Gebze Technical University

CSE344 System Programming

Homework-2

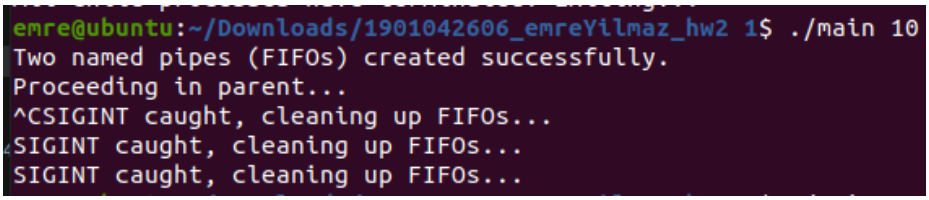
2023-24

Lecturer: Erkan Zergeroglu

Emre YILMAZ 1901042606

1 Notes

1. I only test the command "mult" as T.A. said in the Teams channel.
2. I put a signal handler for ctrl-c `SIGINT` signal. If program catches that signal, it cleans fifos.
3. In parent process, when I encounter an error, I cleaned FIFOs before exiting.
4. You can see the processes' exit status when program terminates. Bonus Parts 1-2 are done.



SIGINT signal handling example.

2 How to Run

Firstly compile the project using command `make`
After that type the command `./main [size of random number array]`

2.1 Example of How to Run

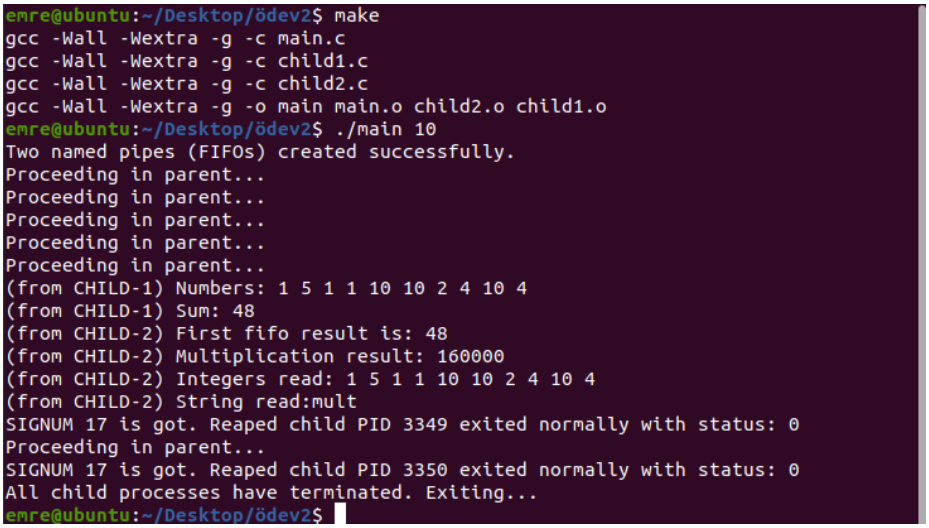


Fig-0: How to Run

3 Parent Process

3.1 Why Did I Open FIFOs After Forking

Firstly, I have to explain the most important part of the parent process. In the project, the standard procedure for opening FIFOs involves a decision point regarding when to open them—before or after forking. Opening FIFOs before forking often leads to blocking issues, where the FIFO waits indefinitely for the other end to open for writing or reading, stalling the program before it can proceed to fork. This issue necessitates opening the FIFO in read-write `(RDWR)` mode to avoid blocking; however, this mode is not typically recommended in FIFO communication scenarios due to potential issues such as confusing read/write behavior and possible deadlocks.

Given the inherent risks and impracticalities associated with `(RDWR)` mode—such as potential data handling errors and synchronization complexities—it is more robust and safer to open FIFOs after forking. This approach allows each process (parent and child) to independently open and manage their end of the FIFO as needed (either for reading or writing). This method ensures clearer and safer communication channels, as each forked process handles its specific task (either reading or writing), which significantly mitigates the risks of deadlocks and data mismanagement.

Therefore, I have opted to open the FIFOs after forking to ensure reliable and safe operation of the program, aligning with best practices for IPC (Inter-Process Communication) using FIFOs. This decision is intended to enhance the stability and correctness of the program under various operating conditions.

First, we take the command line argument to determine the number of random numbers to generate. We store this in a local integer variable.

