CS 6210 Advanced Operating Systems
ASSIGNMENT 2: Inter-Process Communication Services

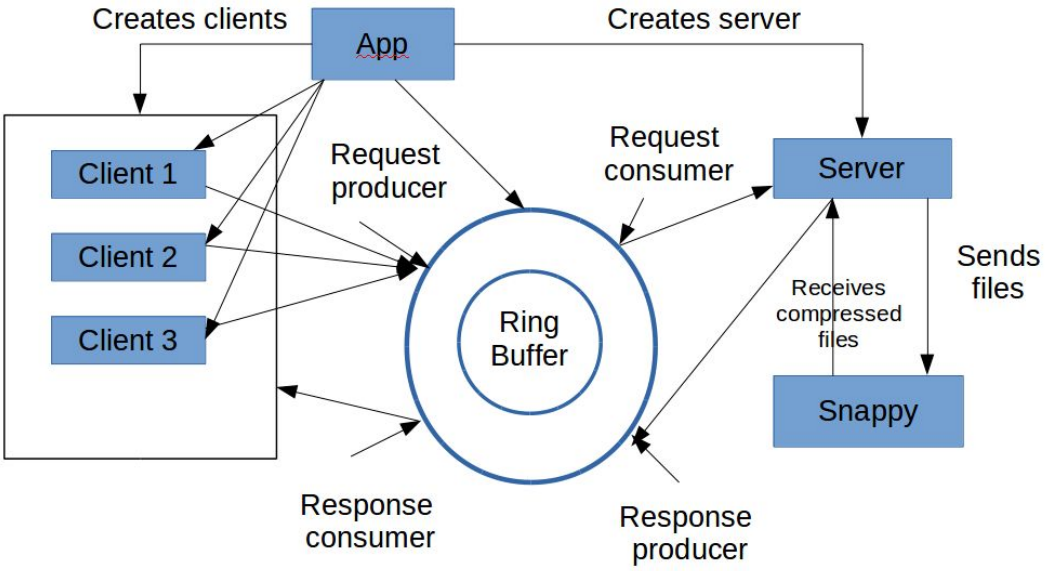
NAME: NIHAR MEHTA
GT USERNAME: nmehta80

NAME : KHUSHHALL CHANDRA MAHAJAN
GT USERNAME: kmahajan32

A. Design and Implementation

The design involves a master application "app.c" that creates a server thread and multiple client threads. This application takes file path and file names as required parameters and number and size of shared memory segments as a configurable runtime parameter. So, the code can be executed using "**./app . sample1.txt sample2.txt sample3.txt num_segments 500 size_segments 1000**" .

The master application first initializes the shared memory to be used for Inter-Process communication (IPC) between the clients and the server. It flushes the memory first and then reassigns it to ensure a fresh communication. Four pointers are defined for this memory globally - Request Producer, Request consumer, Response Producer and Response Consumer. Request producer and Response Producer are used by both server and client. Only the client uses Response Consumer whereas only the Server uses Request Consumer. Since the request producer and the request consumer functioning is independent of response producer and response consumer, the shared memory is divided in half and each individual halves are considered as IO RIng Buffers for requests and responses respectively. Two global locks are defined for Request Producer and Response Consumer pointers. The server and client creation, as well as communication, varies depending on whether we have synchronous or asynchronous communication.



1. Synchronous Communication

The master first creates the server. Then it reads all the files from the input and creates the same number of client threads as the number of files. It then assigns one file to each thread. Each thread is responsible for writing their files on the shared memory and waiting for the server response. Once they receive a compressed file as a response from the server, it is their responsibility to return this file to the master. The master saves the returned compressed output into files.

The server initially waits till anything is written in the shared memory. A client as soon as initialized, first checks if the Request Producer lock is available or not. The client waits until the lock is available. If it is available, then it acquires the lock and starts writing on the shared memory. It first writes the client id, then the size of the file and then the file contents in the shared memory. Then it releases the lock and waits till it receives a message from the server. The server uses a global variable current_response_thread to inform the client that the message is ready in the shared memory. The client spins on this variable and when it becomes equal to its client ID, the client acquires the Response Consumer lock and starts reading the compressed file and returns it to the application. It exits on completion.

