HW3: Programming Assignment v10.1.2024 6:00PM

IPC with Pipe and shared Memory

For this assignment you will write and test a program with multiple processes that communicate using pipes and shared memory.

Due Date: Thursday October 3rd

Extended Due Date with 10% penalty per day: Saturday October 5th

This assignment is a modification of HW2. You will be creating 6 programs: `Generator.c`, `Reader.C`, `Primes.c`, `Perrin.c`, `Composite.c`, `and Square.c` they will be described below. Furthermore, instead of using WEXITSTATUS to retrieve the return from the child programs. You will use **pipes** and **shared memory**.

First `Generator` will create a child process to spawn `Reader`. `Reader` will take the file name as input. The input file will be a text file containing only numbers. `Reader` will return to `Generator` the sum of all the lines in the file via a pipe. The parent program reads the contents of the pipe and stores it in a one-dimensional array. `Generator` then creates a shared memory segment for three child processes and invokes the `Prime`, `Square`, and `Composite` executables passing the entire array at once. As in assignment two, the child programs `Prime`, `Square`, and `Composite` will perform their calculations and printouts as previously done in HW2. Additionally, `Composite` will spawn a new child process and invoke the `Perrin` executable. `Composite` will also create a pipe and shared memory segment for the `Perrin` child process. Once a child process has calculated all its respective values, it will write the last calculated value to the shared memory segment to be read by its parent process. At the end of the computation, `Generator` will have the results from all executables. The child processes `Prime`, `Square`, and `Composite` will run **concurrently.** `Composite` and `Perrin` will run **sequentially.**

Please see the Notes at the bottom before starting to write your program.

1. Description of Task

This assignment builds on HW2. Specifically, we will be using inter-process communication(IPC) for communications between the `Generator`, `Reader`, `Prime`, `Square`, `Composite`, and `Perrin` processes. All instances of `Prime`, `Square`, `and Composite` should run **concurrently**. `Composite` and `Perrin` instances should run **sequentially**.

1. `Generator` creates a pipe and forks a child process. The child process executes the `Reader` program with the name of the .txt file filled with numbers, one per line, as an argument. `Reader` will open the file and read the contents one line at a time, keeping a running sum of the numbers in the file. `Reader` will then close the file and write the calculated sum to the write end of the pipe it inherited as an argument from the parent process when the parent called the execlp() function. When control returns to `Generator`, it will read the content from the pipe.

2. `Generator` will read the contents from the pipe into a char array[] (or char *) and print them. Then, let N be the converted Integer value of the pipe contents. `Generator` must calculate the sum of all digits of N. i.e. If N is 42 the sum would be 6. This sum will be used as an argument of `Prime`, `Square`, `Composite`, and `Perrin` programs.

3. `Generator` creates a shared memory process for each `Prime`, `Square`, `Composite` processes. It then forks three child processes and uses exec to run the `Prime`, `Square`, `Composite`

programs. After `Composite` completes its calculations, it should spawn a `Perrin` child and return to `Generator` the values of both its calculations well as `Perrin`. Each `Prime, Square, and Composite` program should be running concurrently, and `Composite` and Perrin should run sequentially.

4. The sum of all digits N (as defined in step 2.) is passed to each program from its parent process. Each child should print out the first N numbers of its series

5. Then, `Prime/Square/Composite /Perrin` calculate the sum of the first N numbers of its series and prints out the sum to stdout. `Prime/Square/Composite /Perrin` will each write this new sum to their respective shared memory segment.

6. `Composite` will take the results from its `Perrin` child process and write them to the shared memory it shares with `Generator` along with the results of its own computation.

7. Once control returns to `Generator`, it will read the shared memory segments, and print out the respective sums to stdout.

`Generator` does the following:

1. Create a pipe using the following steps

- Create an integer array of size 2 and create a pipe using that integer array.

2. Send the details to the child process `Reader` using the following steps:

A. sprintf to get the file descriptor of the write end of the pipe into the character array.
B. Fork a child process and replace its executable with the `Reader` executable.
C. Pass the character array to the `Reader` as a second argument after file name.