Then, an algorithm for the signal handler is implemented. I would like to describe this under a subsection named "Signal Handler Algorithm".

3.2 Signal Handler Algorithm

```c
int child_counter = 0;
const int NUMBER_OF_CHILDREN = 2; // Total number of child processes

void sigchld_handler(int signum) {
    if (signum == SIGCHLD) {
        pid_t pid;
        int status;

        while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
            printf("SIGCHLD received. Reaped child PID %d exited normally with status: %d\n", pid, WEXITSTATUS(status));
            child_counter++;
        }

        if (child_counter >= NUMBER_OF_CHILDREN) {
            printf("All child processes have terminated. Exiting...\n");
            unlink("/tmp/eyhw1_fifo1");
            unlink("/tmp/eyhw1_fifo2");
            exit(0);
        }
    } else if (signum == SIGINT) {
        printf("SIGINT caught, cleaning up FIFOs...\n");
        unlink("/tmp/eyhw1_fifo1");
        unlink("/tmp/eyhw1_fifo2");
        exit(0);
    }
}
```

Fig-1: Signal Handler

The code defines a signal handler, `sigchld_handler`, which is triggered whenever a child process terminates. The handler uses waitpid with the `WNOHANG` option to non-blockingly reap any terminated child processes, effectively preventing them from becoming zombies. The handler also keeps track of the number of child processes that have been reaped via the `child_counter` global variable.

Detailed Explanation:

- Signal Handler (`sigchld_handler`): This function is called when a `SIGCHLD` signal is received, indicating that a child process has changed its state (e.g., terminated). Within the handler, waitpid is iteratively called to reap all available terminated child processes without blocking, ensuring that the system efficiently cleans up child resources. The handler checks if the number of terminated children has reached a pre-defined threshold (`NUMBER_OF_CHILDREN`), and if so, performs necessary cleanup actions and exits the main process.
- Setting up the Signal Handler (`setup_signal_handler`): This function configures the system to use `sigchld_handler` when `SIGCHLD` signals are received. It initializes the signal mask to block no additional signals during the execution of the handler.

- Cleanup and Exit: Upon confirming that all child processes are reaped, the handler removes the FIFOs used for inter-process communication and terminates the parent process to prevent resource leakage and ensure that the program does not leave orphan processes or unneeded files (fifo1 and fifo2) on the system.
- If a Ctrl-C SIGINT signal is received, as seen in the implementation, the FIFOs are cleaned up and then the program terminates.

3.3 Preventing Zombies (Bonus Part-1)

Zombie Processes Definition: A zombie process is created when a child process has completed its execution, but its parent has not yet called `wait()` or `waitpid()` to read the child's exit status. The process remains in the operating system's process table as a "zombie," consuming resources unnecessarily.

Signal Handler (`sigchld_handler`): This function is triggered by the `SIGCHLD` signal, which the operating system sends to the parent process whenever one of its children terminates. The handler uses the `waitpid` function with the `WNOHANG` option in a loop to reap all terminated child processes:

- Non-blocking Call: The `WNOHANG` option ensures that the call to `waitpid` does not block if no child has exited, making it efficient and responsive.
- Looping through All Terminated Children: By looping until `waitpid` returns 0, the handler ensures it reaps all child processes that have terminated since the last `SIGCHLD` signal. This is crucial because multiple child terminations can generate a single `SIGCHLD` due to the way signals are coalesced.
- Effective Reaping: Each call to `waitpid` in this context checks for a terminated child, retrieves its exit status, and removes it from the system's process table, thereby preventing it from becoming a zombie. If a child process exits while the parent process is busy (e.g., in a sleep or blocked state), the `SIGCHLD` signal interrupts the parent process and invokes the signal handler, which then cleans up the child process.

Continue to General Flow of Parent Process:
After successfully implementing the signal handler and zombie protection methods, the program's general flow continues. Firstly, two necessary FIFO files are created. Then, the parent process forks to create the first child process. Immediately after, another fork is executed to create the second child process. These child processes open the FIFO files in read mode and, as mentioned in the PDF, they sleep for 10 seconds before starting execution to ensure synchronization.

Afterwards, the parent process opens the FIFO files in write mode and writes the required data to them, generating random numbers based on the command line argument and writing to both FIFOs. Additionally, it sends the string "mult" as a command to the second FIFO. Then, it closes the FIFOs and, as long as the child processes have not terminated, it prints the string "Proceeding" every 2 seconds, as indicated in the PDF.

