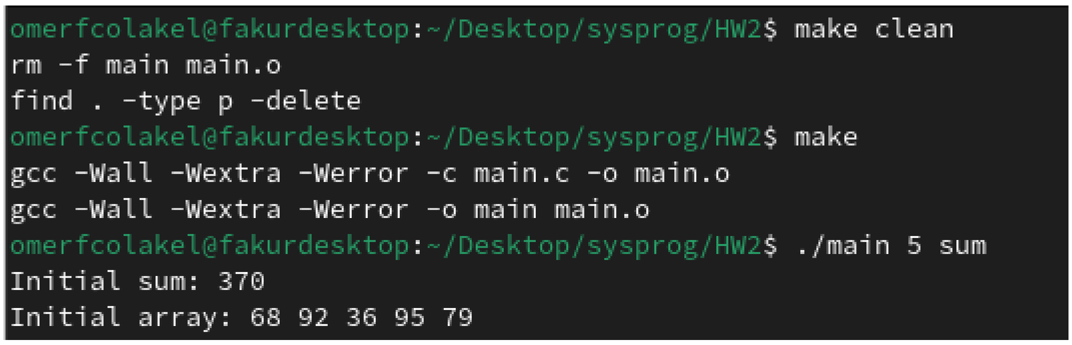
CSE344 Systems Programming

Homework 2 Report

Ömer Faruk Çolakel

200104004043

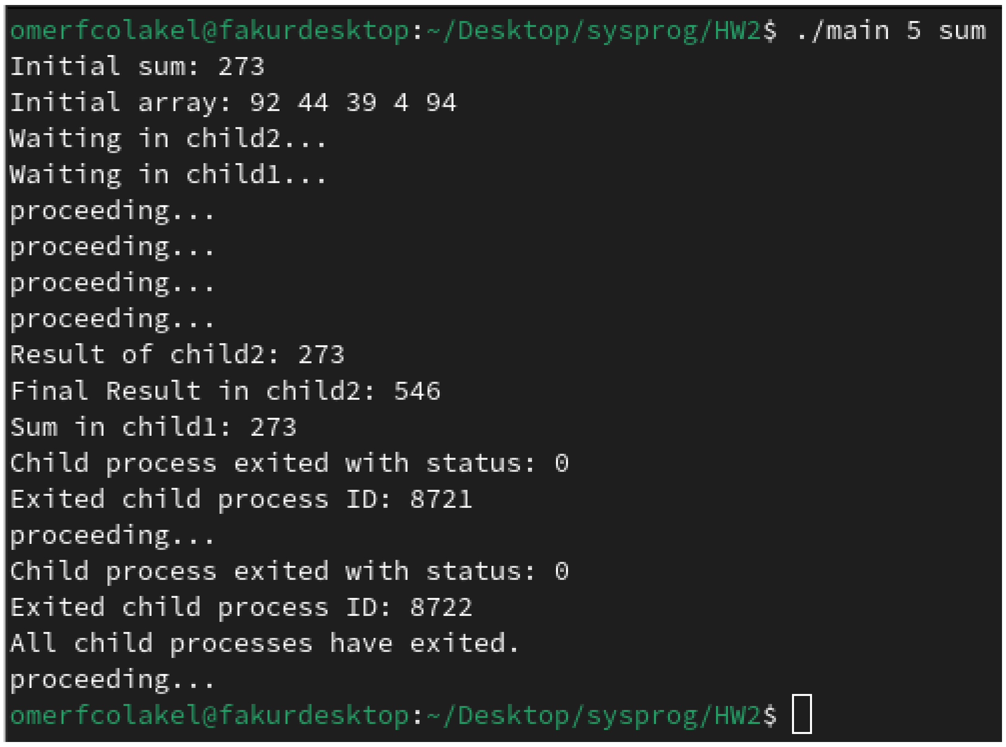
How to Compile and Run



```
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ make clean
rm -f main main.o
find . -type p -delete
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ make
gcc -Wall -Wextra -Werror -c main.c -o main.o
gcc -Wall -Wextra -Werror -o main main.o
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ ./main 5 sum
Initial sum: 370
Initial array: 68 92 36 95 79
```