3. Read the content from the pipe into a character array of size 10 using the following steps:

- Close the writing end, and then read the content from the read end of the pipe using the read() function. Don't forget to close the read end of the pipe too.
- Convert the contents into an integer, and then prints out a message with the integer value.

4. Save the content into an integer pointer and let N be the number which is pointed to by the pointer. Then calculate the sum of all the digits of N. After the sum of the digits is calculated, convert the sum into a string through sprint(). This sum is to be used as an argument of `Prime/Square/Composite /Perrin`.

5. Create three shared memories with the names "SHM_Prime", "SHM_Square", and "SHM_Composite ". Print out the name of each shared memory segment. Then fork three child processes that run the `Prime/Square/Composite` executable and each child process gets the appropriate shared memory name which is sent as the third argument to the execlp() function, the fourth argument to the execlp() function is the content that is read from the pipe:

- The shared memory segment should be of size 32 bytes. Since a shared memory is created for each child, use O_CREAT and open in read write mode (ORDWR). It uses mmap to create a pointer to the shared memory. The name of the shared memory should follow the standard "SHM_Name".

- Fork a child using the execlp, and the arguments are the executable name, shared memory name, and the char array[] where you calculated the sum of all
digits of N
- Once control returns from `Prime/Square/Composite`, it then reads from the shared memory segment and places the calculated values, placed in the shared memory by each of the respective child processes, into local variables which are the sums of the sequences.

- It then prints out these variables as stdout after all the processes have finished execution

`Reader` does the following:

1. Receives the name of the file and the file descriptor of the write end of the pipe as arguments from `Generator`.
2. Using atoi() copies the pipe reference (3rd argument) in argv into an integer variable.
3. Read the contents of the file line by line, keeping a running sum of each line.
4. Keep the sum under 99,999,999 using modular arithmetic.
5. Write the final sum into the pipe.

`Prime, Square, Composite and Perrin` each do the following:

1. Receives a shared memory name and a character array. In this step, it needs to check the number of arguments
2. Converts the character array into an integer N using the atoi() function.
3. Prints out the first N numbers of the series.
4. Then print out the sum of the first N numbers of the sequence.
5. Once all printing is done, the value of the sum that was calculated is written to the shared memory.

Here are the examples of each sequence when N is 8:

Prime: 2 3 5 7 11 13 17 19
Square: 1 4 9 16 25 36 49 64
Composite: 4 6 8 9 70 12 14 15
Perrin: 3 0 2 3 2 5 5 7

`Composite specific behavior:`

1. Just like `Generator`, `Composite` will create new section of shared memory called "SHM_Perrin" and print the name of the shared memory segment.
2. `Composite` will fork a child process and invoke the `Perrin` executable using the same methodology as the `Generator` executable.
3. Once control returns to `Composite`, it will read `Perrin's` results from the shared memory segment and write them into the `Generator/Composite` shared memory segment.

Background: For the background of the assignment, review the related material (sections on POSIX shared memory and ordinary Pipes in the text book), the related self-exercise example you ran recently and consult the man page (`shm_open()`, `ftruncate()`, `mmap()`, `shm_unlink()`) as needed. You may find a search for "`man shm_open()`" etc. to be beneficial. Please note that this is not conventional serial C programming.

More details: shm_open(), ftruncate(), mmap(), shm_unlink(), POSIX Shared Memory

2. Task Requirements

1. `Generator` creates a pipe and checks if pipe creation failed. Then forks a child process to execute `Reader`.

2. `Reader` reads the file, keeps running sum, writes the result as text to the pipe, and then closes the write end of the pipe.

3. `Generator` then reads the contents from the read end of the pipe into a char []

4. `Generator` calculates the correct sum of digits N using the pipe contents.

5. `Generator` then creates three shared memory segments with appropriate attributes (truncate to the size of 32 bytes, use mmap with PROT_READ and MAP_SHARED). It prints the name and the file descriptor of the shared memory.

6. `Generator` then forks appropriate `Prime/Square/Composite` programs as child processes.

