EE442 HOMEWORK 2

Server-Client Problem

Due: April 9, 2017, 23:55

A. Doğa Hakyemez ARC-201 hdoga@metu.edu.tr
Okan Çalış calis.okan@metu.edu.tr

You are asked to implement a server program and a client program. The programs will communicate with two different queues. You will use shared memory to implement them. The content of the shared memory is in the figure.

Number of queue elements
Front index of client-to-server queue
Back index of client-to-server queue
Client-to-server queue
.
.
.
Front index of server-to-client queue
Back index of server-to-client queue
Server-to-client queue
.
.
.

The programs should work as follows:

- Server program will take an optional argument for the size of queues. For example: "`./server -q 100`" will initialize the both queue sizes to 100. Default value for the size should be 10.
- Server program will create a shared memory and named semaphores to be used between processes. It will also initialize necessary sections of the shared memory and wait for client program to start.
- In another terminal, client program will take an optional argument for the number of threads. For example: "`./client -t 100`" will create 100 sender and 100 receiver threads. Default value for the number should be 10.
- Client will look at the first block of the shared memory and get the size of the queue. Then, it will make sender and receiver **threads**. Client program will not exit until they are all finished.

- If an empty spot is available in the client-to-server queue, a sender thread will push a message to the queue. Otherwise, it will wait for a space to be emptied. The message consists of sender thread ID and a **random** 3-digit integer.
- If an item is present in the client-to-server queue, server program will create a child **process**. The child process will pop an item from the client-to-server queue. Then, in a non-critical section, it will sleep for a **random** amount of time (up to 1 second with increments of 1 millisecond).
- Then, if an empty spot is available in the server-to-client queue, the child process will push another message to this queue. Otherwise, it will wait for a space to be emptied. The message consists of the sender thread ID, child PID and two times the sent number.
- If an item is present in the server-to-client queue, a receiver thread will pop a message from the queue. Otherwise, the receiver threads will wait. Each receiver thread will print the message it gets to the screen. When all threads are finished, the client program will exit.
- Server program will properly exit (closing and unlinking the shared memory and any other synchronization elements it created) when the user presses Ctrl+C.

Specifications:

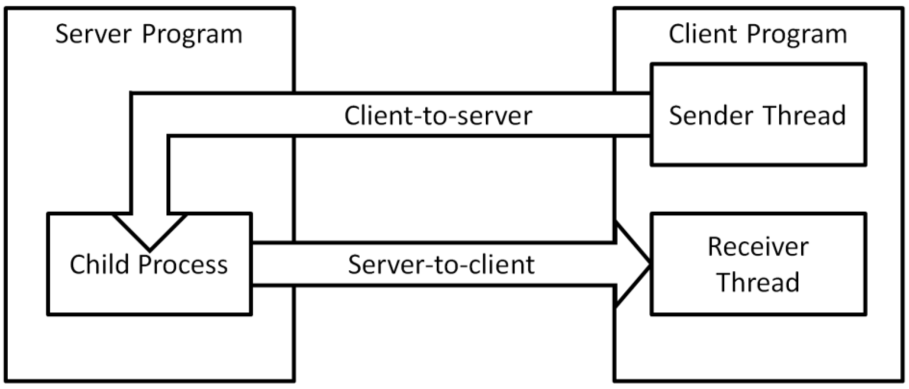
- Your program should not have deadlocks and should not leak memory.
- The server program should work for multiple client program calls.
- The queues should be synchronized **separately**.
- For shared memory and named semaphores use <sys/mman.h> and <semaphore.h> headers. Do **not** use <sys/shm.h> or <sys/sem.h>.
- Available header list is here with only the above restriction.
- You should compile your code with GCC (GNU Compiler Collection).

Hints:

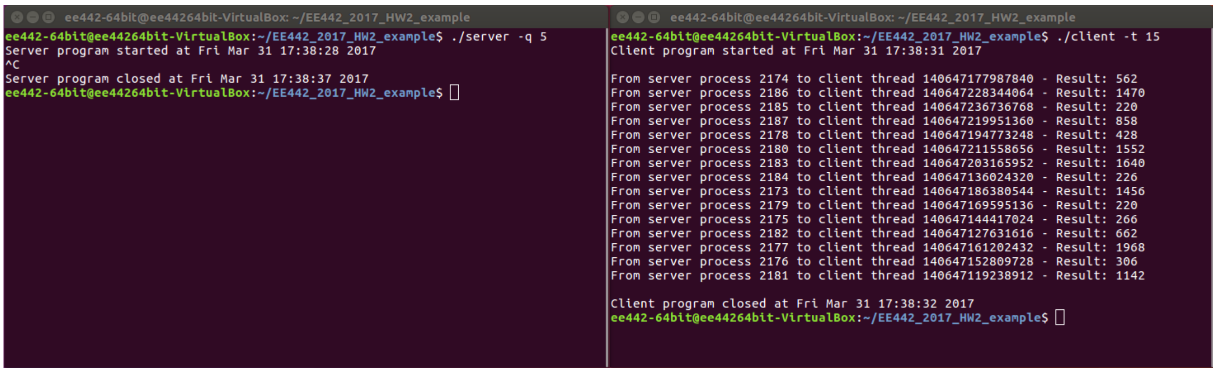
- You can use the following C structures for messages:
```
struct client_message {
    pthread_t client_id;
    int question;
};

struct server_message {
    pthread_t client_id;
    pid_t server_id;
    int answer;
};
```
- Named semaphores and shared memory are persistent. If you do not properly unlink them, they may create errors when you call your programs again. You can check "`/dev/shm`" directory to check if they still exist.
- Since the sizes of types are platform dependent, use `sizeof` operator to properly allocate memory. Furthermore, in order to correctly point to a part of the shared memory, you may need to use type casting.
- To generate random sequences, you can use `srand(time(NULL) ^ (getpid()<<16))` and `srand(time(NULL) ^ pthread_self()<<16)`. Note that `srand()` and `rand()` do not have to be thread-safe. Be careful where you use them.
- As an example, to compile your client program, you can use this command:
```
gcc client.c -o client -pthread -lrt
```

- The connection diagram of the programs (for a single child process and thread pair):



- Example output:



```
ee442-64bit@ee44264bit-VirtualBox: ~/EE442_2017_HW2_example
ee442-64bit@ee44264bit-VirtualBox:~/EE442_2017_HW2_example$ ./server -q 5
Server program started at Fri Mar 31 17:38:28 2017
^C
Server program closed at Fri Mar 31 17:38:37 2017
ee442-64bit@ee44264bit-VirtualBox:~/EE442_2017_HW2_example$ 
```

```
ee442-64bit@ee44264bit-VirtualBox: ~/EE442_2017_HW2_example
ee442-64bit@ee44264bit-VirtualBox:~/EE442_2017_HW2_example$ ./client -t 15
Client program started at Fri Mar 31 17:38:31 2017

From server process 2174 to client thread 140647177987840 - Result: 562
From server process 2186 to client thread 140647228344064 - Result: 1470
From server process 2185 to client thread 140647236736768 - Result: 220
From server process 2187 to client thread 140647219951360 - Result: 858
From server process 2178 to client thread 140647194773248 - Result: 428
From server process 2180 to client thread 140647211558656 - Result: 1552
From server process 2183 to client thread 140647203165952 - Result: 1640
From server process 2184 to client thread 140647136024320 - Result: 226
From server process 2173 to client thread 140647186380544 - Result: 1456
From server process 2179 to client thread 140647169595136 - Result: 220
From server process 2175 to client thread 140647144417024 - Result: 266
From server process 2182 to client thread 140647127631616 - Result: 662
From server process 2177 to client thread 140647161202432 - Result: 1968
From server process 2176 to client thread 140647152809728 - Result: 306
From server process 2181 to client thread 140647119238912 - Result: 1142

Client program closed at Fri Mar 31 17:38:32 2017
ee442-64bit@ee44264bit-VirtualBox:~/EE442_2017_HW2_example$ 
```

Remarks:

1. You should insert comments to your code at appropriate places without including any unnecessary detail. Comments will be graded. You have to write to-the-point comments in your code, otherwise it would be very difficult to understand. If your output is wrong, the only way we can grade your homework is through your comments.
2. Send your homework compressed in an archive file with the name "e<student_ID>_HW2" (e.g. e1234567_HW2.tar.gz). The archive file should include your **source files** and **header file(s)** (if they exist).
3. Your work will be graded on its correctness, efficiency and clarity as a whole.
4. Late submissions are welcome, but penalized according to the following policy:
   - 1 day late submission: HW will be evaluated out of 70.
   - 2 days late submission: HW will be evaluated out of 50.
   - 3 days late submission: HW will be evaluated out of 30.
   - 4 or more days late submission: HW will not be evaluated.

Good Luck!