Once clients start writing requests in the shared memory, the server can start processing these requests. It reads the client ID, file size and reads as many segments corresponding to the file size. Then it compresses the file content using the snappy tool. Once it is done, it writes client ID, compressed file size and compressed file contents in the shared memory and sets the current response thread variable to notify the clients about the processed file. A global parameter num_files is used to keep track of the number of processed files. Once all the files are processed the server exits.

Finally, the application uses all the returned output buffers to generate the compressed files.

2. Asynchronous Communication

The master application first creates the server. The number of threads is a configurable runtime parameter for asynchronous communication. The application creates an input buffer array, an output buffer array, an input size array, and an output size array. It also creates a pointer array which shall store the pointers of the responses produced by the server in the shared memory. These arrays are only accessible to clients and not to the server. The pointer array also has a lock associated with it. The application first reads all the files into the input buffer array. It saves a dummy data at the end and a dummy input size of -1 at the end to note the end of files to be read. Next, it creates a number of threads as provided by the user as runtime parameters. These client threads now run in parallel.

In the client thread, some memory is allocated to the response that may be received from the server in the future. The client first checks if there is any compressed file left globally to be returned to the application. If not, it exists. If still all files haven't returned, next it checks if there is any message from the server. If yes, it saves the message (a reference to the compressed file in the shared memory) in the global pointer array. Now it checks if the Request producer lock is available and whether there are more files to be read. If yes, it acquires the lock and starts writing to the shared memory. It writes the client id, file size and

the file contents in the shared memory. After the write is complete, it releases the lock. If the lock wasn't available in the first place, then it looks at whether there are any files in the pointer array to be processed and whether the pointer array lock is available or not. If yes, then it acquires the pointer array lock and starts reading from the pointer array. A variable processed_file_counter is used to mark the position to start reading the array. In this respect, the pointer array simply works as a queue with front and rear pointers. Once a pointer is read from the pointer array, the client accesses the address of the compressed file data in the shared memory, reads it and writes it to the output buffer array, thus returning the compressed file to the application. The whole processes run in a single while loop.

To summarize, the clients first check if they can read the input files. If yes, they will do it. If not, they will check if they should return any existing compressed file. This will be continued until all compressed files are written. There is no blocking of the clients and hence they are performing asynchronous communication.

3. Quality of Service (QoS)

Quality of Service is analogous to fair share scheduling. We have implemented it for synchronous communication. The master allocates approximately equal files to the clients. One after another, the clients get a chance to make requests by writing in the shared memory. For this purpose, we have used a global circular counter which decides whose turn is it to make a request. Since, synchronous communication is a blocking communication, the concept of quality of service can be introduced by ensuring that each client gets its own fair share of requests to be made for file compression. The global counter loops through all the clients in a circular fashion just like the game of passing the ball. So, initially all the clients make their request for compressing one file each and wait for the response. Then each client receives a response and make a fresh request again and start waiting for the response of this new request. In this way, we ensure that there is no bias in processing requests from a specific client and the requests are processed in a fair way.

B. Summary of Project Directory Structure:

a) snappy-c folder:

1. app.c : The master application: Creates shared memory, server, clients. Provides input files to clients and saves output buffer returned by clients as compressed files. It initializes the client and the server. Multiple clients are initialized using pthreads. When clients are finished, we terminate the corresponding threads. After all, this is done we terminate the server.

