Task 03 - Shared Memory and Semaphores

1 Problem 1 [4 marks]

Write two programs, a `client` program and a `server` program, that exchange information via shared memory.

The server issues tickets (simply integer numbers) which are being read by the client, and the client subsequently notifies the server that it has taken the ticket. Both sides repeat this process 10 times and protocol what they have done at what precise time into a personal log file.

The implementation should be based on the following idea: the server process allocates some shared memory that holds a record with (at least) two fields: a field `bool isTaken`, a field `int ticket` and a field `bool soldOut`. It also holds a global ticket counter that starts with 1.

The protocol between the server and the client is as follows: The server "issues a ticket" by simply copying the global counter into `int ticket` and by setting both flags, `isTaken` and `soldOut`, to `false`. Thereafter the server increments the counter and waits for the client to "buy" the ticket.

The client first reads the `int ticket`, deliberates for a while, and then purchases the ticket by setting `isTaken` to `true`. Model the deliberation phase by inserting a random wait between reading the ticket and setting `isTaken` to `true`.

Once `isTaken` is set to `true`, the server repeats this action until the pre-defined maximum number of tickets has been issued. When that happens, the server sets `soldOut` to `true`.

When the client finds `soldOut` to be `true` it terminates.

To get started you may want to look at the template code we provided on vision. It contains a first cut on both, the server (in `server.c`) and the client (in `client.c`). It also contains a module `shm.c` with header file `shm.h` which is meant to provide nice abstractions for handling the shared memory and for producing appropriate error messages using `perror(...)`. As a starting point, you may want to look at the man pages for

- `shm_open(...)`,
- `ftruncate(...)` and
- `mmap(...)`.

Generate those two programs and run one instance each. Inspect your log-files to see what has happened. Make sure that both programs, the erver as well as the client protocol the adress of where the `ticket` is located. Are these adresses the same? Is your solution free of race conditions? Are you terminating both processes properly? If not, what needs doing to achieve this?

What happens if you run more than one client?

2 Problem 2 [4 marks]

Extend your previous solution by adding a semaphore to enable more than one client to be run at the same time. Introduce a semaphore module `sem.c` with header file `sem.h` to encapsulate the semaphore handling with proper error messaging. Use the semaphore to protect your entire shared memory. Make sure that any access to the shared memory, be it by the client or by the server, is turned into a critical region.

You may find the following man pages to be helpful:

- `sem_open(...)`,
- `sem_post(...)`, and
- `sem_wait(...)`.

Run one server and 5 clients, increase the number of tickets to 20 and inspect your log-files. Is your solution free of race conditions? Do you have some fairness?

3 Problem 3 [2 marks]

The above solution enforces all clients to wait while the client that has entered the critical region deliberates. Modify your above solution to improve that situation. The idea is to issue more than one ticket at a time and, thus, allow more than one client to deliberate at the same time.

Create 3 ticket channels, i.e. 3 pairs of a `ticket` variable and an `isTaken` variable and re-purpose the original `ticket` / `isTaken` pair to assign channel numbers to clients. Make sure that only the channel assignment is within the critical region but access to the three ticket channels can happen in parallel.