PA#3: Interprocess Communication Mechanisms

Due: Sunday 3/21/21 at 11:59pm

Introduction

The goal of this PA is to use several IPC methods to enable client-server communication and compare their performance under large volume of data exchange.

Your PA1 used `FIFORequestChannel` class, which was pre-written and given to you. `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/new`, 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 ourselves 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

So far, we have used only parent-child synchronization using `wait()/waitpid()` functions where the parent can only wait for only the termination of the child or some other-other change of its state (e.g., stopped/continued). While this works for some applications, in other cases, you need more sophisticated form of synchronization, where:

- Two unrelated processes, outside family tree, should be able to synchronize with each other
- The involved processes can communicate about some event (e.g., some task completion). These communications can occur while both processes are alive, i.e., neither process has to terminate to send the signal to the other side

Kernel Semaphore is a synchronization method that supports the above features. Processes sharing memory using the shared memory technique do not know when the producer puts the data or when the consumer takes it, because that cannot be derived from the data itself. Such processes must use Kernel Semaphore to indicate both these events: data availability and data consumtion (i.e., buffer availability). Therefore, We will use Kernel Semaphore in conjunction with shared memory to enable a proper communication between two processes. Note that for a bidirectional communication you are going to need two sets of logistics: one set for clinet-to-server and another for server-to-client.

The Assignment

Code

You have to start off of your code from PA1. We are assuming that you have a working PA1. If that is not the case, please contact us for a working version of it.

You then have to make up 3 versions (i.e., really just 3 modes of running the same code) of your PA1 each using a separate IPC-method-based request channels: FIFO, message queue, and shared memory. Call these modes `PA3_FIFO`, `PA3_MQ`, `PA3_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 PA1 in the sense that you just need to to make it a derived class of `RequestChannel` without making any functional changes. After that you are going to add the other two classes. The API of base class `RequestChannel` should be as follows. Its structure reflects some basic principals of Object Oriented Programming (OOP). For instance, moving things common to all derived classes higher up in the hierarchy, leaving specific things to the lower/derived classes.

```
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() = 0;
    /* destruct operation should be derived class specific */


    virtual int cread (void* msgbuf, int bufcapacity) = 0;
    /* Blocking read, returns the number of bytes read.
    If the read fails, it returns -1. */

        virtual int cwrite (void* msgbuf, int bufcapacity) = 0;
    /* Write the data to the channel. The function returns
    the number of characters written, or -1 when it fails */
};
```

You must resolve the derived type of `RequestChannel` class in the runtime using `polymorphism` and `run-time binding` in C++. For instance, you cannot recompile your code if you want to switch from FIFO to SHM.

0.1 Compiling and Running Your Code

Modify the makefile to include compile commands for FIFO, MQ and SHM Request Channels. Make the `client.cpp` take an additional runtime argument option "-i" whose value would be one of the following:

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

The following is the command format to run PA3. The following command will create the specified number of new channels of the given type given by the "-i" argument, then request the first 1K ecg data points for the given person through each of those channels.

```
    ./client -c <\# of channels> -p <person no>
    -e <ecg no> -i <f|q|s>
```

And, the following command will get the specified file using the given number of channels:

```
    ./client -c <no channels to create> -f <filename>
     -i <f|q|s> -m <buffer capacity>
```

Note that for the above command, with number of channels c and the file size s, collect the first $\lceil s/c \rceil$ bytes through the first channel, the next $\lceil s/c \rceil$ bytes from the second channel and so forth; and each time using the given buffer size.

0.2 Clean Up

You must clean up all IPC objects from the kernel memory and all temporary files you created. For FIFO, check the current directory, and for MQ and SHM, check the `/dev/mqueue` and `/dev/shm` directories, respectively, to make sure that your clean up has worked correctly. In addition, you should clean all heap-allocated objects.

Report

- Gather the time to collect 1K data point each for $c = 5$ for instance under `PA3_FIFO`, `PA3_MQ`, and `PA3_SHM` and present the results to compare their performance.
- Repeat the previous with file transfers and again comare performance.
- Include the link to a video where you demonsrtate your work according the grading instructions to be published soon.

Bonus - 15 points

- Using only one shared memory segment for all request channels: 5 pts
- Using separate processes (i.e., by calling `fork()`) for each new channel to collect data or file messages. The result is the client, using all its c children processes can finish a task c times faster.

What to Turn In

Turn in a single zip file `PA3.zip` containing the report, all the class files (separated into `.h` and `.cpp`) and a `makefile`. The only exception is `RequestChannel` which can be in a header file. The `makefile` should build all request channel versions (FIFO, MQ and SHM).