4 Why Did I Put Sleep Just Before Execution of Child Processes and Just After Opening FIFOs?

If we put a sleep before opening the FIFOs in the child processes, the parent process will block because the FIFOs are not open, causing all three processes to sleep simultaneously. Consequently, the parent process won't finish all write operations before the child processes start execution, leading to various race conditions. Since semaphore and mutex methods for synchronization have not been covered in the course yet, handling these parts would be extremely challenging. Therefore, when the child processes open the FIFOs before sleeping, they can successfully read and write simultaneously without getting stuck in race conditions after the parent process finishes its write operations.

The 10-second sleep is already there to ensure that the parent process can complete its write operations before the child processes start.

5 First Child Process

```c
else if (child1_pid == 0)
{
    // Open the second named pipe for writing
    int fd2 = open("/tmp/eyhw1_fifo2", O_WRONLY);
    if (fd2 == -1) {
        perror("Error opening fifo2 for writing");
        exit(1); // Exiting because this is a fatal error for this child process
    }
    // Open the first named pipe for reading
    int fd1 = open("/tmp/eyhw1_fifo1", O_RDONLY);
    if (fd1 == -1) {
        perror("Error opening fifo1 for reading");
        exit(1); // Exiting because this is a fatal error for this child process
    }
    // Child process 1 logic
    sleep(10);
    child1_function(fd1, inputNumber, fd2);
}
```

Fig-2: Child-1 Starts

In the implementation of Child-1, as mentioned before, FIFO-1 is first opened in read mode and FIFO-2 in write mode. Then, a 10-second sleep is initiated before the child execution begins.

Initially, dynamic memory allocation is performed for an array based on the command-line argument received from the terminal. Subsequently, FIFO-1 is read, and the generated random numbers are stored in the dynamic array. The populated array is then printed.

Next, the numbers in the array are summed up, resulting in the `"sum"` value. After successfully writing this sum value to FIFO-2, both FIFOs are closed, and the child process is terminated with an exit call.

Note: Errors are handled effectively in this process as well, and in case of any potential issues, memory is freed, FIFOs are closed, and the process is terminated with an exit command. Ensuring memory is freed even in error-free scenarios has been taken care of.

6 Second Child Process

```
} else if (child2_pid == 0) {
    // Child process 2 logic
    int fd2 = open("/tmp/eyhw1_fifo2", O_RDONLY);
    if (fd2 == -1) {
        perror("Error opening fifo2 for reading");
        exit(1); // Exiting because this is a fatal error for this child process
    }
    sleep(10);
    child2_function(fd2, inputNumber);
}
```

Fig-3: Child-2 Starts

In the implementation of Child-2, as mentioned before, FIFO-2 is first opened in read mode. Then, a 10-second sleep is initiated before the child execution begins.

Initially, necessary local and dynamic variables are defined. The maximum size of the string command is set to 1024 bytes, and a fixed-size array named `str` is created for the string variable. A dynamic integer array named `numbers` is created with a size equal to the number specified in the command line argument. `numInts` is a variable used to count the number of numbers read, and `tempInt` is created to store each number read. The `bytesRead` variable is used to keep track of the number of bytes read with the read function.

After variables are initialized, the command sent from the FIFO is read as a string. Reading is done byte by byte until encountering the null character `'0'`. Immediately after, the specified number of random numbers is read from the input and stored in the array created, filling it with the random numbers from FIFO-2. Finally, the summation result sent by Child-1 to FIFO2 is also read and stored in the `tempInt` variable.

After reading operations are completed, the string value is checked, and if the `"mult"` command is successfully received, the random numbers in the array are multiplied together to obtain the result. Similarly, `"sum"`, `"sub"`, and `"div"` commands perform the necessary mathematical calculations to print the result.

Lastly, the random numbers written by the parent are printed by Child-2, the read string is printed, and the process is terminated with an exit call.

Note: Errors are handled effectively in this process as well, and in case of any potential issues, memory is freed, FIFOs are closed, and the process is terminated with an exit command. Ensuring memory is freed even in error-free scenarios has been taken care of.

7 Screenshots and Results

It is successfully observed that synchronization remains intact despite an increase in the number of inputs. However, due to the increase in the result of multiplication, overflow occurs, making the multiplication result unclear. Nevertheless, the program can still run successfully with large numbers.

