Project 2 – IPC

CS6210 – Advanced Operating Systems

Lecturer: Dr. Ada Gavriloska

Completed by Nathan Hicks

Table of Contents

[Introduction 3](#_Toc507795653)

[Basic Service Performed 3](#_Toc507795654)

[Protocol for Server/Client Interaction 3](#_Toc507795655)

[Client Registration with the Service 3](#_Toc507795656)

[Service Acknowledgement of Registration over Client’s “Receive” Queue 3](#_Toc507795657)

[Client Writes to Shared Memory and sends “Caesar” instruction to Service 3](#_Toc507795658)

[Service Encodes/Decodes message written by Client and Sends “Fin” to Client 3](#_Toc507795659)

[Client Receives Fin, Reads Processed Message from Shared Memory and Outputs to Stdout 4](#_Toc507795660)

[Client’s API with Service 4](#_Toc507795661)

[Shared Memory Structure 5](#_Toc507795662)

[Description of the Service Program 5](#_Toc507795663)

[Message Queue for Synchronization 6](#_Toc507795664)

[Shared Semaphore Mutex around Shared Memory Access 6](#_Toc507795665)

[Signal Handling 6](#_Toc507795666)

[Description of the Client Program 6](#_Toc507795667)

[Implementation of Quality of Service (QoS) 7](#_Toc507795668)

[Performance 7](#_Toc507795669)

[Partial Asynchronous Functionality Implementation 8](#_Toc507795670)

[Sample Program Execution 8](#_Toc507795671)

[Server Output 1 (without QoS): 8](#_Toc507795672)

[Client Output 1 (without QoS): 9](#_Toc507795673)

# Introduction

The service I designed implements a Caesar Cipher encoder/decoder. To implement IPC, I used the POSIX API. The service runs using a known shared memory object in /dev/shm and a known registration message queue in /dev/mqueue. Changes to the shared memory is protected using a single mutex semaphore shared by both server and client (also in /dev/shm). More details are covered in [description\_of\_the\_service\_program](#description_of_the_service_program).

# Basic Service Performed

At its most basic level, a client that sends a message, ‘hello’, with a shift value of ‘2’ would receive ‘jgnnq’ back from the service. If the client then sends ‘jgnnq’ with a shift value of ‘-2’, it would receive its original ‘hello’ message from the service. For an example, see the [Sample Program Execution](#sample_program_execution) at the bottom of this writeup.

# Protocol for Server/Client Interaction

## Client Registration with the Service

The service enters the main event loop by listening to a “client registration” message queue in blocking mode. You can consider this message queue to be operating in half-duplex for all intents and purposes, because the service never sends any replies on this queue – it only receives registrations from clients. The client sends a string (i.e. “client1”) to the service over the registration queue, and the service uses that string to build two additional message queues specifically for that client (a send and receive queue). The names of the two queues are concatenations of the string sent by the client and two strings known to both service and client.

## Service Acknowledgement of Registration over Client’s “Receive” Queue

When the service has acquired a client’s identifier from the registration queue, it builds the two queues mentioned previously and sends an “ack” message to the client on its receive queue. Of course, once the client registered with the service, it immediately enters a listening state on its receive queue to receive the “ack” from the service.

## Client Writes to Shared Memory and sends “Caesar” instruction to Service

Once received, the client writes its data to shared memory and sends the instruction, “caesar”, to the service over its send queue. The service is already listening on the client’s send queue for this instruction as soon as it finished sending the ack on the client’s receive queue.

## Service Encodes/Decodes message written by Client and Sends “Fin” to Client

When the service receives the “caesar” instruction, it encodes or decodes the message string in shared memory by the amount specified in the shift value. A positive shift value moves the string’s characters *shift* values to the right, and a negative shift value moves the string’s characters *shift* values to the left. Once it returns from the rotation function, it sends a “fin” message to the client on its receive queue.

## Client Receives Fin, Reads Processed Message from Shared Memory and Outputs to Stdout

The client receives the “fin” message, reads from shared memory, and prints out the result.

Figure 1 below summarizes how the interaction between client and service works.

