PA4: Interprocess Communication (IPC) Mechanisms

Introduction

The goal of this programming assignment is to compare performance of various IPC methods during the exchange of large volumes of data between the client and the server.

PA1 used the pre-written `FIFORequestChannel` class that implemented a mechanism called "named pipes" or "FIFOs" to facilitate client-server IPC communication. In this programming assignment, we will be introducing 2 additional IPC mechanisms: **Message Queues** and **Shared Memory**. We will also be exploring the use of **Kernel Semaphores** in Shared Memory for synchronization. Each IPC mechanism has its own uses that make it suited to particular applications.

Background

Message Queues

While pipes provide a byte stream between two processes, message queues allow for the exchange of messages between processes. Both a reader and writer process must exist to enable a message to be written over pipes, but in message queues a process can write a message to the queue (if current queue size is less than max capacity) without having to wait for a reader process to connect. There are POSIX library functions for operations involving message queues: opening/creation, closing, deleting, sending messages, receiving messages, and modifying the message queue's attributes. You most likely won't be able to use default attributes for this assignment, but those defaults vary by system. Visit the man pages for `mq_overview` to check how to set message queue attributes. **Note: We are using POSIX IPC — the current Unix standard — not System V IPC, which is the older way.**

Shared Memory

IPC communication using FIFOs and Message Queues results in the information exchanged having to go through the kernel. If the server wishes to write to the client, the data written by the server must be copied into the kernel's IPC buffer. Similarly, when the client wishes to read, this data must again be copied from the kernel's IPC buffer into the client's buffer. How do we reduce or minimize this overhead caused by system calls and kernel interaction?

Shared memory is a segment of common memory that can be read and modified (depending on its configuration) by 2 or more processes. The changes made to the shared memory segment by any one process can be viewed by another process. A shared memory segment is semantically identical to any memory segment that can be obtained from malloc/new (with the exception of IPC and synchronization considerations). Any process can read and modify it using memory writing or reading operations such as memset, strcpy, and memcpy. This brings in synchronization concerns between the writer and reader, which we will have to solve ourselves using shared memory locking mechanisms like **Kernel Semaphores**. An important point to note here is that while shared memory segments may be efficient for the reasons mentioned above, the overhead associated with kernel semaphores is still large. The use of shared memory segments as an IPC mechanism then becomes useful when we wish to transfer large amounts of data. In such a scenario, the overhead associated with pipes/message queues is a lot more significant than the constant overhead caused by Kernel Semaphores (which does not scale with data).

We will use a pair of shared memory segments to facilitate bidirectional communication. 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

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

- Two unrelated processes, outside the process' 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.

Processes using a shared memory segment do not know when one process writes the data or another process reads it because this information cannot be derived from the data itself. These processes must use semaphores to indicate both data availability and data consumption. Semaphores are essentially integers whose value can never fall below zero. There are two functions of semaphores that we can use to facilitate synchronization: `sem_wait()` and `sem_post()`; `sem_wait()` decrements the value of the semaphore by 1 and `sem_post()` increments the value of the semaphore by 1. When the value of the semaphore is 0, a process calling `sem_wait()` will block till another thread calls `sem_post()` and the value becomes positive.

Let's say the client is attempting to read from a shared memory segment which the server will write to. How do we ensure that the client does not attempt to read from the shared memory segment before the write operation is complete? **Hint: We must initialize each of our semaphores with the correct initial value to ensure proper synchronization. The fact that the write operation is performed before a read should guide your choice in initializing your semaphore values.**

Also note that you must use 2 pairs of semaphores - one pair for each shared memory segment. The SHMQueue class in the starter code encapsulates a single pair of semaphores and a shared memory segment. You will need to implement the SHMQueue class functionality such that you can use 2 SHMQueue objects within the SHMRequestChannel to facilitate synchronization. We will use named semaphores in this programming assignment. Visit the man pages for sem_overview. Again, note that you are required to use POSIX semaphores, not the System V version.

