Inter-Process Communication Services

CS 6210 Advanced Operating Systems-Project 2

By Sharmila Raghu and Gayatri Singh

1 Table of Contents

2 Introduction

This assignment creates an application which allows multiple client processes to request a server process to send some bytes via shared memory so that the returned bytes can be compared to bytes generated by a function within the address space of the client process.
This is intended as one component of a secure communication application between processes.

3 Design

3.1 High Level Architecture

Figure 1 summarizes the main entities involved in the Inter-Process communication application constructed for this assignment.

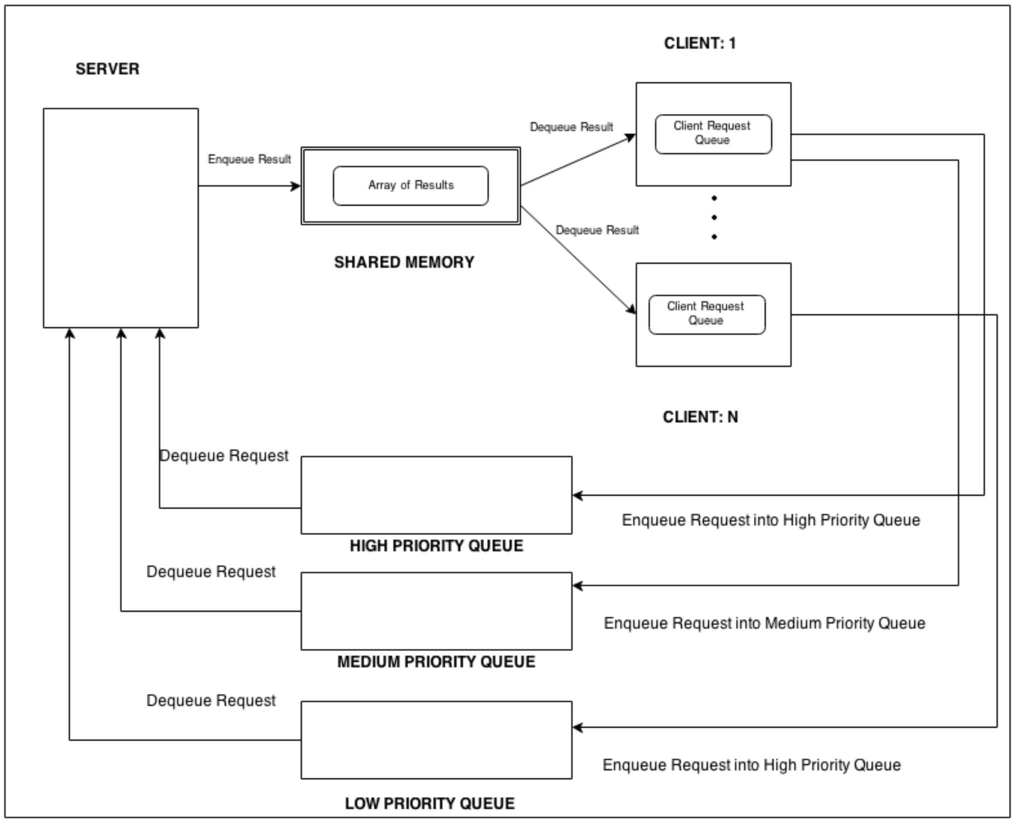

Figure 1: High Level Architecture Diagram for Inter Process Communication

It shows that inter process communication between the server and multiple clients is facilitated through the use of:

1. Message queues of differentiated priority: These pass potentially multiple requests from clients to the server and arbitrate a quality of service mechanism.
2. Shared Memory for Results: These are the location through which the server process communicates its generated result bytes to the target client. The size of this shared memory is dynamic and based on the number of requests the client anticipates it will make to the server in order to get its required results. Each client has its own shared memory with the server through which the results are communicated.

3.2 Detailed Design

3.2.1 Server API

The server API provides the following functions:

Function	Description
initializeServer()	This function creates the data elements used in the private address space of the server as well as the shared message queues
requestSchedule()	This function creates an infinite loop through which the server process continuously polls the low, medium and high priority queues for another request to service. When another request is found, it is sent to the helper function req_handler()
req_handler()	This function retrieves from the request parameter given to it, the shared memory id and shared memory semaphore id of the client who has made the request. The server can use this information to find and attach to the shared memory in which the client expects the returned bytes. A From this function, the server calls the returnBytes() function in order to get the bytes which it then places in the shared memory.
returnBytes()	This function returns some random bytes.

