HW3: Programming Assignment v29.16.2021 8:00PM

WORKING WITH INTER-PROCESS COMMUNICATIONS

The objective of this assignment is to write and test programs with pipes and shared memory.

Due Date: Thursday, February 25, 2021, 11:00 pm

Extended Due Date with 20% penalty: Friday, February 26, 2021, 11:00 pm

For this assignment you will modify your submission for HW2. There will be one additional program called `Reader`. Furthermore, instead of using WEXITSTATUS to retrieve the return from the child programs. You will use a pipe and shared memory.

First `Initiator` 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 `Initiator` 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. `Initiator` then creates a shared memory segment for each of the three child processes and invokes the `Pell`, `Composite`, and `Total` executables and passes the entire array at once. As in assignment 2, the child programs `Pell`, `Composite`, and `Total` will perform their calculations and printouts as previously done in HW2. Once each of the child programs has calculated all values, it will write the last calculated value to the shared memory, which will be read by the `Initiator`. The child processes `Pell`, `Composite`, and `Total` executables that are performing the count operations on the passed character array should run *concurrently* (i.e., not one after the other).

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 `Initiator`, `Reader`, `Pell`, `Composite`, and `Total` processes. All instances of `Pell`, `Composite`, and `Total` should run concurrently. `Pell`, `Composite`, and `Total` processes will here on be referred to as `Pell/Composite/Total`.

1. `Initiator` creates a pipe and forks a child process. The child process executes the `Reader` program with the name of the .txt file as an argument. `Reader` will open the file and read the contents one line at a time. `Reader` reads all the lines and will keep a running sum. `Reader` then closes the file and then writes the calculated running sum to the write end of the pipe it inherited as an argument from the parent process when the parent process called the `execlp()` function. When control returns back to the `Initiator`, it will read the content from the pipe.

2. `Initiator` reads the contents from the pipe and stores it in a char array`[]`, then prints it. The `Initiator` then creates a shared memory segment for each `Pell/Composite/Total` processes. It then forks three child processes to run the `Pell/Composite/Total` programs. Each `Pell/Composite/Total` program should be running concurrently.

3. After calculating the values on the passed argument, `Pell/Composite/Total` program writes the last value calculated to the shared memory segment and the program returns control to `Initiator`. The `Initiator` then reads these shared memory segments and stores these counts to their respective `Pell/Composite/Total` counters.

4. Initiator then prints out these counters to `stdout`.

Initiator does the following:

1. Create a pipe using the following steps.
   - Create an integer array of size 2, and create a pipe using the integer array.

2. Send the details to the child process `Reader` using the following steps:
   - r`sprintf` to get the file descriptor of the write end of the pipe into this character array.
   - Fork a child process and replace its executable by the `Reader` executable.
   - 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 and close this end too.

4. It creates three shared memories with the names "SHM_Pell", "SHM_Composite", and "SHM_Total". It then forks three child processes that run the `Pell/Composite/Total` executable, and each child process gets the appropriate shared memory name, which is sent to the `Pell/Composite/Total` program, 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_CREATE and open in read write mode (O_RDWR). It uses mmap to create a pointer to the shared memory. The name of each shared memory should follow the standard.
   - Fork a child using `execlp`, and the arguments it will take are the name of the executable, the name of the shared memory and the char array`[]`where you stored the return value from `Reader`.
   - Once the control returns from `Pell/Composite/Total` processes, 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 counter variables.
   - It then writes these counter variables to the screen 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 the Initiator.

2. Using `atoi` copies the pipe reference which is the third argument in `argv` into an integer variable.

3. Read the contents line by line from the text file, keeping a running sum.

4. Write the contents into the pipe.

Pell, Composite, and Total each do the following:

1. It receives a shared memory name and a character array.

2. Calculated its respective operation on the value passed (character array) and print the results. Once all the counting is done, the last value that was calculated is written to the shared memory, and the control is returned.

Background: For the background of the assignment, review the related material (sec. 3.5.1 POSIX shared memory and 3.6.3.1 ordinary pipes), the related self-exercise example you ran recently and consult the man pages (`shm_open()`, `ftruncate()`, `mmap()`, `shm_unlink()`) as needed. You can simply search for "`man shm_open()`" etc. Please note that this is not conventional serial C programming.

2. Task Requirements

1. The `Initiator` creates a pipe and checks if pipe creation failed. It then forks a child process to execute `Reader`.

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

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

4. `Initiator` 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.