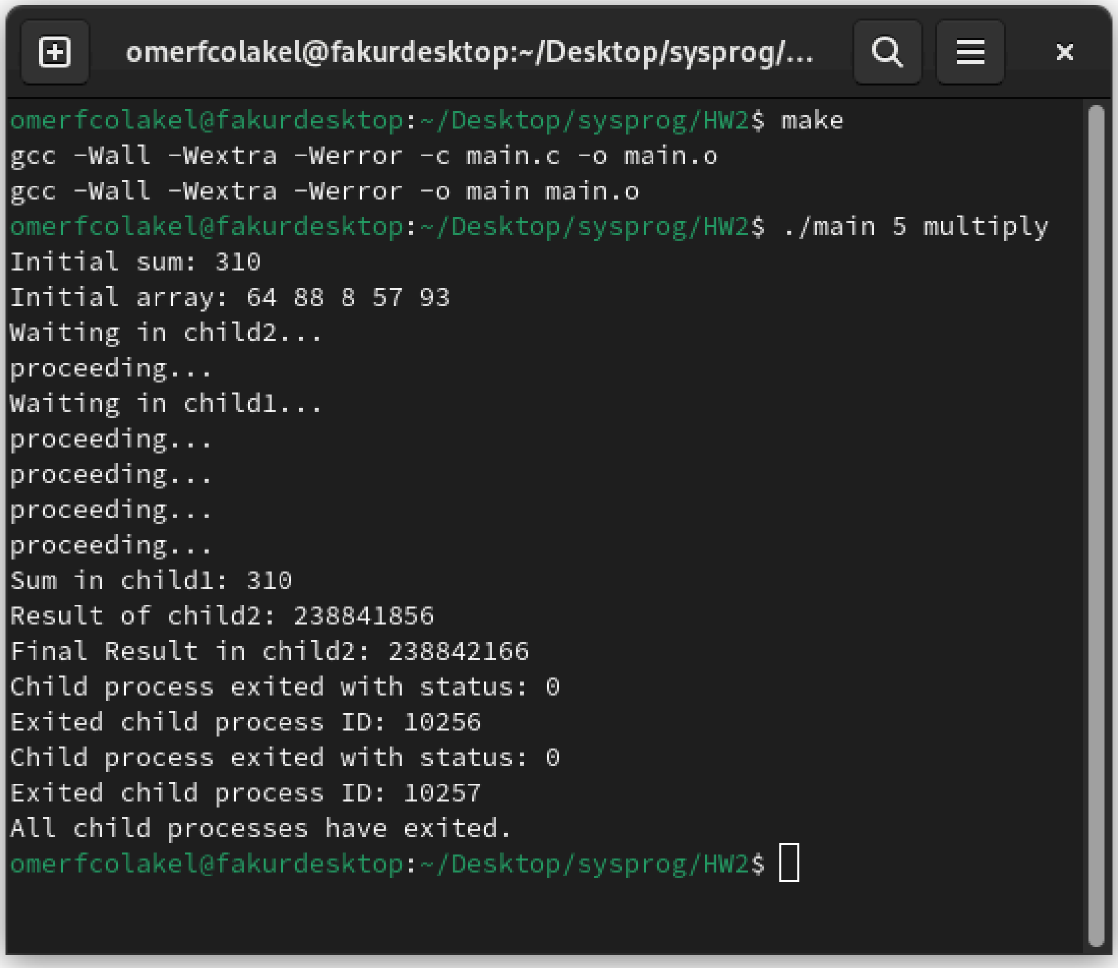
- In the root directory, you can run "make" in the terminal to compile the program.
- "./main <integer number> <operation>" runs the compiled program.
- If the input for the program is not valid, then it prints the valid usage and exits with failure.
- "make clean" deletes the executable and the pipes.