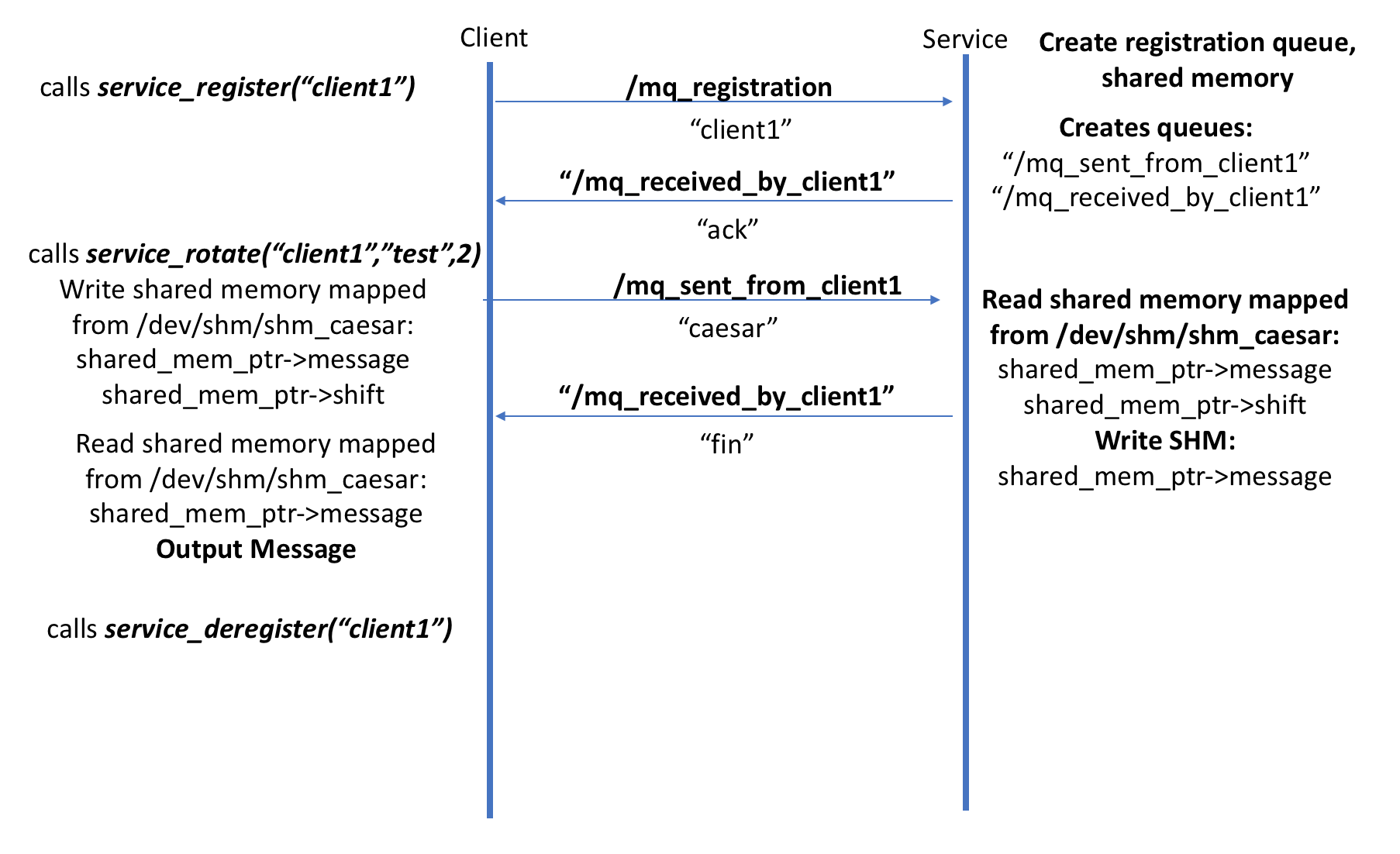


Figure 1 The protocol interactionl between Client and Service

# Client’s API with Service

/\*\*

\* service\_rotate() - request caesar encode/decode from caesar service

\* @client\_q\_name: The base name of the client

\* @message: a character array containing the message to be encoded/decoded

\* @shift: a positive or negative direction to shift the message, where

\* positive values shift the message right, and negative values shift

\* the message left.