7. For each of the `Prime/Square/Composite/Perrin` executables the appropriate shared memory name is written into the third to last position in the execlp() argument list. The second to last element in the list is the character array. The last element is set to NULL. Use execlp() for executing the `Prime/Square/Composite /Perrin`.

8. `Prime/Square/Padova/Perrin` processes perform their respective operations on the character array. When all operations are completed, it copies the final calculated value into the shared memory segment. The `Prime/Square/Composite/Perrin` processes also display their calculated values per the standards that can be seen in the sample output.

9. After `Composite 's` calculations are complete, it will fork a child process and exec the `Perrin` executable using the same process as `Generator`. Once `Perrin` is complete it writes to the memory it shares with `Composite` and returns control to `Composite`. `Composite` will then read `Perrin's` result from shared memory and forward it to `Generator` along with its own result.

10. `Generator` copies the values from shared memory into the appropriate integer variable, unlinks the shared memory, and then prints these variables to stdout.

11. `Prime, Square, and Composite/Perrin` processes should execute **concurrently**. `Composite and Perrin` should run **sequentially**.

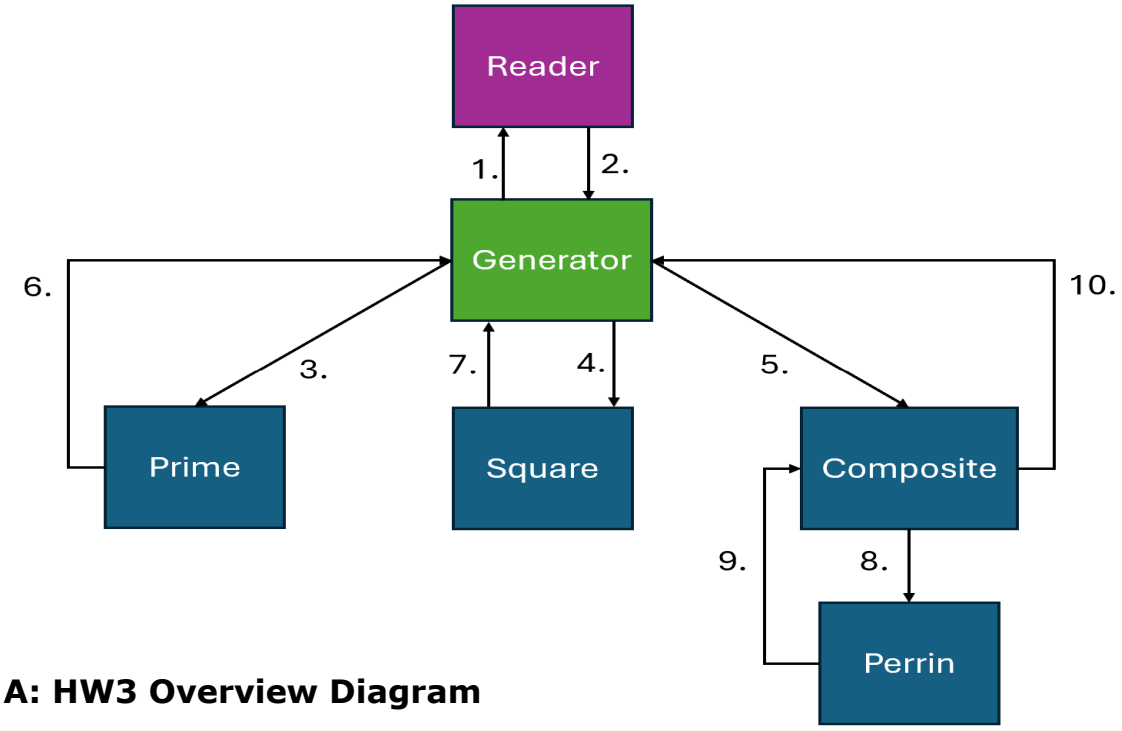

Fig. A: HW3 Overview Diagram

3. Files Provided

- Files provided for this assignment include the description file (this file).

- A Makefile that creates all of the target executables.