Example Inputs and Outputs



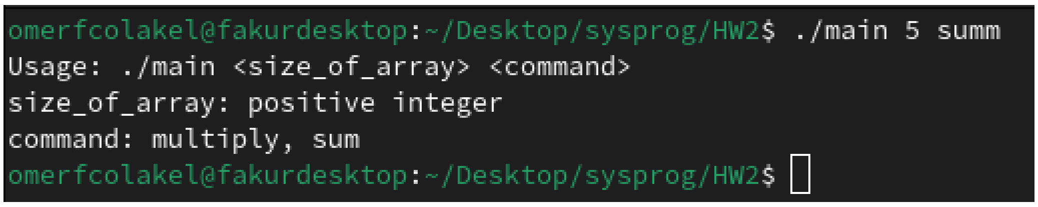
```
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ ./main 5 sum
Initial sum: 273
Initial array: 92 44 39 4 94
Waiting in child2...
Waiting in child1...
proceeding...
proceeding...
proceeding...
proceeding...
Result of child2: 273
Final Result in child2: 546
Sum in child1: 273
Child process exited with status: 0
Exited child process ID: 8721
proceeding...
Child process exited with status: 0
Exited child process ID: 8722
All child processes have exited.
proceeding...
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$
```

1. Summation of numbers in an array of 5 on both of the children without an issue.



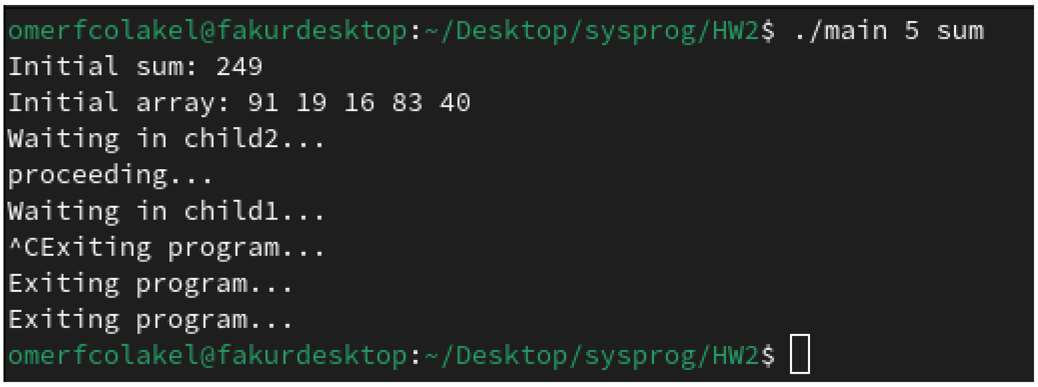
```
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ make
gcc -Wall -Wextra -Werror -c main.c -o main.o
gcc -Wall -Wextra -Werror -o main main.o
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ ./main 5 multiply
Initial sum: 310
Initial array: 64 88 8 57 93
Waiting in child2...
proceeding...
Waiting in child1...
proceeding...
proceeding...
proceeding...
proceeding...
Sum in child1: 310
Result of child2: 238841856
Final Result in child2: 238842166
Child process exited with status: 0
Exited child process ID: 10256
Child process exited with status: 0
Exited child process ID: 10257
All child processes have exited.
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ 
```

2. Summation in first and multiplication in second child of an array with the length of 5.



```
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ ./main 5 summ
Usage: ./main <size_of_array> <command>
size_of_array: positive integer
command: multiply, sum
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ 
```

3. Invalid run.



```
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ ./main 5 sum
Initial sum: 249
Initial array: 91 19 16 83 40
Waiting in child2...
proceeding...
Waiting in child1...
^CExiting program...
Exiting program...
Exiting program...
omerfcolakel@fakurdesktop:~/Desktop/sysprog/HW2$ 
```

