CS4210/CS6210 Project 2

Project2: Inter-process Communication Services

Due: Friday March 1, 2019 @ 11:59 pm

1. Goal

Your goal in this project is to create a service "domain" that can be accessed from any process on the system. In order to support this service, you will be creating a library of calls to access this service. This service must allow blocking (synchronous) calls, non-blocking (asynchronous) calls, and QoS mechanisms. You may work in pairs on this project. You may share ideas with other pairs, but you may not share code. You also may not download code or use any other external libraries without consulting the instructor or TA first.

To get full credit for this project you must implement the following:

- Blocking service calls;
- Non-blocking service calls
- QoS service - possible extra credit, look at Section 7.
- An application demonstrating your functionality
- A write up discussing your design and the performance of your inter-process communication facility.

Each will be discussed in turn.

You must submit your deliverables by 11:59 pm on Friday, March 1st, 2019.

2. Overview

In Xen, there is a domain called Dom0 which, among other things, handles requests from operating systems for performing I/O. The operating systems place their data for the operation (e.g., a file write) on a ring buffer, and Dom0 handles the requests in turn. You will be creating an analog of Dom0 which will handle requests for a service. Specifically, your task is to implement a shared memory based file compression service (TinyFile) and a client library.

- Clients submit file name, file-content, etc. to the TinyFile service over shared memory.
- TinyFile service will respond to clients with, compressed version of the file.
- Original file content and the compressed file content (and any other service data) must be exchanged between clients and service via shared memory.
- The TinyFile service is capable of handling any kind of file content. (e.g. text, binary, etc)
- TinyFile service supports synchronous calls - caller has to block until the call is complete and the result is delivered.
- Finally, TinyFile service supports asynchronous calls - the caller will call the service and then continue to run while the service request is being processed. The asynchronous call must have some method to deliver the result back to the caller while the caller is still executing, or potentially for the caller to retrieve or even wait on the results, once it has nothing else to do.
- Optionally, TinyFile service may support Quality of Service (QoS) mechanisms which implement fairness among the processes.

3. Synchronous (Blocking) Communication

Your job is to build a library for accessing the TinyFile service. This library will consist of:

- An API for sending requests and getting back a response (in the same function call)
- A library (or at least object files) that your applications can link in
- A service process that will manage the input requests and send back a response.

The library you write will need an initialization function that establishes the control structures that you may want to use in your library - queue(s), any common shared memory region you want to use in your library, and so on. Next, you will need functions for your library that allow blocking (synchronous) calls to the TinyFile service. Here, a blocking/synchronous call means that the sender will block until the caller has gotten a response from the service.
Your design must incorporate the concept of a service process that actually executes the service.
The service process can use one or more queues on which the requests are queued up; how many you choose to use is left to your design, but must be justified. Note that you will need to use shared memory for all of the data that is shepherded back and forth. The service process also can be started as a daemon if you wish, which also means that the service process can be started as a separate process from the command line (which will almost certainly be needed).
Since you have freedom in your design, you must document the design itself, any logic as to why you chose it, any shortcomings that the design may have, and any part of your design that you failed to implement.

In order to get full credit for this section, you must create the synchronous TinyFile service, the library & API to support the service, and write up its design as described.

4. Asynchronous (Non-Blocking) Communication

There are times when a process will send a request, but it does not want to wait for the response. This form of request is called asynchronous because the process does not have a specific time ordering with respect to its computations and the receipt of a response. At first glance this may seem easier than synchronous (blocking) communication because the sender does not have to wait for a response, but that is not the case. The service process must still potentially need to know when a response has been received back by the calling process - there may be structures (including the response itself) that must be cleaned up once the response has been received, or certain application logic may start up or continue once the recipient has acknowledged the response.
For this part of the project you will need to implement asynchronous communication. You have a choice as to how the service process can choose to determine that a response has been delivered. The first choice is to implement a blocking function call in the caller that at a later time retrieves the result and initiates the cleanup. The second choice is to let the caller register a callback function that is called by the service process when a response is ready. This also requires that the callback function be called with a reference to the original response. In either case, you will need to think about how to notify the calling process that the request is complete and to deliver any data in the response.
The library you implement must have some way to specify a blocking call versus a non-blocking call. Ideally, it should be possible to have both uncompleted synchronous and uncompleted asynchronous calls pending completion at the same time. As before, you have a lot of latitude on your design and you must document it, its shortcomings (if any), and shortfalls in implementation.

For full credit on this section, you must implement the asynchronous facility and write up its design as previously described.