Additionally, in some cases, you may observe the "proceeding" printing operation in the parent process while the child processes are running.

```
emre@ubuntu:~/Desktop/ödev2$ ./main 8
Two named pipes (FIFOs) created successfully.
Proceeding in parent...
Proceeding in parent...
Proceeding in parent...
Proceeding in parent...
Proceeding in parent...
(from CHILD-1) Numbers: 2 8 4 4 5 6 8 4
(from CHILD-1) Sum: 41
(from CHILD-2) First fifo result is: 41
(from CHILD-2) Multiplication result: 245760
(from CHILD-2) Integers read: 2 8 4 4 5 6 8 4
(from CHILD-2) String read:mult
SIGNUM 17 is got. Reaped child PID 5993 exited normally with status: 0
Proceeding in parent...
SIGNUM 17 is got. Reaped child PID 5994 exited normally with status: 0
All child processes have terminated. Exiting...
```

Fig-4: Result with 8 numbers

```
emre@ubuntu:~/Desktop/ödev2$ ./main 10
Two named pipes (FIFOs) created successfully.
Proceeding in parent...
Proceeding in parent...
Proceeding in parent...
Proceeding in parent...
Proceeding in parent...
(from CHILD-1) Numbers: 4 7 8 6 4 6 7 3 10 2
(from CHILD-1) Sum: 57
(from CHILD-2) First fifo result is: 57
(from CHILD-2) Multiplication result: 13547520
(from CHILD-2) Integers read: 4 7 8 6 4 6 7 3 10 2
(from CHILD-2) String read:mult
SIGNUM 17 is got. Reaped child PID 5812 exited normally with status: 0
SIGNUM 17 is got. Reaped child PID 5813 exited normally with status: 0
All child processes have terminated. Exiting...
```