4. Interrupted (FIFOs are deleted)

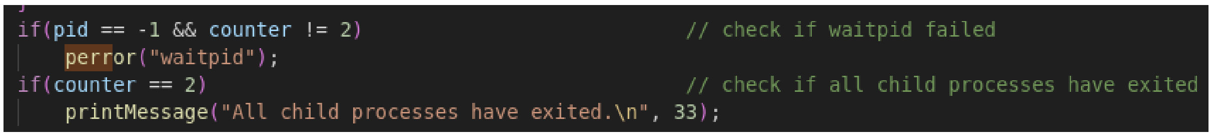
Functions

1. printMessage

    - It prints the given char array to the standard output with the given length.
    - Number of bytes written is returned or -1 on error.
    - All printing operations to the standard output are handled by this function.

2. sigchld_handler

    - It is the handler for SIGCHLD signals.
    - It takes the id of the child and prints both the status and id of it.
    - When both processes end, prints that all of the children are exited.
    - Uses perror when there was an error with waitpid().



```
if(pid == -1 && counter != 2)                         // check if waitpid failed
    perror("waitpid");
if(counter == 2)                                      // check if all child processes have exited
    printMessage("All child processes have exited.\n", 33);
```

3. sigint_handler

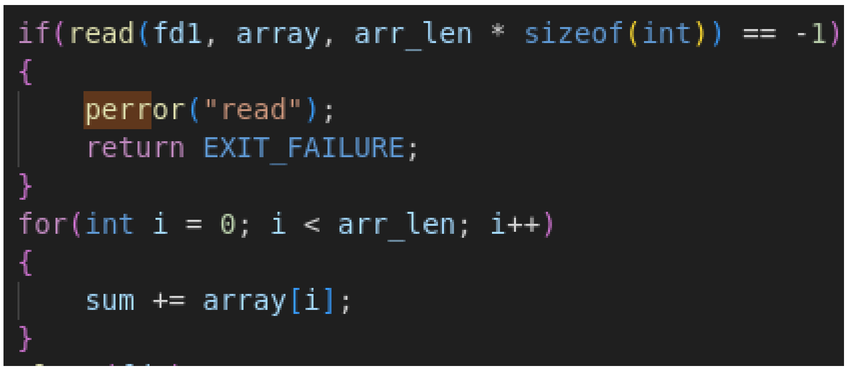
    - It is the handler for SIGINT signals.
    - Unlinks both FIFOs and exits the program.
    - Prints that the process is exiting.

4. printUsage

    - Prints how to use the program.