3.2.2 Client API

The client API provides the following functions:

FUNCTION	DESCRIPTION
initializeClient ()	Initializes the client with the required parameters. It is also used to create requests and assign values to it. The assigned request is also inserted into the client's private request queue.
initializeRequestParam()	Initializes the request structure with the necessary values.
callServiceAsync()	It locks the server queues . While in the locked state, the process removes the requests from its private request queue and adds it to the server queues based on the assigned priority.
callServiceSync()	It locks the server queues. While in the locked state, the process removes the requests from its private request queue and adds it to the server queues based on the assigned priority. It also calls waitsforResults() function to immediately get the results.
waitForResults()	It maps to the shared memory were the results are stored. It collects and adds all the values for its respective requests, after the server releases the lock. If the lock is not available the client waits. Finally the shared memory is detached and removed and also returns the calculated value to the main function.
doSomething ()	Client loops through and finally returns a random number to the main function, from were its been called.
useBytes ()	Compares the two parameters and returns whether it is equal or not.

3.2.3 Inter Process Communication Data Structures

The communication between the server and clients is facilitated by

- Message queues
- Shared memory

Both of these constructs have synchronized access through the use of semaphores.

3.2.3.1 Message Queue

The message queue transits request parameters between clients and servers in a FIFO manner.
The **request parameters** are C structs that register a client's intention to request the service for some bytes. They contain the following information to be used by the server:

- Request id
- Process id
- Shared memory id
- Shared memory semaphore id
- Total number of requests by client process

3.2.3.2 Shared Memory Result Location

Each client creates a shared memory location the size of the number of requests that it has for the server. It passes the shared memory id to the server via the request parameter so that the server knows where to drop its result bytes in response to a request from the client.

The shared memory is detached and destroyed once its purpose is served.

3.2.3.3 Semaphores

Each shared memory queue and each shared memory result location has an associated semaphore. The semaphore is a construct that allows synchronized access to these shared resources by multiple processes. Each process must lock the semaphore before they can access the resource, and they must release the lock when they are done with the shared resource.
Semaphore locking is not required to read from shared memory, however it is required to write to shared memory. The reason this works in our case is that:

1. Shared Memory is only read by the client process when it wants to receive results. The client process will poll the shared memory in order to confirm whether the bytes have been placed by the server process before collecting the results. The server process should not be blocked from writing results when the client is polling.

2. Shared Memory is written to by the client process when it is initializing it to a value that allows it to realize when the server has placed bytes on the shared memory.
3. Shared Memory is also obviously written to by the server when it is returning result bytes.

4 Implementation

4.1 Application

This application allows processes to compare that the bytes generated by the service process is the same as the bytes generated by it. When this is the case, the client process can trust the service process.

The service process and client process should have a former agreement on how to derive the encryption bytes for this security mechanism.

4.2 Synchronous Calls

The synchronous calls have been implemented as follows:

- The client process puts request parameters on the service queue corresponding to its priority.
- The client process waits on the server process to return the requested result bytes.
- The client process eventually gets the result bytes after waiting some time.
- The client process goes to the doSomething() function in order to get bytes from its private address space.
- The client process uses the result bytes for the purposes of the application (ie compare whether the result bytes from the server matches bytes generated within the client process' private address space).

4.3 Asynchronous Calls

The asynchronous calls have been implemented as follows:

- The client process puts a request parameter on the service queue corresponding to its priority.
- The client process goes to the doSomething() function in order to get other bytes from within its private address space in the time the service process will take to respond with results.
- When the client process has finished with doSomething() it goes to wait on the server process to return the requested result bytes.
- The client process eventually gets the result bytes (possibly after some time of waiting on the server process).

- The client process uses the result bytes for the purposes of the application (ie compare whether the result bytes from the server matches bytes generated within the client process' private address space).

4.4 Quality of Service

Quality of Service is implemented through three types of priority queues. One has low priority, the other has medium priority and the final has high priority.

The server process will handle 1 request at a time from the low priority queue, before moving to the medium priority queue from which it will handle 2 requests sequentially, and then it will move to the high priority queue from which it will handle 3 requests sequentially. The server will repeat this cycle.

The consequence of this is explained through the following example. If there is one client process A that has 5 requests and 1 client process B that has 1 request, and if we schedule the client process B behind the client process A, in a single queue then the client process B will have to wait an unfairly long time. Instead, if we were to put the client process B in the high priority queue and the client process A in the low priority queue, then the client process B will get access to the service sooner, since the process A will no longer be receiving continuous service after 1 of its requests have been serviced and other client processes such as B will get a chance.

The priority of requests is determined at the client process level. That is, all requests coming from the same client process have the same priority level.

5 Evaluation

5.1 Test Case

Our test cases are specified in the below table

Client ID	Sync or Async	Num Requests	Priority
0	Sync	1	Medium
1	Async	1	High
2	Sync	2	Low
3	Async	3	Medium

5.2 Test Results

Client ID	Sync or Async	Num Requests	Priority	Time Taken
0	Sync	1	Medium	15 seconds

				and 99 milliseconds
1	Async	1	High	14 seconds and 43 seconds
2	Sync	2	Low	74 seconds and 7 milliseconds
3	Async	3	Medium	31 seconds and 69 milliseconds

6 Conclusion

Our results show that asynchronous inter-process communication is faster than synchronous inter-process communication.

Furthermore, it is observed from the results obtained that the usage of 3 priority queues helps in fair provision of the service to clients across multiple requests.