

# W11-SP Report

190022658

April 21, 2021

## 1 Overview

- The objective of this practical was to gain experience with processes and simple Inter-process Communication (IPC) in C.
- This objective was to write a program that creates multiple processes in C, and uses pipes for inter-process communication.

## 2 Design

In order of implementation -

### 2.1 Pipeline

**Pipeline.h** We had to implement a Pipeline struct for use in Pipeline.c.

This is to store the “user-defined” functions.

The Pipeline struct has two variables for good use

- funcs (Function\*)
- funcsSize (int)

**Pipeline.c** Implementation of all the functions for the pipeline.

**new\_Pipeline** This uses malloc to create the **Pipeline** structure, assign 0 to funcSize (as there are no functions in the array yet), and to create the **Function** array. This returns a pointer to the pipeline structure created.

**Pipeline\_add** This reallocates space in the **Function** array to add more functions. If there is no more space available, the function returns false and the program that the user defined exits. Otherwise, the function is added to the pipeline and the size of array is incremented.

**Pipeline\_execute** This function runs all of the functions in the array (which is inside the Pipeline structure), in different processes. There were two ways of doing this, one is to create multiple process for a single parent and giving them a “function to do”. This makes quite a lot of sense conceptually, so I implemented this (**Pipeline\_execute\_alt**) along with what the specifications said. The specifications said to create a child for every parent.

In the specification’s one implementation I used a prevFd[2] variable to keep track of the last pipe used. This is to be used as input for the process in the current iteration. After that there is a loop that creates a pipe (to be used as output for the function). There is a fork function then and the child pid value is stored inside a variable called child. After that the program checks for an error and responds accordingly. Then for the parent, we close the pipes not needed (although what happens to the ones already opened, do they need to be closed? The alt method contains another way to handle pipes). After closing the pipes, the function of current index in the pipeline object is executed with the prevFd[0] being input and fd[1] being output. After that the parent waits for its child and exits without error. Note that it does not exit if it is the parent of all the other functions. It would still work but the cleanup function will not be called. Lastly the prevFd is assigned as fd.

This loops on till all the functions are executed. The last process created is not needed since the parent does its job. So it is exited using an if statement. The program somehow works without it.

The other way is to “int child = i == N-1 ? 1 : fork();”. This way the process is never created and the current process becomes the “father” (according to the if statement).

## 2.2 TestPipeline

This is described in the Testing section.

## 3 Testing

This program works as showed in the requirements.

```
make
clang -g -Wall -Wextra -c Pipeline.c
clang -g -Wall -Wextra SumSquaresPipeline.o Pipeline.o -o SumSquaresPipeline
clang -g -Wall -Wextra SievePipeline.o Pipeline.o -o SievePipeline
```

```
./SievePipeline < sieve1.in
Setting up pipeline to sieve out the first 5 prime numbers up to 15
generateInts: process 378264, parent 342925
sieveInts: process 378265, parent 378264: prime = 2
sieveInts: process 378266, parent 378265: prime = 3
sieveInts: process 378267, parent 378266: prime = 5
sieveInts: process 378268, parent 378267: prime = 7
sieveInts: process 378269, parent 378268: prime = 11
```

```
./SumSquaresPipeline < squares.in
Setting up pipeline to calculate the sum of squares of integers 1 to 10.
generateInts: process 378320, parent 342925
squareInts: process 378321, parent 378320
sumIntsAndPrint: process 378322, parent 378321
sumIntsAndPrint: result = 385
```

Along with that, I used a copy of TestStack.c (RPN calculator practical) as TestPipeline.c, there are multiple functions that are used to test the Pipeline implementation. They are self-explanatory and they all pass. The third test uses the pipeline implementation to double and print the result

(idea from specification).

This should double the number using a pipeline

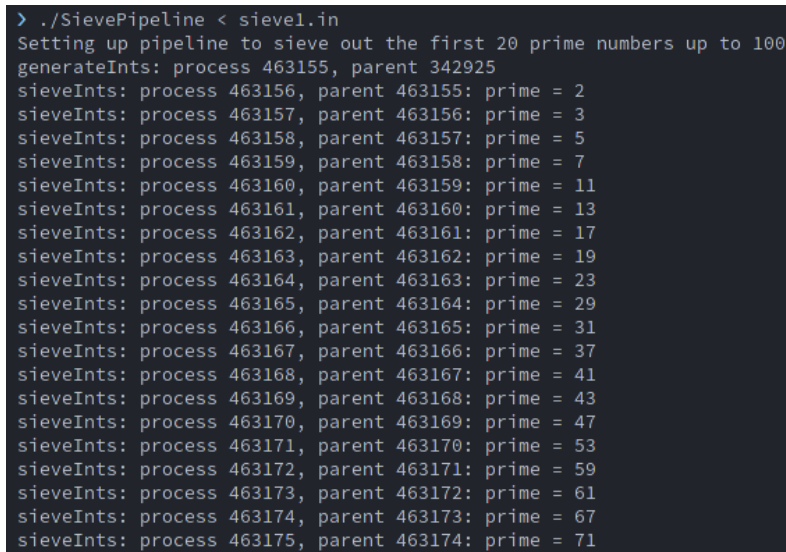
Enter a number: 3

Received int: 6

Is that the expected output (y/n): y

Pipeline Tests complete: 3 / 3 tests successful.

-----

A terminal window with a dark background and light-colored text. The prompt is '> ./SievePipeline < sieve1.in'. The output shows the setup of a pipeline to sieve the first 20 prime numbers up to 100. It then lists 20 lines of output, each showing a process ID, a parent ID, and a prime number. The primes listed are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, and 71.

```
> ./SievePipeline < sieve1.in
Setting up pipeline to sieve out the first 20 prime numbers up to 100
generateInts: process 463155, parent 342925
sieveInts: process 463156, parent 463155: prime = 2
sieveInts: process 463157, parent 463156: prime = 3
sieveInts: process 463158, parent 463157: prime = 5
sieveInts: process 463159, parent 463158: prime = 7
sieveInts: process 463160, parent 463159: prime = 11
sieveInts: process 463161, parent 463160: prime = 13
sieveInts: process 463162, parent 463161: prime = 17
sieveInts: process 463163, parent 463162: prime = 19
sieveInts: process 463164, parent 463163: prime = 23
sieveInts: process 463165, parent 463164: prime = 29
sieveInts: process 463166, parent 463165: prime = 31
sieveInts: process 463167, parent 463166: prime = 37
sieveInts: process 463168, parent 463167: prime = 41
sieveInts: process 463169, parent 463168: prime = 43
sieveInts: process 463170, parent 463169: prime = 47
sieveInts: process 463171, parent 463170: prime = 53
sieveInts: process 463172, parent 463171: prime = 59
sieveInts: process 463173, parent 463172: prime = 61
sieveInts: process 463174, parent 463173: prime = 67
sieveInts: process 463175, parent 463174: prime = 71
```

Figure 1: Image showing that code works with a big number

## 4 Conclusion

This practical was quite helpful in understanding how pipes and processes work. This was quite new to me, and understanding the system processes was very interesting too.