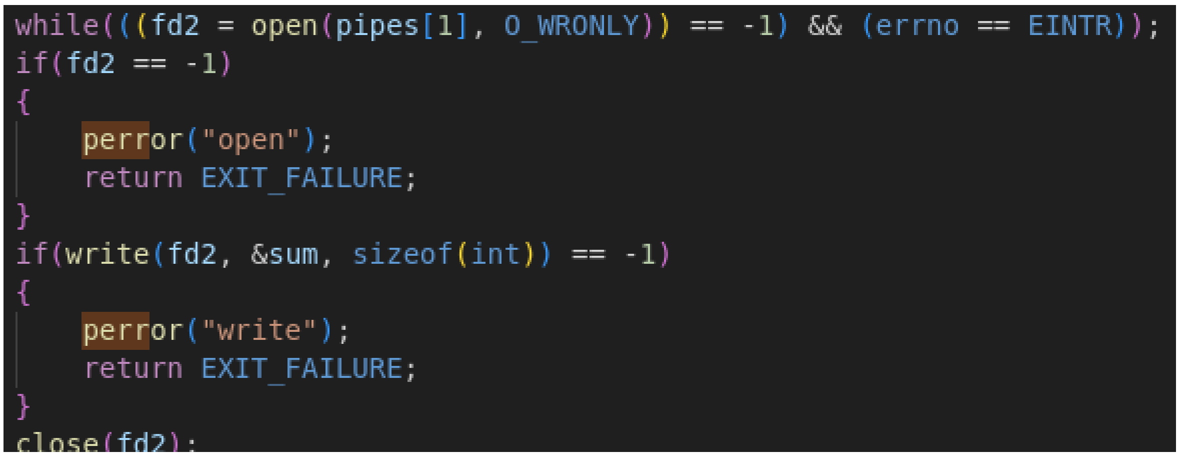
5. child1

    - It does everything that the first child should do. It is called inside the child process and not in the parent process.



```c
if(read(fd1, array, arr_len * sizeof(int)) == -1)
{
    perror("read");
    return EXIT_FAILURE;
}
for(int i = 0; i < arr_len; i++)
{
    sum += array[i];
}
```

    - Reads an array of integers from FIFO1 and sums the numbers.
    - Writes the result to the FIFO2.

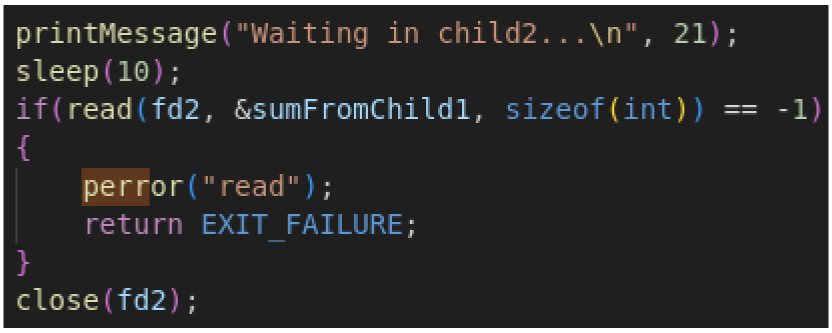


```c
while(((fd2 = open(pipes[1], O_WRONLY)) == -1) && (errno == EINTR));
if(fd2 == -1)
{
    perror("open");
    return EXIT_FAILURE;
}
if(write(fd2, &sum, sizeof(int)) == -1)
{
    perror("write");
    return EXIT_FAILURE;
}
close(fd2);
```

    - When there is a problem during open(), read(), write() etc., uses perror() and exits with failure.
    - Requires the name of the FIFOs and length of the array.
    - After open, read and sum operations; sleeps for 10 seconds using sleep().
    - Prints the result using printMessage().

6. child2

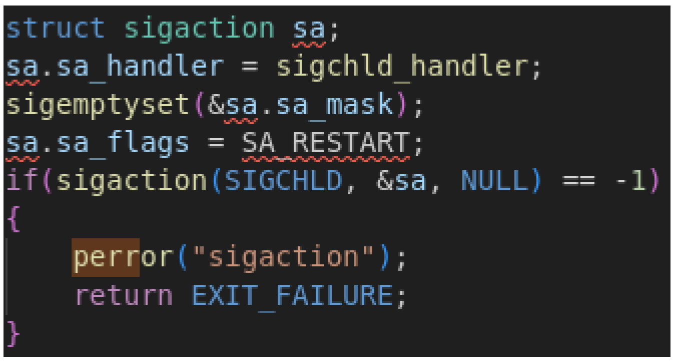
- It does everything that the second child should do. It is called inside the second child process and not in the parent process.
- Reads the command, an array of integers first and then the result of the first child from the second FIFO.
- When there is a problem during open(), read(), write() etc., uses perror() and exits with failure.
- Requires the name of the FIFOs, length of the array and length of the command
- In total, sleeps for 10 seconds.
- Prints the result using printMessage().



```
printMessage("Waiting in child2...\n", 21);
sleep(10);
if(read(fd2, &sumFromChild1, sizeof(int)) == -1)
{
    perror("read");
    return EXIT_FAILURE;
}
close(fd2);
```