\*

\* Implements protocol following initial client registration.

\*

\*/

void service\_rotate(const char client\_q\_name[], char message[], int shift);

/\*\*

\* service\_register() - register client queues with service

\* @client\_q\_name: The base name of the client

\* @priority\_arg: defaults to 0, priority provided optionally as a command-line argument

\*

\* Implements registration protocol with service by first sending the client base name,

\* and waiting for an "ack" from service.

\*/

void service\_register(const char client\_q\_name[], int priority\_arg);

/\*\*

\* service\_deregister() - deregister client queues

\* @client\_q\_name: The base name of the client

\*

\* Calls mq\_unlink on the send and receive queues associated with the base name

\*

\*/

void service\_deregister(const char client\_q\_name[]);

# Shared Memory Structure

struct shared\_memory {

char message[BUFSIZE+1];

int shift;

};

# Description of the Service Program

The service program is designed to encode and decode arbitrary strings that have standard ASCII encodings in the uppercase and lowercase English alphabet. The program will run without arguments as a foreground process, or optionally will run as a daemon process by passing the “-d” flag as an argument to the program. It will also provide “—help” and “—version” responses. If not running as a daemon, it will print output of the program to stderr and stdout. Different types of output are color-coded in Linux environments using terminal control and output formatting.

Example commands as follows:

bin/caesar\_service --help

bin/caesar\_service --version

bin/caesar\_service

bin/caesar\_service -d

The process begins by creating a POSIX shared memory object at **/dev/shm/shm\_registration** and mapping that to virtual address space. It also creates a registration message queue at **/dev/mqueue/mq\_registration** so that clients can register with the service. The registration queue acts as the initial queue that clients must wait in until the service can process their requests.

## Message Queue for Synchronization

The message queues used by the service program implement a form of synchronization primitive that forces the registered client and service processing requests to adhere to the protocol. This is because “half-duplex” message queues are used with each side having to wait on a receive before proceeding.

## Shared Semaphore Mutex around Shared Memory Access

The service also implements a single semaphore around shared memory access at **/dev/shm/sem.sem\_mutex** that acts as a mutex to prevent memory corruption during asynchronous operations.

After shared memory and the registration queue are ready, the service process enters an indefinite event loop that immediately goes into a blocking call to mq\_receive() listening for incoming messages from the registration queue. Once a client registers with an identifier for its message queues, the service process will process that client’s request, and then return to check for another message on the queue, or simply wait for one to come in.

## Signal Handling

Due to the fact that the service runs in an indefinite event loop, it registers a function called clean\_up() to handle SIGINT signals. This makes it easy to clean up shared memory object and message queues objects from /dev/{shm, mqueue} when you hit CTRL+C to stop the program.

# Description of the Client Program

The client program is very simple. It takes at least three arguments to the program that are described if you call **bin/caesar\_client --help** .

A standard call to the program would look something like this:

$ bin/caesar\_client -m hello -s 2 -q client1

A QoS’d client would be implemented as follows:

$ bin/caesar\_client -m hello -s 2 -q client1 -p 5

These are the obligatory UNIX arguments:

$ bin/caesar\_client --help

$ bin/caesar\_client --version

The message (argument that follows the -m flag) can be any arbitrary ASCII string up to 255 characters. The message buffer size is 256. The shift value can be any positive or negative integer of size int. Realistically though, the numbers should be within the span of a 26-letter alphabet. Any values in your message that are not uppercase or lowercase letters will be ignored by the service’s Caesar function. The -q argument takes a name up to 239 characters because the max size of a queue’s name according to the POSIX standard is 255 characters, and the value that will be appended to this base name is between 14 and 16 characters. The -p argument is the priority that allows you to implement QoS on the client registration queue.

# Implementation of Quality of Service (QoS)

Quality of service is implemented naturally using priority values set when a client registers over the registration queue. These values can be fed from the CLI as an argument to the program using the -p parameter. For example, a client that registers with the service using a priority value of 10 will be processed before a client that registered with a value of 0.