4. Example Outputs (**Note** – The process IDs and the order of Prime/Sqaure /Composite /Perrin may vary)

$ ./Generator file_01.in

[Generator][378649]: contents read from the read end pipe: 1299

[Generator][378649]: Created Shared memory "SHM_Prime" with FD: 3

[Generator][378649]: Created Shared memory "SHM_Square" with FD: 4

[Generator][378649]: Created Shared memory "SHM_Composite " with FD: 5

[Prime][378651]: The first 21 numbers of the Prime sequence are:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73

[Prime][378651]: The sum of the first 21 numbers of the Prime sequence is: 712

[Square][378652]: The first 21 numbers of the Square sequence are:

1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400 441

[Square][378652]: The sum of the first 21 Square sequence is 3311

[Composite][378653]: The first 21 numbers of the Composite sequence are:

4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 32, 33

[Composite][378653]: The sum of the first 21 numbers of the Composite sequence is 400

[Composite][378653]: Created Shared memory "SHM_Perrin" with FD: 2

[Perrin][378654]: The first 21 numbers of the Perrin sequence are:

3 0 2 3 2 5 5 7 10 12 17 22 29 39 51 68 90 119 158 209 277

[Perrin][378654]: The sum of the first 21 numbers of the Perrin sequence is 1128

[Generator][378649]: Prime last number: 712

[Generator][378649]: Square last number: 3311

[Generator][378649]: Composite last number: 400

[Generator][378649]: Perrin last number: 1128

Note: The contents of the input file `file_01.in`:

```
370
929
```

5. What to Submit

Use the CS370 *Canvas* to submit a single .zip file that contains:

- All **.c** files listed below and descriptive comments within,
  - `Generator.c`
  - `Reader.c`
  - `Prime.c`
  - `Square.c`
  - `Composite.c`
  - `Perrin.c`
- a **Makefile** that performs both a *make* and a *make clean* (notice the targets). Use the supplied file.
  - a **README.txt** file containing a description of each file and any information you feel the grader needs to grade your program.

For this and all other assignments, ensure that you have submitted a valid .zip file. After submitting your file, be sure to download it and try to unzip it. This will help you to make sure that it was created successfully. If we can't unzip your assignment, we can't grade it.

Filename Convention: The archive file must be named as: <FirstName>-<LastName>-HW3.zip. E.g., if you are John Doe and submitting assignment 3, then the zip file should be named **John-Doe-HW3.zip**

6. Grading

The assignments must compile and function correctly on machines in the CSB-120 Lab. Assignments that work on your laptop on your particular flavor of Linux/Mac OS X, but not on the Lab machines are considered unacceptable. Solutions that do not compile when the `make` command is executed will receive a grade of zero.

The grading will be done on a 100-point scale. The points are broken up as follows:

You are required to **work alone** on this assignment.

Objective	**Points**
Correctly performing Tasks 1-11	85 points
Descriptive comments for important lines of code	5 points
Compilation with no warnings	10 points

Click here for the class policy on submitting late assignments.

Notes:

1. The filename argument to `Generator` is mandatory, ***not*** optional.
2. For your convenience, an assignment overview diagram is provided (Fig. A).
3. For your testing purposes sample input contents are mentioned at the end of Sample Outputs (section 4). You must write your own tests to test the program thoroughly.
4. This program may not work on your Mac OS X or other systems. Try to run the program on a lab system, especially if you keep getting a segmentation fault when the code seems correct.
   a. Your solution **will** be tested on the lab machines.
5. If you are receiving an exit state with the message `undefined reference to 'smh_open'` or `undefined reference to 'smh_unlink'`, try using the flag "`-lrt`" to compile.
6. Please remember to **unlink the shared memory**. Failing to do that may cause problems for other users of the machine.
7. Beware the dangers of race conditions and deadlocks!

a. Design your programs carefully.
b. Test your programs for correctness individually before multi-threaded debugging.

Updates: Any updates will be noted below.

9/23: link for mmap() updated to a valid link.
10/1: Since there are no questions, answers to the four questions are not needed.