Fig-5: Result with 10 numbers

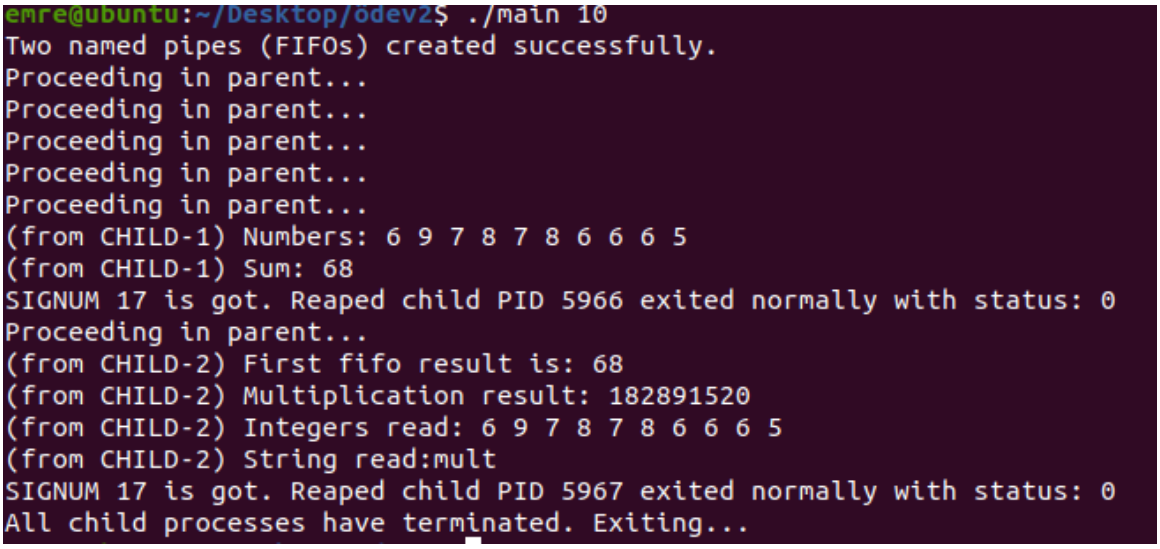


Fig-6: Result with 10 numbers

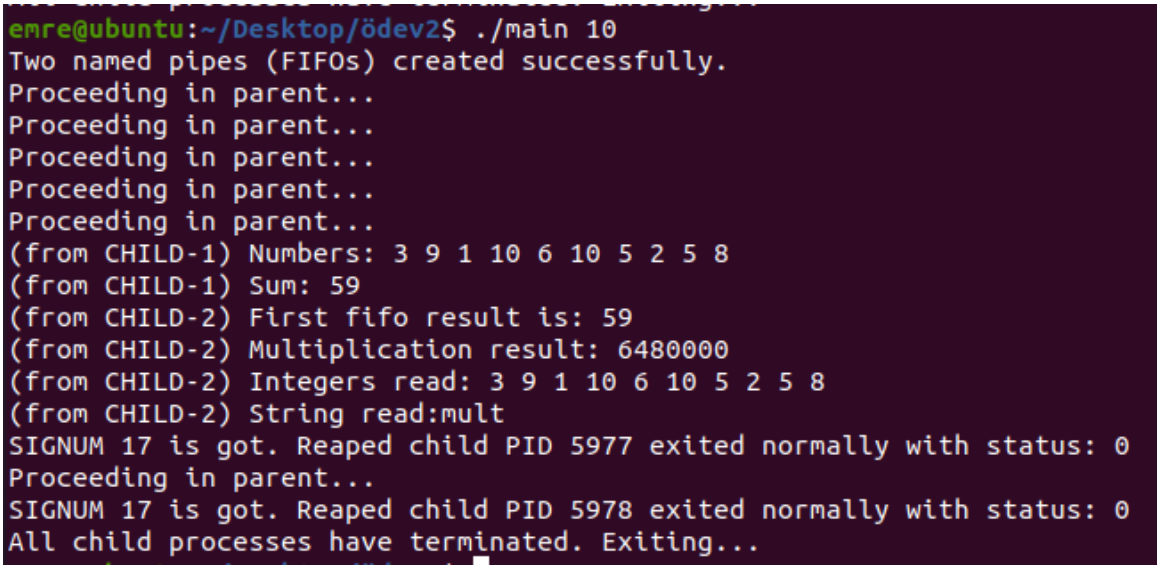


Fig-7: Result with 10 numbers

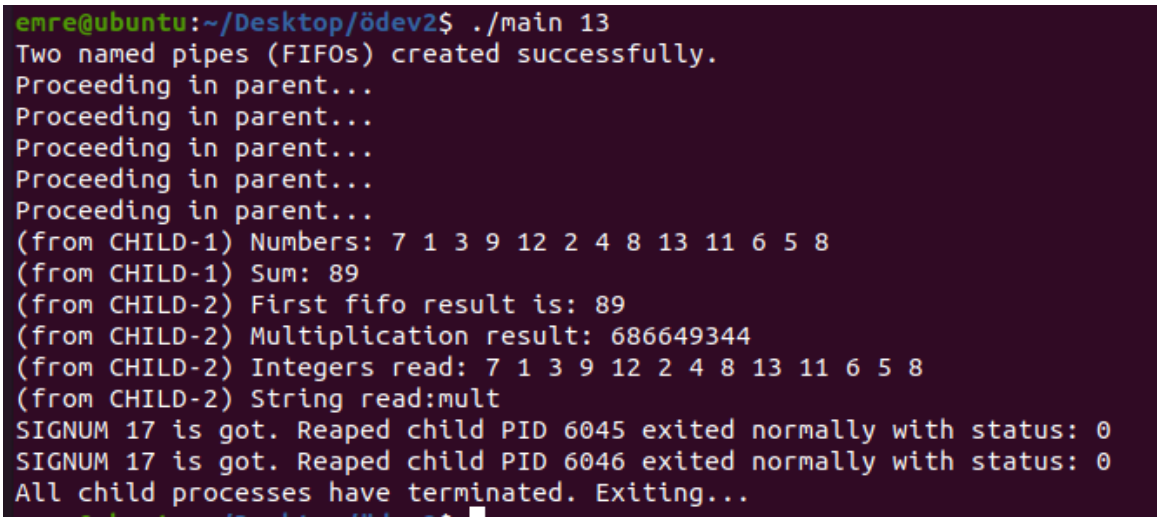