# Performance

The shared memory access is by far the fastest performance of all the IPC mechanisms in this program. The semaphores are also implemented as shared memory and thus share similar performance in terms of speed of access. However, due to the organization, priority, and overhead associated with message queues, they perform noticeably slower than pure shared memory accesses.

Running *time* measurements on the client produce the following results:

$ time bin/caesar\_client -m hello -s 2 -q client

real 0m0.006s

user 0m0.000s

sys 0m0.000s

# Partial Asynchronous Functionality Implementation

The use of a shared POSIX binary semaphore as a mutex around shared memory access between service and clients makes it safe for asynchronous use, and the use of half-duplex message queues for each client prevents messages from being misdirected. Thus, shared memory and messaging is safe for asynchronous use.

However, the program does not yet support a situation with multiple clients interleaving protocol commands to the service. The reason for this is that the service takes whichever client registered with the highest priority from the registration queue and then processes its encode/decode request all at once before returning to the registration queue to process the next highest priority in the client registration queue.

In order to modify the program for perfect asynchronous use, I would need to set all message queues for non-blocking and have the client and services poll each other in a loop until each side receives the next step in the protocol required. This would obviously require the service to track state (progress) in the protocol for each client.

That being said, if multiple clients were to run at the same time, this would work just fine. Each client would be processed either in the order they arrived (FIFO with a priority value of 0) if they didn’t provide a priority argument to the program, or the highest priority value at registration first.