The Assignment

Code

You will need to start based on your code from PA1. (We are assuming that you have a working PA1. If that is not the case, please contact your section's TA right away.) **DO NOT USE YOUR PA3 CODE.**

You then have to make up 3 versions (really just 3 modes of client-server communication) of separate IPC-method-based request channels: FIFO, Message Queue, and Shared Memory. You should have an abstract class `RequestChannel` and 3 derived classes:

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

Here, the first one (`FIFORequestChannel`) is taken almost directly from PA1 in the sense that it is made a derived class of `RequestChannel` without making any functional changes. Since the FIFORequestChannel is implemented for you in the starter code, you are not required to make any changes to this class. You are tasked with adding the other two classes. The `RequestChannel` structure reflects some basic principles of Object-Oriented Programming (OOP): we move constructs common to all derived classes higher up the hierarchy (e.g., `Side`, `ipc_names`, and the base constructor) and leave specific implementations to the lower/derived classes (e.g., the pure virtual functions `cread` and `cwrite`). The definition of our base class `RequestChannel` is as follows:

```
class RequestChannel {
public:
    enum Side{SERVER_SIDE, CLIENT_SIDE};

protected:
    std::string my_name;
    Side my_side;

public:
    RequestChannel (const std::string _name, const Side _side);

    virtual ~RequestChannel();

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

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

    std::string name();
};
```

Your code must select between the channel types based on command-line options. You cannot recompile your code if you want to switch from FIFO to SHM.

Compiling and Running Your Code

The `client.cpp/server.cpp` take an additional runtime argument option "`-i`" whose value would be one of the following:

- "`f`" for FIFO (default value)
- "`q`" for Message Queue
- "`s`" for Shared Memory

Note 1: You will have to extend the functionality of the "`-c`" argument to take in the number of new channels to create since, in PA1, it only served as a flag to make a new channel. The number of new channels *c* is in addition to the `control` channel (i.e., if $c = 5$, you'll have the `control` channel and five new channels). The `control` channel should not be used to request data points or transfer file segments when the c flag is provided. If no c flag is provided, use the `control` channel.

Note 2: When the "-c" flag is provided for a file transfer, you must distribute the work as evenly as possible across each channel. For example, given c channels and the file size *s*, you must collect $\sim \lceil s/c \rceil$ bytes through each channel, using the given buffer size for each file message (handling edge cases as in PA1). When the "-c" flag is provided for a multiple data point transfer, however, you must implement the multiple data point transfer once for every channel; the work is not shared, rather repeated across each channel. You must write one channel to one file, like x1.csv for the first channel, x2.csv for the second channel, etc.

The following command will create the specified number of new channels of the given type by the "`-i`" argument, then request the first 1K ECG data points — of the format (time, ecg1, ecg2) — for the given person through each of those channels:

```
./client -c <# of new channels> -p <person no> -i [f|q|s]
```

And, the following command will get the specified file using the given number of channels of type given by "`-i`":

```
./client -c <# of new channels> -f <filename> -i [f|q|s] -m <buffer capacity>
```

The following command line options are valid for this programming assignment: *i, p, t, e, f, m, c*. The options *p, t, e, f, and m* have the same functionality as PA1. The following combinations of command line arguments are possible for PA4:

- *p, t, and e* [single data point transfer with optional *i* parameter]
- *p* [multiple data point transfer with optional *i, c* parameters]
- *f* [file transfer with optional *i,c,m* parameters]

Clean Up

You must clean up all IPC objects from the kernel memory and all temporary files you created on process exit. 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.

Note: If the /dev/mqueue folder does not exist on your system, you can follow the instructions on the man pages under the "Mounting the message queue filesystem" header to mount this folder.

Successful Completion of the Assignment

Code Requirements [85 points]

- [10 pts] Make necessary changes to client.cpp/server.cpp to allow use of multiple channels and choosing IPC method. This should be done first using the already implemented `FIFORequestChannel`.
  - Points may be deducted for not properly dividing a file transfer across all data channels
- [50 pts] Fulfill the requirements of each derived IPC class in `MQRequestChannel`, and `SHMRequestChannel`.
  - [20 pts] `MQRequestChannel`
    - Points may be deducted for failure to use mq_attr struct
  - [30 pts] `SHMRequestChannel`
- [25 pts] Secret tests

Report [15 points]

- [5 pts] High-level description of your code design and implementation (½-1 page).
- [3 pts] Gather the time to collect 1K data points across $c = 5$ channels for each instance of IPC method and present the results to compare their performance.
- [3 pts] Repeat the previous with file transfers for a 10MB file across $c = 50$ channels and again compare performance.
- [4 pts] You do not need graphs for your timing comparisons (tables are fine), but please discuss them (1-2 paragraphs).

Assignment Logistics

GitHub Classroom

This assignment will be hosted using the **GitHub Classroom** platform. You must visit https://classroom.github.com/a/B5Qz8kM3 to create a repository for your project.

What to Turn In

Once you have completed the assignment and submitted your final code to GitHub Classroom, on Canvas, turn in a text document named *FirstName_LastName_UIN.txt* containing the link to the commit hash that you are submitting (PA1 example below) and your report named *PA4_FirstName_LastName_UIN.pdf*.