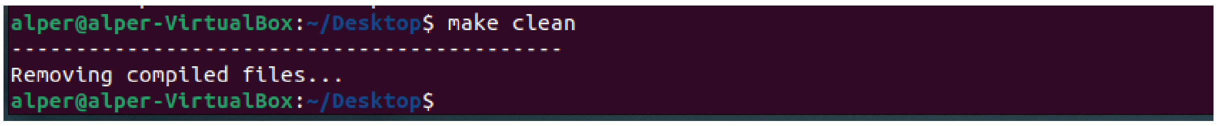
```
1  all: clean compile run
2  
3  compile: ipc_comm.c
4          @echo "-------------------------------------------"
5          @echo "Compiling..."
6          @gcc -o program ipc_comm.c -lm
7  
8  run:
9          @echo "-------------------------------------------"
10         @echo "Running the tests...."
11         @echo "================================================================================="
12         ./program 3
13         @echo "================================================================================="
14         @echo "Completed tests...."
15 
16 clean:
17         @echo "-------------------------------------------"
18         @echo "Removing compiled files..."
19         @rm -f *.o
20         @rm -f *.out
21         @rm -f program
22         @rm -f  *.txt
23         @rm -f  *.log
24 
```

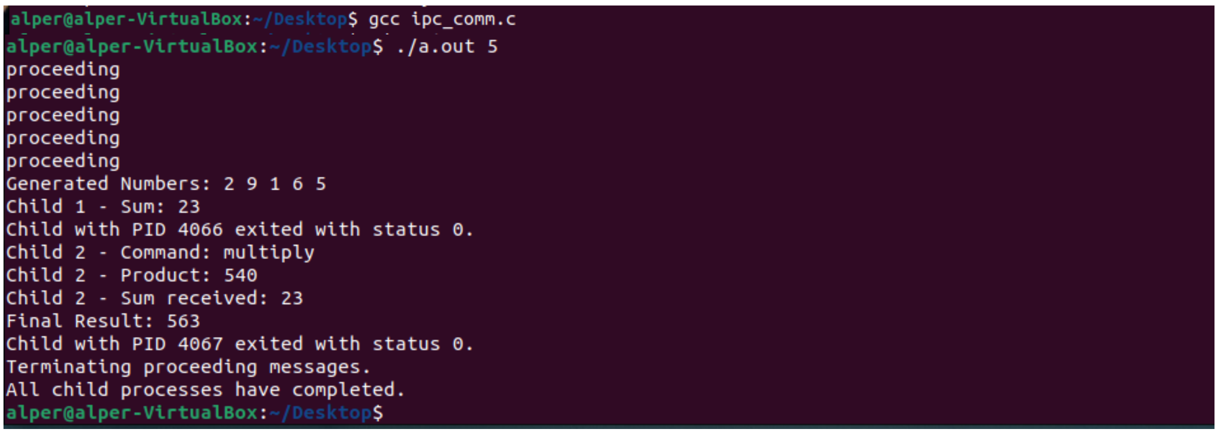
Examples on next page


```
alper@alper-VirtualBox: ~/Desktop
alper@alper-VirtualBox:~/Desktop$ make
------------------------------------------
Removing compiled files...
------------------------------------------
Compiling...
------------------------------------------
Running the tests....
================================================================================
./program 3
proceeding
proceeding
proceeding
proceeding
proceeding
Generated Numbers: 6 7 3
Child 2 - Command: multiply
Child 2 - Product: 126
Child 1 - Sum: 16
Child with PID 3899 exited with status 0.
Child 2 - Sum received: 16
Final Result: 142
Child with PID 3900 exited with status 0.
Terminating proceeding messages.
All child processes have completed.
================================================================================
Completed tests....
alper@alper-VirtualBox:~/Desktop$
```

