CMSC 312: Assignment 1: IPC Mechanisms

Due date: Monday Feb 21st (midnight) 2022.

Question 1: Shared Memory (15 pts):

Write a simple program (example code available on Canvas; can be used as starting point) where *processes synchronize via polling* such that a process A prints out the strings of two other separate writing processes (B first and then C second) from *shared memory*. Process A needs to 'wait' by polling until B and C finish writing their strings to memory. Each of processes A, B and C should be in different code files. Create TWO different shared memory locations: (i) one for storing the integer identifier and (ii) for storing the string.

Here is the sequence of events that needs to be implemented:

1. Process A writes "1" to the integer shared memory location. It also writes the string "I am Process A" into the string shared memory location and prints it out. Process A then waits until B and C completes.
2. Process B first checks if the Integer shared memory location has a "1" in it. When true, it writes the string "I am Process B" into the string shared memory location and then signals A & C that it is complete by writing "2" into the integer shared memory location (note: B must wait to write into the integer shared memory location until after process A writes "1" there). Then process B quits.
3. Process C writes the string "I am Process C" into the string shared memory and then signals A that it is complete by writing "3" into the integer shared memory location (note: C must wait until process B writes "2" into the integer shared memory). Then process C quits.
4. Process A needs to keep polling the integer shared memory location continuously. After process B writes "I am Process B", process A should be able to print it out. Similarly, after process C writes "I am Process C", process A needs to be able to print it out. Here some ad-hoc synchronization is needed. Specifically, consider using usleep() function to delay processes B and C between writing the string and the integer to give process A a chance to poll and print. Adjust the delay such that prints happen properly.
5. Process A is the last one to quit and prints out a "GoodBye" message at the very end before quitting.

Deliverable: THREE different code files; one each for Process A, B and C.

Question 2: Shared Memory (3+3 pts)

Create processes A, B and C from the same code file using fork. In this case you will have a single code file where the parent process can serve as Process-A, and two child processes serve as Process B and C.

Consider TWO implementations here: (i) Process B and C are two separate child processes (using two different forks from the parent process); (ii) Process B and C are nested forks (here Process C is forked from within process B and NOT from the parent process).

Deliverable: TWO different code files; one with regular forks, other one with nested forks.

Question 3: Message Queues (9 pts)

Review the programs (spock.c and kirk.c).

Answer (or discuss) questions listed below:

a) Discuss and evaluate what happens when you're running both in separate windows and you kill one or the other. (3)

b) Discuss what happens (and why) when you run two copies of kirk. (3)

c) Discuss what happens (and why) when your run two copies of spock.(3)

Deliverable: A pdf document with the answers to (a), (b) and (c).

Remote Procedure Calls

Note that you will need a .rhost file under your home directory, it includes authority to the machine that the 'server' accepts clients <hostname> and from what login id <loginid>.

The format for a .rhost file is:

```
<hostname1> <loginid1>
<hostname2> <loginid2>
.
.
.

<hostnameN> <loginidN>
```

The instructor's .rhost file may look like the below (login id is 'pghosh'):

```
127.0.0.1 pghosh
```

Great RPC tutorial - Prof David Marshall @ Cardiff University (optional reading):
http://www.cs.cf.ac.uk/Dave/C/node33.html

On rpcgen: http://www.cs.cf.ac.uk/Dave/C/node34.html#ch:rpcgen (note: you may not have permission to print on another server's Console for the printmessage tutorial).

Wikipedia: http://en.wikipedia.org/wiki/Remote_procedure_call (also check the link listed under 'external link" to the SGI Tutorial "Introduction to RPC Programming").

Question 4: Compute the median (10 pts)

Modify the average RPC programs (avg.x, avg_proc.c and ravg.c) so that it computes the **median** of a maximum of 100 numbers instead of the average of a maximum of 200 numbers. Test your program on the server.

The average RPC program is available on Canvas - a Makefile is included). Just type **make** (make sure you have an .rhost file under your home directory, and that 'white' space works correctly in the makefile (i.e., cut and paste the makefile will not work).

For the avg program: Run the RPC server program by starting the server: ./avg_svc

For the avg program: Run the RPC client program by: ./ravg 127.0.0.1 1 2 3 4

Try to use a different 'port number' than other classmates by changing the 22855 number given in the avg.x file. Suggestion, use the last 4 digits of your student ID number or some other unique number.

Deliverable: A zip or tar file with the full directory of files implementing this logic.

Question 5: Remote Procedure Calls (RPC): Sort numbers (10 pts)

Modify the average RPC program so that the server sends back a list of 'sorted' numbers to the client either in descending or ascending order given by an input parameter when invoking the client, e.g., the word "ascending:, "descending" (or just -a, -d) using command line arguments. You can rearrange the contents of the input array to additionally store this identifier to be sent to the client.

Here, the entire array needs to be returned instead of a single variable. Consider changing the return type of the average_1 function on the server from double * to input_data *. Sort the numbers in the input_data structure itself. Then the client can access the sorted numbers through the input_data structure.

Deliverable: A zip or tar file with the full directory of files implementing this logic.

Submission:

Upload the relevant files on Canvas.

Late Policy:

Late assignments will incur a penalty of 5 points per day for a maximum of 2 days.