# Sample Program Execution

## Server Output 1 (without QoS):

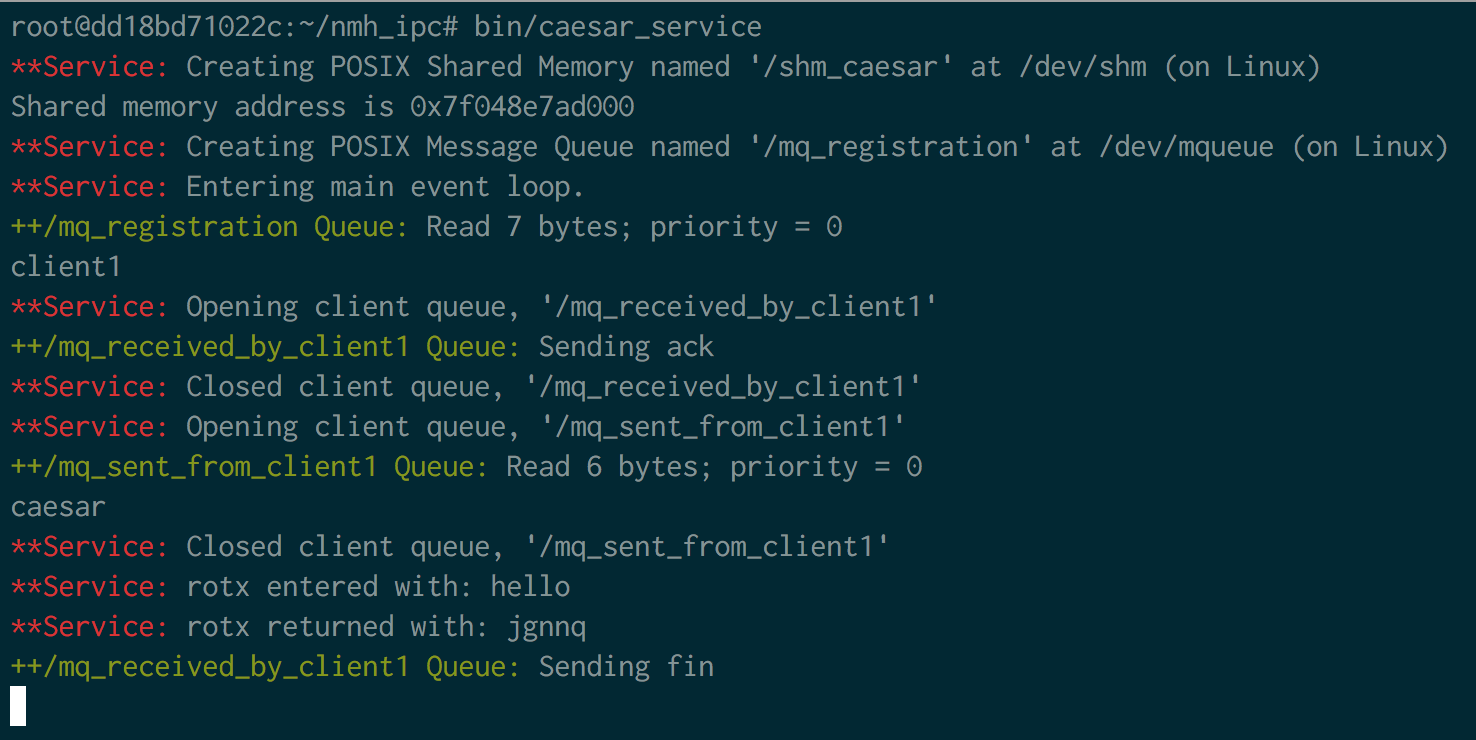


Figure 2 A sample output from the service after receiving input from the client

## Client Output 1 (without QoS):

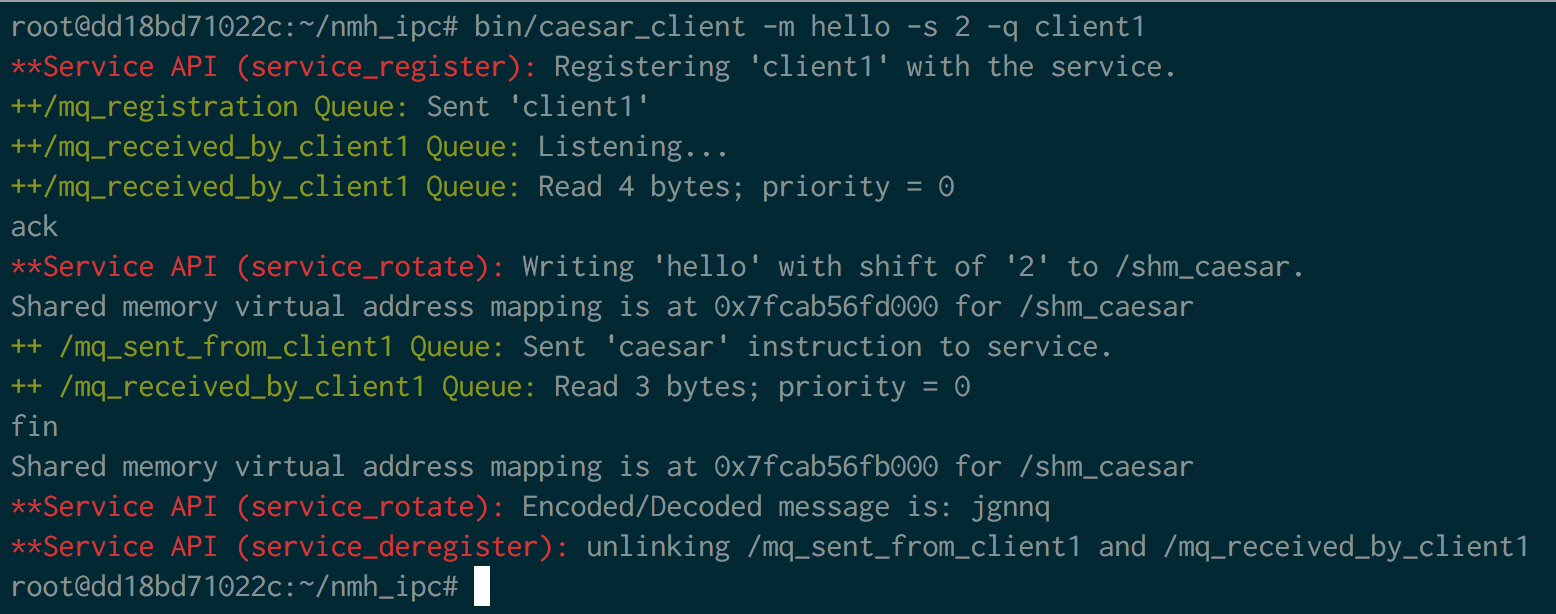


Figure 3 The client program's argument and output that generated the service's output in Figure 2