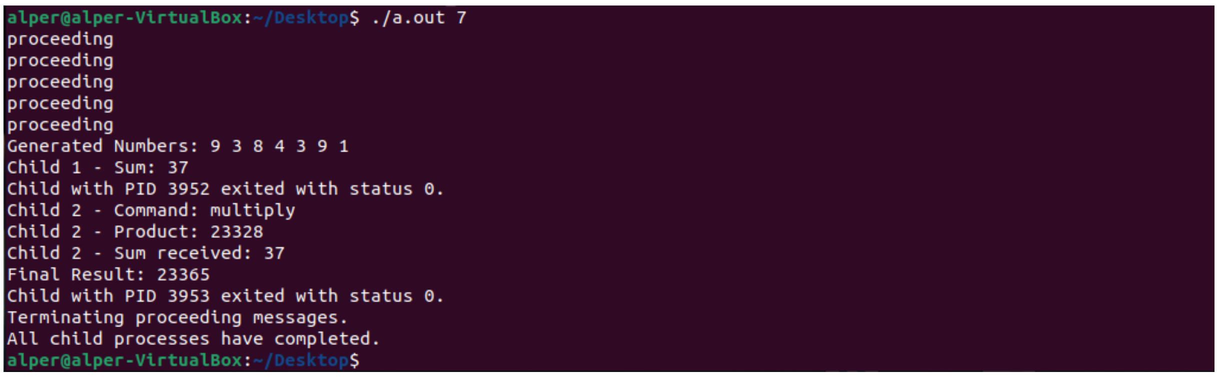

```
alper@alper-VirtualBox:~/Desktop$ ./program 9
proceeding
proceeding
proceeding
proceeding
proceeding
Generated Numbers: 6 2 0 4 5 3 4 8 4
Child 1 - Sum: 36
Child with PID 3952 exited with status 0.
Child 2 - Command: multiply
Child 2 - Product: 0
Child 2 - Sum received: 36
Final Result: 36
Child with PID 3953 exited with status 0.
Terminating proceeding messages.
All child processes have completed.
alper@alper-VirtualBox:~/Desktop$
```


```
alper@alper-VirtualBox:~/Desktop$ ./program 5
proceeding
proceeding
proceeding
proceeding
proceeding
Generated Numbers: 6 1 8 1 4
Child 1 - Sum: 20
Child with PID 3906 exited with status 0.
Child 2 - Command: multiply
Child 2 - Product: 192
Child 2 - Sum received: 20
Final Result: 212
Child with PID 3907 exited with status 0.
Terminating proceeding messages.
All child processes have completed.
alper@alper-VirtualBox:~/Desktop$
```

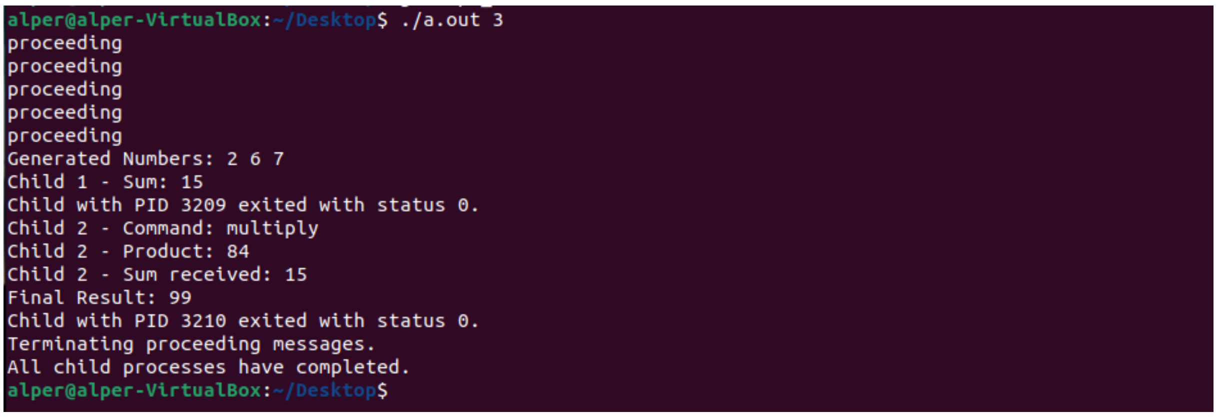

```
alper@alper-VirtualBox:~/Desktop$ make clean
------------------------------------------
Removing compiled files...
alper@alper-VirtualBox:~/Desktop$
```



```
alper@alper-VirtualBox:~/Desktop$ gcc ipc_comm.c
alper@alper-VirtualBox:~/Desktop$ ./a.out 5
proceeding
proceeding
proceeding
proceeding
proceeding
Generated Numbers: 2 9 1 6 5
Child 1 - Sum: 23
Child with PID 4066 exited with status 0.
Child 2 - Command: multiply
Child 2 - Product: 540
Child 2 - Sum received: 23
Final Result: 563
Child with PID 4067 exited with status 0.
Terminating proceeding messages.
All child processes have completed.
alper@alper-VirtualBox:~/Desktop$
```