Fig-8: Result with 13 numbers

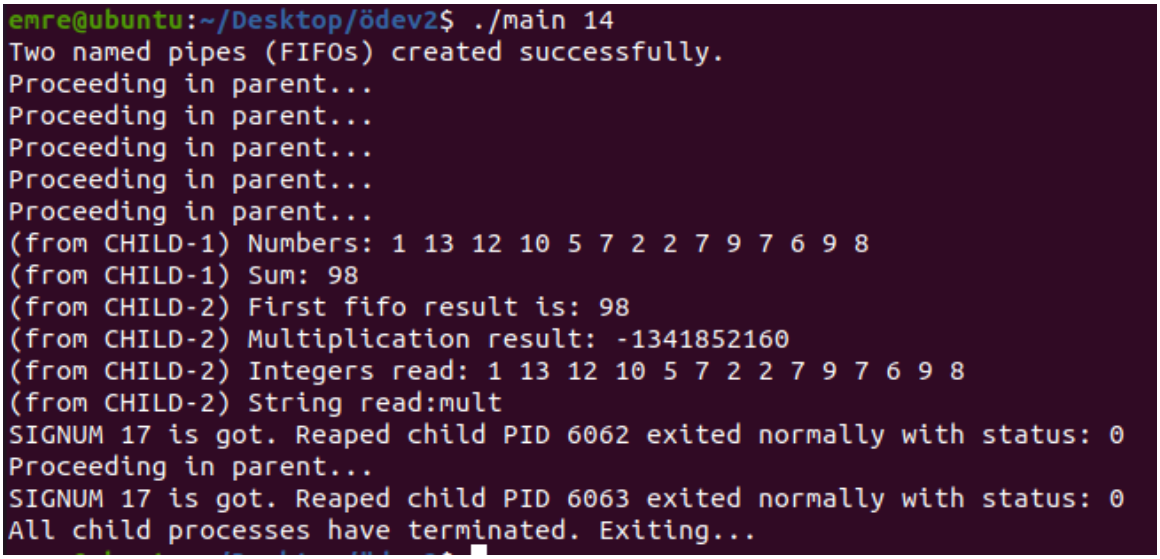


Fig-9: Result with 14 numbers

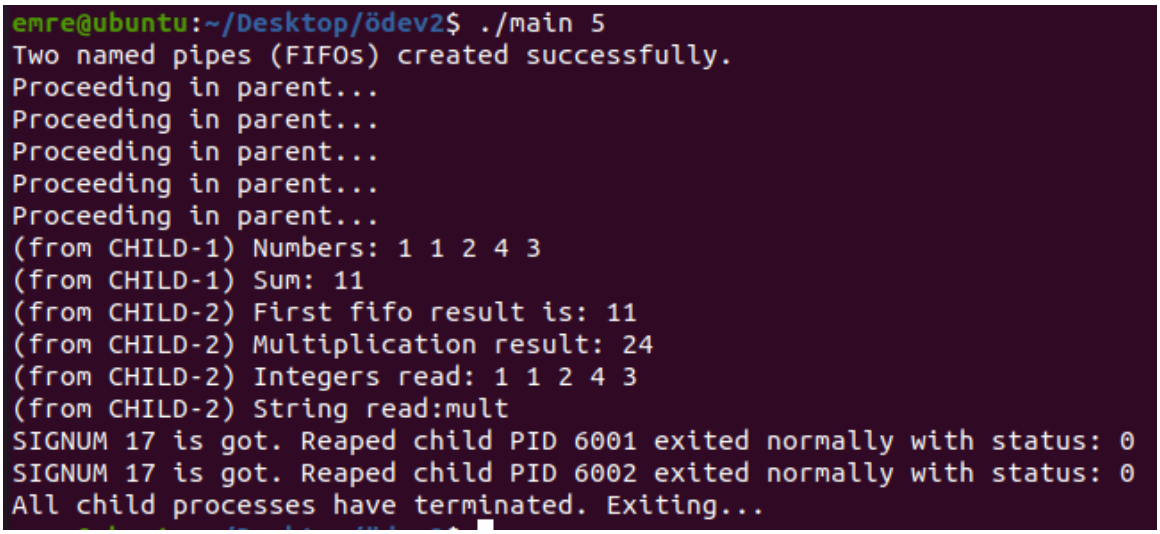


Fig-10: Result with 5 numbers