5. `Initiator` then forks appropriate `Pell/Composite/Total` program as a child process. For each of the `Pell/Composite/Total` processes, 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 argument list is the character array. The last element in the argument list is set to NULL. Use execlp( ) for executing the `Pell/Composite/Total` executables.

6. `Pell/Composite/Total` process performs the respective operation on the character array. When all the operations are completed, it copies the final calculated value to the shared memory segment. The `Pell/Composite/Total` process also displays the calculated values as per the standards that can be seen in the sample output.

7. `Initiator` then copies the values from the shared memory into the appropriate integer counter variables. It unlinks the shared memory. It then prints these counter variables to screen.

8. All the `Pell/Composite/Total` processes should be forked to execute concurrently, that is all of them should be running at the same time. **(Hint**: *Fork* and *exec* should be in one for loop and *wait* should be in a different for loop.)

3. Files Provided

Files provided for this assignment include the description file (this file). Sample output files are provided to you on Canvas.

You are needed to answer the questions in the README file.

4. Example Outputs (Note – The process IDs and the order may vary)(

```
$ ./Initiator input1.txt

Initiator[2242447]:  contents read from the read end pipe: 15
Initiator[2242447] : Created Shared memory "SHM_Pell" with FD:  3
Initiator[2242447] : Created Shared memory "SHM_Composite" with FD:  4
Initiator[2242447] : Created Shared memory "SHM_Total" with FD:  5
Composite[2242450]: First 15 composite numbers are:
Pell[2242449] : Number of terms in Pell series is 15
4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25,
Pell[2242449] : The first 15 numbers of the Pell sequence are:
0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741, 13860, 33461, 80782,
Total[2242451] : Sum = 120
Initiator[2242447] : Pell last number:  80782
Initiator[2242447] : Composite last number:     25
Initiator[2242447] : Total last number: 120


$ ./Initiator input2.txt

Initiator[2242463]:  contents read from the read end pipe: 22
Initiator[2242463] : Created Shared memory "SHM_Pell" with FD:  3
Initiator[2242463] : Created Shared memory "SHM_Composite" with FD:  4
Initiator[2242463] : Created Shared memory "SHM_Total" with FD:  5
Pell[2242465] : Number of terms in Pell series is 22
Pell[2242465] : The first 22 numbers of the Pell sequence are:
Composite[2242466]: First 22 composite numbers are:
Total[2242467] : Sum = 253
4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30, 32,
33, 34,
0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741, 13860, 33461, 80782,
195025, 470832, 1136689, 2744210, 6625109, 15994428, 38613965,
Initiator[2242463] : Pell last number:  38613965
Initiator[2242463] : Composite last number:     34
Initiator[2242463] : Total last number: 253
```

Note: `input1.txt` has the contents "`5\n10`" and `input2.txt` has the contents "`7\n15`".

5. What to Submit

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

- All .c files listed below and descriptive comments within,
  - `Initiator.c`
  - `Reader.c`
  - `Pell.c`
  - `Composite.c`
  - `Total.c`
- a Makefile that performs both a *make build* as well as a *make clean* (notice the targets)
- a README.txt file containing a description of each file and any information you feel the grader needs to grade your program, and answers for the 3 questions (see section 6, Grading)

For this and all other assignments, ensure that you have submitted a valid .zip/.tar file. After submitting your file, you can download it and examine to make sure it is indeed a valid zip/tar file, by trying to extract it.

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

Objective	Points
Correctly performing Tasks 1-8 (10 points each)	80 points
Descriptive comments for important lines of code	5 points
Compilation with no warnings	5 points
Providing a working Makefile	5 points
Questions in the README file	5 points

Questions: (To be answered in README file.)

1. Name the function that is used to create a pipe. (1 points)
2. Which ends of the pipe denote the read and the write ends of a pipe? (1 points)
3. Name the function used to map files or devices in to memory? (1 point)
4. Name the function used to open a shared memory object? What does it return? (2 points)

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

7. Late Policy

Click here for the class policy on submitting late assignments.

Notes:

1. The filename argument to Initiator is mandatory, ***not*** optional.
2. For your testing purposes two sample input contents are mentioned at the end of Sample Outputs (section 4).
3. This program may not work on your Mac OS X or other systems. Try to run the program on a lab system, if you keep getting a segmentation fault and the code seems correct. Your solution will be tested on the lab machines.
4. If you are receiving an exit states with the message `undefined reference to 'smh_open'` or `undefined reference to 'smh_unlink'`, try using the flag "`-lrt`".

5. Please remember to unlink the shared memory. Failing to do that may cause problems for other users of the machine.

Updates: Any updates will be noted below.