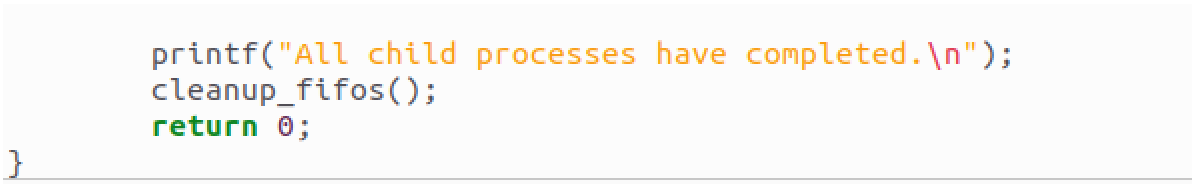


```
alper@alper-VirtualBox:~/Desktop$ ./a.out 7
proceeding
proceeding
proceeding
proceeding
proceeding
Generated Numbers: 9 3 8 4 3 9 1
Child 1 - Sum: 37
Child with PID 3952 exited with status 0.
Child 2 - Command: multiply
Child 2 - Product: 23328
Child 2 - Sum received: 37
Final Result: 23365
Child with PID 3953 exited with status 0.
Terminating proceeding messages.
All child processes have completed.
alper@alper-VirtualBox:~/Desktop$
```

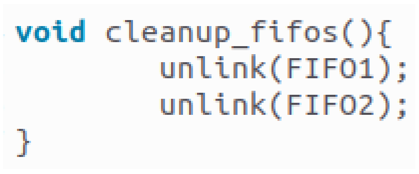


```
alper@alper-VirtualBox:~/Desktop$ ./a.out 3
proceeding
proceeding
proceeding
proceeding
proceeding
Generated Numbers: 2 6 7
Child 1 - Sum: 15
Child with PID 3209 exited with status 0.
Child 2 - Command: multiply
Child 2 - Product: 84
Child 2 - Sum received: 15
Final Result: 99
Child with PID 3210 exited with status 0.
Terminating proceeding messages.
All child processes have completed.
alper@alper-VirtualBox:~/Desktop$
```

After comile code FIFO files removing automaticly with unlink.



```
        printf("All child processes have completed.\n");
        cleanup_fifos();
        return 0;
}
```