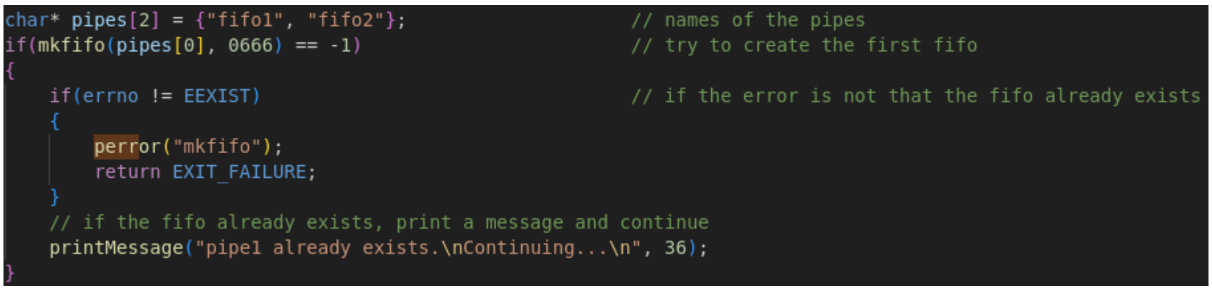
7. main

- It checks if the argument count and arguments themselves are valid
- Creates sigaction objects for signals and sets the handlers



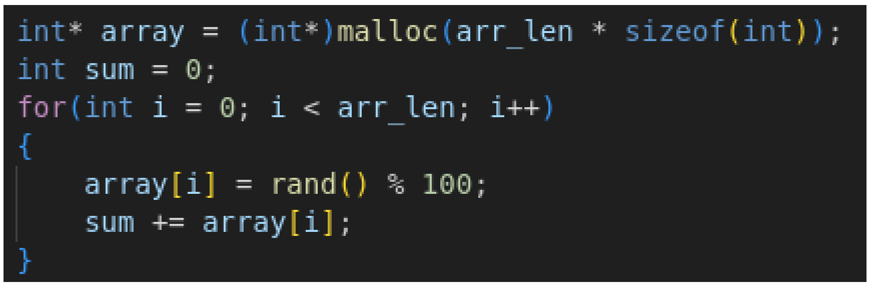
```
struct sigaction sa;
sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if(sigaction(SIGCHLD, &sa, NULL) == -1)
{
    perror("sigaction");
    return EXIT_FAILURE;
}
```

- Creates pipes and checks errors



```c
char* pipes[2] = {"fifo1", "fifo2"};                        // names of the pipes
if(mkfifo(pipes[0], 0666) == -1)                            // try to create the first fifo
{
    if(errno != EEXIST)                                     // if the error is not that the fifo already exists
    {
        perror("mkfifo");
        return EXIT_FAILURE;
    }
    // if the fifo already exists, print a message and continue
    printMessage("pipe1 already exists.\nContinuing...\n", 36);
}
```

- Creates child process with fork and checks errors
- Creates an array with given size and fills it with random integers in the range of 0 and 9



```c
int* array = (int*)malloc(arr_len * sizeof(int));
int sum = 0;
for(int i = 0; i < arr_len; i++)
{
    array[i] = rand() % 100;
    sum += array[i];
}
```

- Opens and writes necessary arguments to the pipes. Checks errors
- Creates the second child
- In a while loop, writes “proceeding…” every two seconds.
- Frees all used memory
- Unlinks all fifos.
- At any error, exits with failure

Synchronization

Second process is created before the first one. While the main function opens the second FIFO in write only mode and writes into it, child2 opens it in read only mode and reads it. While child2 continues working, main creates the first child, opens the first FIFO and writes into it. While it is writing, the first child reads from the first FIFO. At this point, main doesn’t have to write anything so it just closes the FIFOs from its end. First child writes its result to the second FIFO and sleeps. Second child also calculates the first result using the command and the array and then sleeps. After 10 seconds (main is printing “proceeding…” in the process), child1 writes and child2 reads to/from the second FIFO. Child1 ends after printing its result and child2 ends after calculating the final result and printing it.

Bonus Part

I have implemented both of the bonus parts as you can see from the rest of the report.