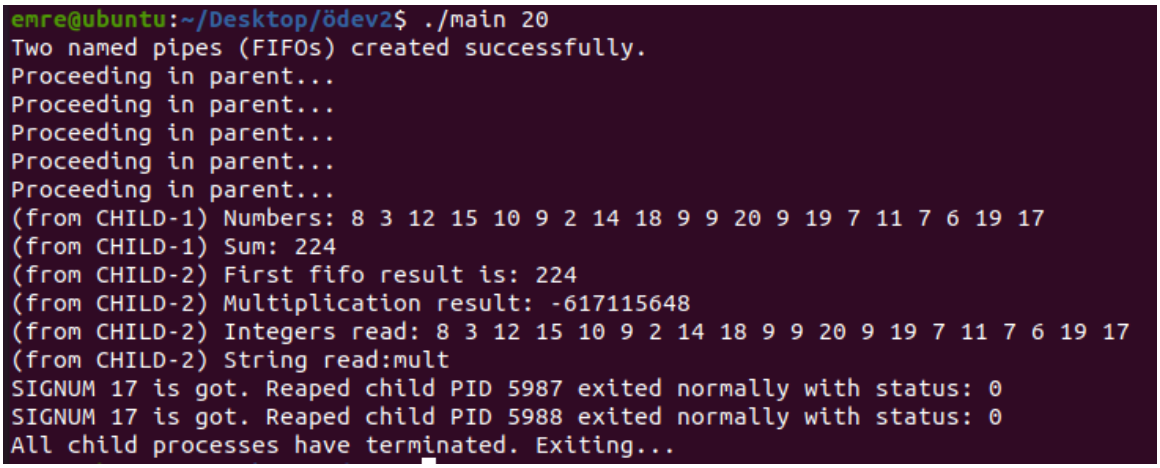


Fig-11: Result with 20 numbers

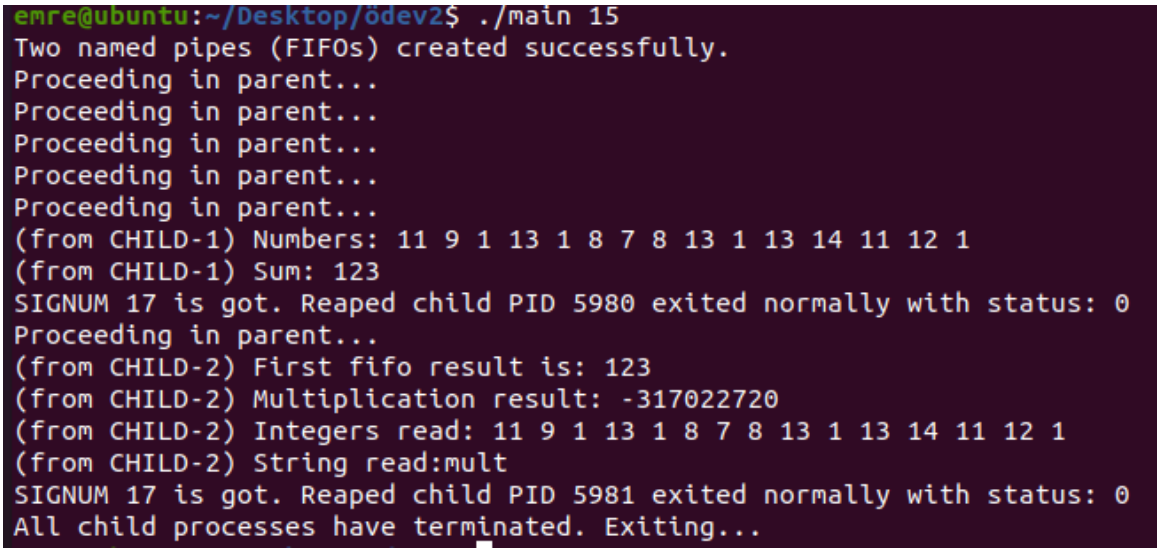


Fig-12: Result with 15 numbers

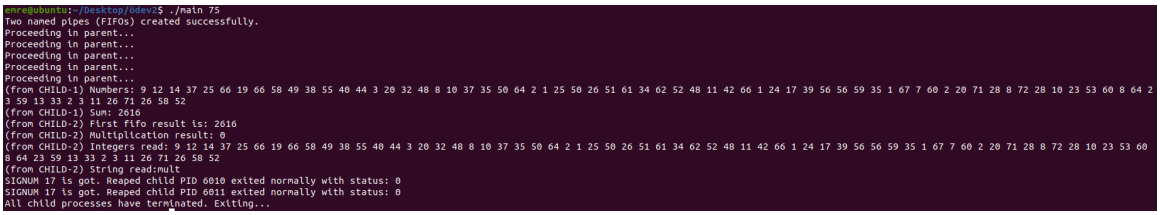


Fig-13: Result with 75 numbers