PA#5: Interprocess Communication Mechanisms

Due: 4/19/19 at 11:59pm

Introduction

Two programming assignments PA2 and PA4 have made heavy use of the `FIFORequestChannel` class, which was pre-written and given to you along with the dataserver code so that you could focus on learning synchronization techniques.

If you did look at them, you may have noticed that the `FIFORequestChannel` class uses a mechanism called "named pipes" or "FIFOs" to communicate between the two sides of the channel. However, FIFOs are only one of several different IPC mechanisms, each of which have their own particular uses that make them suited to particular applications. In this programming assignment, we are going to expand our toolbox by learning about 2 "new" IPC mechanisms in addition to named pipes: *message queues* and *shared memory*, where the latter would in turn use *Kernel Semaphores*.

Background

Message Queues

While pipes provide a byte stream between two processes, message queues allow for the exchange of messages between processes. There are library functions (POSIX library, not STL) for message queue opening/creation, sending messages, receiving messages, closing the message queue, deleting the message queue, and modifying the message queue's attributes. You may be able to use default attributes for this assignment, but those defaults vary by system. Visit the man pages for `mq_overview` (note that this is POSIX IPC, not the System V IPC, which is the older way) for how to check and set default message queue attributes.

Shared Memory

Up until now there have been IPC mechanisms to provide byte streams and message passing, but what if something a little more versatile is needed? Shared memory is exactly what it sounds like: a segment of memory that can be read and modified (depending on its configuration) by multiple different processes. You will notice that there are no system calls for reading and writing the shared memory segment. This is because a shared memory segment is semantically identical to any other memory segment, such as can be obtained from `malloc`, except for the IPC and synchronization considerations. One can read and modify it using `memset, strcpy, memcpy`, or just about any other memory-reading/writing operations. This brings in synchronization concerns between the writer and reader, which we will have to solve using Kernel Semaphores.

Visit the man pages for `shm_overview`. Again, note that you are required to use POSIX shared memory, not the System V version.

Kernel Semaphores

You are already familiar with the concept of semaphores from PA2 and PA4. However, note that semaphore was only "process-local" or "thread-shared," since it was allocated within the address space of a given process. What if two different **processes** needed to synchronize on the same semaphore, a "process-shared" or kernel semaphore?

For this programming assignment, part of what you will need to do is fill in the methods of the `KernelSemaphore` class. `KernelSemaphore` implements the familiar Semaphore interface but is distinct from the process-local semaphore from PA2 and PA4.

The Assignment

Code

You have to start off of your code from PA4. We are assuming that you have a working PA4. If that is not the case, please contact us for a working version of PA4.

You then have to make up 3 versions (i.e., really just 3 modes of running the same code) of your PA4 each using a separate IPC-method-based request channels: FIFO, message queue, and shared memory. Call these versions `PA5_FIFO`, `PA5_MQ`, `PA5_SHM`, respectively. You should have an abstract `RequestChannel` class and 3 sub-classes:

- `FIFORequestChannel`
- `MQRequestChannel`
- `SHMRequestChannel`

Here, the first one `FIFOReqeustChannel` would be taken "almost" directly from PA4 and you need to implement the others. The API of base class `RequestChannel` should be as follows:

```
class RequestChannel {
public:
    typedef enum {SERVER_SIDE, CLIENT_SIDE} Side;
    typedef enum {READ_MODE, WRITE_MODE} Mode;

    /* CONSTRUCTOR/DESTRUCTOR */
    RequestChannel (const string _name, const Side _side);
    virtual ~RequestChannel();

    virtual char* cread(int* ptr)=0;
    /* Blocking read of data from the channel. Returns a string of
    characters  read from the channel. Returns NULL if read failed. */

    virtual int cwrite(char* ptr, int len)=0;
    /* Write the data to the channel. The function returns
    the number of characters written to the channel. */
};
```

You will also need `KernelSemaphore` class for a properly functioning `SHMRequestChannel`, where the semaphoreś API should look like the following:

```
class KernelSemaphore {
private:
    /* INTERNAL DATA STRUCTURES */
public:
    /* CONSTRUCTOR/DESTRUCTOR   */
    KernelSemaphore (int _val);
    ~KernelSemaphore(); // make sure to remove all allocated resources

    void P();  /* Acquire Lock*/
    void V();  /* Release Lock */
};
```

0.1 Compiling and Running Your Code

There should be only 1 `makefile` to compile everything together.

Take an additional runtime argument option "-i" which should get one of:

- "f" for FIFO
- "q" for message queue
- "s" for shared memory

The following is an example command to run PA5:

```
./client -n <requests/person> -b <bounded buffer size>
                        -w <number of request channels> -i <f|q|s>
```

You must resolve the derived type of `RequestChannel` class in the runtime using `polymorphism` and `run-time binding` in C++.

0.2 Clean Up

You must clean up all IPC objects from the kernel memory and all temporary files you created for doing `ftok()`. Checking the `/dev/mqueue` and `/dev/shm` directories (or alternatively, running `ipcs` command in shell) will aid you in this clean-up process. In addition, you should clean all heap-allocated objects.

Report

- Gather timing data on the same set of n, b, w arguments on each of `PA5_FIFO`, `PA5_MQ`, and `PA5_SHM`.
- Present a performance comparison of the different IPC mechanisms based on this data, and attempt to provide explanation for any differences and similarities.
- Present the results in separate graphs using `PA5_FIFO` (i.e., PA4) as the baseline for comparison

- What is the maximum w and thus the max number of `RequestChannels` that you can use for each IPC? How much more can you go beyond the limit in `PA4`? (recall that the limit imposed by how many file descriptors each user can have.
- What are some of the limits encountered by each class, either due to the specific implementation or to operating system limitations, and how does the program behave when it encounters them?
- Describe the clean-up activity you have done for each IPC

Bonus worth 12 points

- Using only one MQ object for all the request channels: 5 pts
- Using only one shared memory segment for all request channels: 5 pts

What to Turn In

Turn in a single zip file `PA5.zip` containing the report, all the class files (separated into `.h` and `.cpp`) and a `makefile`. Note that the same `makefile` should build all request channel versions (FIFO, MQ and SHM).