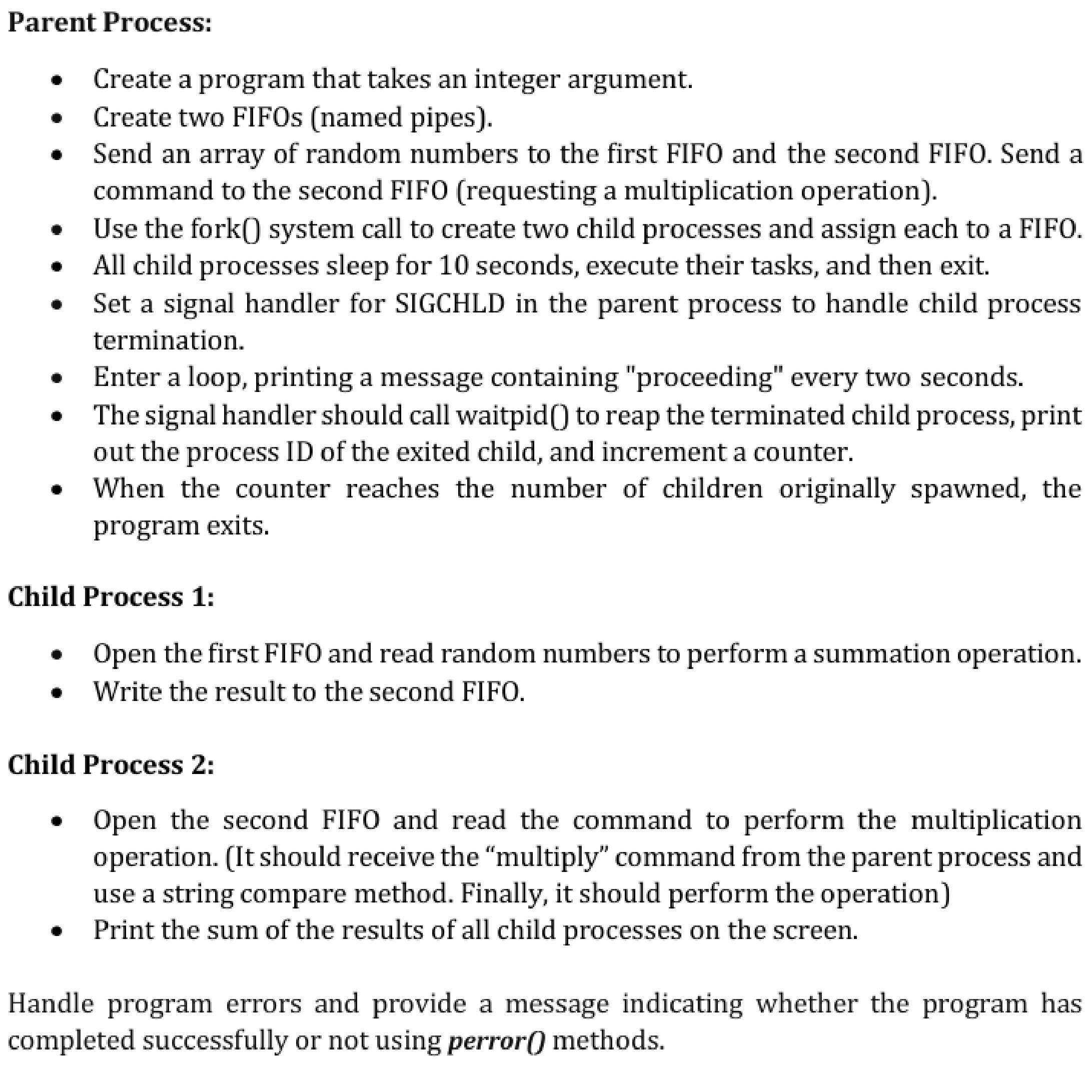


```
void cleanup_fifos(){
        unlink(FIFO1);
        unlink(FIFO2);
}
```

Final notes and Conclusion

I ensured the program compiled without errors using GCC.
With an input argument of user. The program generated random numbers, calculated their sum and product, and displayed the final result as expected.
I confirmed that all file descriptors are closed appropriately. No memory leaks or resource leaks were observed. I used the valgrind tool to confirm that all allocated memory was freed.
The program handled the termination of child processes correctly, with the parent process exiting cleanly after all children had finished.
The program handled all specified error conditions gracefully.

Program performed these steps:



Parent Process:

- Create a program that takes an integer argument.
- Create two FIFOs (named pipes).
- Send an array of random numbers to the first FIFO and the second FIFO. Send a command to the second FIFO (requesting a multiplication operation).
- Use the fork() system call to create two child processes and assign each to a FIFO.
- All child processes sleep for 10 seconds, execute their tasks, and then exit.
- Set a signal handler for SIGCHLD in the parent process to handle child process termination.
- Enter a loop, printing a message containing "proceeding" every two seconds.
- The signal handler should call waitpid() to reap the terminated child process, print out the process ID of the exited child, and increment a counter.
- When the counter reaches the number of children originally spawned, the program exits.

Child Process 1:

- Open the first FIFO and read random numbers to perform a summation operation.
- Write the result to the second FIFO.

Child Process 2:

- Open the second FIFO and read the command to perform the multiplication operation. (It should receive the "multiply" command from the parent process and use a string compare method. Finally, it should perform the operation)
- Print the sum of the results of all child processes on the screen.

Handle program errors and provide a message indicating whether the program has completed successfully or not using ***perror()*** methods.