5. Quality of Service

So far the TinyFile service allows any process to use it, but it has no concept of fairness. For example, suppose in 1 second that process P1 first sends 100 requests, then process P2 sends 10 requests, and then process P3 sends 2 requests. If all of process P1's requests are answered first, then the service is denied to processes P2 and P3. Alternately, it's not clear how many requests of P1 should be serviced before P2's and P3's requests are serviced.
Here you will be developing a Quality of Service (QoS) mechanism for the TinyFile service. The mechanism must take fractional access in to account. For this, the service process alternates servicing requests among processes according to the fraction of the service that they've been granted. For example, if there are 4 processes that are each granted 25% of the access, then the QoS mechanism would essentially round-robin access among the 4 processes. In another example, process A may be granted 1/2 of the service requests, process B may be granted 1/3 of the service requests, and process C may be granted 1/6 of the service requests. Then over the course of 6 messages the service process will handle 6 * 1/2 = 3 of A's messages, 6 * 1/3 = 2 of B's messages, and 6 * 1/6 = 1 of C's messages (though not necessarily in that order). Designing this QoS mechanism is also one of your tasks.
You may choose to implement a single queue that the service process must re-examined to determine what message to process next, or you may choose to implement multiple queues - say, one queue per communicating process. Since there is such freedom in choosing your design, you must document it, the reasons why you chose it, any shortcomings it may have, and what in your design was not implemented.

For full credit, you must implement and document the QoS mechanism as described.

6. Sample Application and TinyFile service

Your sample application must exercise all the functionality of your library and service - synchronous requests, asynchronous requests and QoS, must all be demonstrated. The service process implements a file compression functionality. Coupled with QoS functionality, this non-trivial computation kernel allows you to demonstrate the queue management of your service process.

- Use the snappy-c library to implement file compression functionality. (https://github.com/andikleen/snappy-c)
- The service program/ application should accept the file path and list of filenames as a runtime parameter.
- The number and size of the shared memory segments should be a configurable runtime parameter.
- The application should be able to handle file sizes up to 1MB
- The client processes should accept as a parameter a list of one or more file names that will be requested from the service process (ok to send requests for the same file(s)).

You will receive full credit for this section if your application meets these criteria. You may be asked to demo your application to show that it works.

7. Extra Credits

QoS implementation is optional for 6210 students working alone on this project and 4210 students. However, can get extra credit (10pts) if implemented. For all 6210 students who work in pairs, it is mandatory for full credit.
Students can also receive extra credits if they implement some more creative ring-like structure and notification mechanism, one that's present in Xen.

8. Documentation/Write Up

Finally, your documentation/write up needs to describe your design and implementation. It should describe your API, your TinyFile service process internals, and anything else that is significant in your project. If there is missing functionality or functionality that doesn't work as required, then it must be documented. You will receive full credit for this section if these requirements are met.

9. Hints and Tips

First, you will need some sort of queue to send messages to another process, and you will need to lock access to that queue. For that it is recommended that you check out the POSIX library's message queue and semaphores. The message queue is available via functions that start with "msg" - msgget, which creates the queue; msgsnd, which places a message on a queue; and msgrcv, which pulls a request off of a queue. The semaphore access is similar, with semget being the primary function of interest.
You should also become familiar with shared memory functions such as the following:

- **shmget,** which creates a file for shared memory use OR gives a handle (called a key) to access it;
- **ftok,** which maps a filename to a key for use in shmget;
- **shmat,** which allows (attaches) a piece of shared memory to the calling process's address space;
- **shmdt,** which removes (detaches) a piece of shared memory from the calling process's address space.

You can also consider the following memory mapping functions, which can be a replacement for shmget:

- **shm_open,** which creates a file for shared memory use OR gives a handle (called a file descriptor) for accessing it;
- **mmap,** which maps a file descriptor taken from shm_open to a piece of memory;
- **munmap,** which unmaps the same file from the process's memory;

What makes the shmget/shmat more attractive than shm_open/mmap is that they are part of the POSIX library, which also has access to the potentially useful semaphore and message queue functionality. These calls basically replace "shm" with "sem" (for semaphore) and "msg" (for message queue). What you use is up to you and your partner. **Any other external library must be cleared by the TAs or the instructor first. You must implement in C/C++.**

Here are some general hints and tips:

- **Start early.** Your midterm comes right in the middle of this assignment, but you should at least find a partner as soon as possible.
- **Work incrementally.** Do the synchronous IPC piece first, then the asynchronous facility, finally to QoS option. If you're unfamiliar with shared memory, then write a couple of simple memory sharing programs just to get your feet wet before you start trying to get your message passing facility together.
- **Work in parallel when needed.** You will have a partner for this project, which means that some of the time you will be able to develop in parallel. Note that there will be times that you will need to decide on design elements together, such as what your API will look like, what your internal structures look like, etc.
- **Talk to others.** You can get ideas for your design by talking to other groups. Sometimes others have already gotten past an issue that you're still having. If someone helps you in this way, please post it to the class Piazza forum to pass it on to others. **No matter what you do, do not copy or agree on code together with other groups.**

10. Useful Resources

- Brecht et al, "Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O", EuroSys 2006
- Shared Memory slides in the repo
- API reference in the repo

11. Late Policy

A penalty of 5% per day will be applied for late submissions for up to 5 days. Submissions received more than 5 days late will receive no credit.

12. Submission

Please include all reports, Makefile, etc., in a directory and compress it. If you worked in a group, one can submit the compressed files and the other one can submit a text file with the names of the group members.