2. client.c: The client: It writes the input buffer into shared memory and obtains compressed output from shared memory and returns them to master. It has the following three functions:
    a. client_init()- It takes the input arguments and calls either synchronised compress(), asynchronised_compress() or qos_synchronised_compress() based on the parameters.

b. synchronised_compress(): It waits for request_producer_lock to open. Once the lock is free, it set the lock and writes on the shared memory. After writing on the shared memory, we update the request producer pointer and unlock the lock. After this, it waits for the response from the server. When response consumer lock is free, it will start reading the response(compressed file) from the shared memory and write it to the output buffer. This output buffer will be used in app.c to write down the compressed file and save it.

c. Following are the contents client writes on shared memory:
   i. "client": to denote client has written (6 characters)
   ii. client id: 4 characters are reserved for this. Say if client id is 15 we store it as 0015.
   iii. "memory": to denote that we will write memory size next. 6 characters
   iv. Memory size: We reserve 8 characters to store memory size.
   v. File content: Next, we write file content
   vi. Let's see an example: client id:15, memory size:3, file content: "abc"

c	**l**	**i**	**e**	**n**	**t**	0	0	1	5	**m**	**e**	**m**	**o**	**r**	**y**	0	0	0	0	0	0	0	3	**a**	**b**	**c**

d. Following are the contents client writes on shared memory:
   i. "server": to denote server has written (6 characters)
   ii. client id: 4 characters are reserved for this.
   iii. "memory": to denote that we will write memory size next. 6 characters
   iv. Memory size: We reserve 8 characters to store memory size.
   v. File content: Next, we read the file content
   vi. Let's see an example: client id:15, memory size:3, file content: "abc"

s	**e**	**r**	**v**	**e**	**r**	0	0	1	5	**m**	**e**	**m**	**o**	**r**	**y**	0	0	0	0	0	0	0	3	**a**	**b**	**c**

e. qos_synchronised_compress(): Each client has multiple files which we want to compress. We store the file_pending as the number of files left to be processed. The input buffer has multiple files which we will write on the shared memory. In QOS implementation, we give each client fair access to the server. We check if file_pending>0, we start writing one file on the shared memory. This is similar to the method described in the synchronised_compress(). We check the write lock if it's free, lock it, write on the shared memory. Decrement file_pending by one. Unlock the lock. Now it waits for the response from the server when the file gets compressed. On receiving the response from the server, we read from the shared memory and write the compressed file on output buffer.

f. asynchronised_compress(): In asynchronous communication, any client can process any file (input/ compressed) at any time based on whether it is free and whether the lock is available. In the while loop, the client first checks if there is any message received from the server. If yes, it waits till the global pointer lock is free and then it pushes the response into the global pointer array. Next, it checks if there is any input file to be requested to the server for compression and whether the request producer lock is available or not. If yes, then it writes the file to the shared memory. If the lock is not available or if the client has finished writing and has released the lock, it checks if there is any response pointer in the pointer array to be processed. The processed file counter is the measure of how many responses are pending to be processed relative to the newest response. So, if processed counter is -3, there are 3 more responses that need to be processed. It also checks if the pointer lock is available or not. If yes, it acquires it and reads the response pointer. Using the response pointer, it starts consuming the response from the shared memory and returns this compressed file back to the application. Thus we see that clients first check if any request is to be made. If yes, they make it. If not, they check if any compressed file is to be returned. If yes, they return it. No client waits for any response. Also the clients do not care if the file they are returning to the application was among the files they sent to the memory. So, it maximizes the throughput of the application. Everybody is working in harmony and there is no wastage of time or memory.

3. server.c: The server: It reads the input from shared memory and uses snappy tool to compress the files and saves them back to shared memory. First, we link the shared memory and set up the snappy tool environment.
    a. It will read from the shared memory, the file which has to be compressed. This is done request consumer pointer “req_cons”. We read the file content in same order as we described above: client, client_id, memory, input_memory_size, file contents.
    b. We feed the file content to snappy tool for compression.
    c. Now we have two cases: for sync and async:
        i. In sync mode, we save the the compressed output to the shared memory update the response producer pointer. Now, we will wait for response consumer lock to get free and get the current position of response consumer. Now, we write the output on the shared memory.
        ii. In async mode, we send a message to the client regarding the response producer pointer. The client will get the message and can read the compressed file.

4. include.h: This contains the global variables of the project. These are SHM memory key and size, all the locks, is_synchronized, number of files pending to be processed, pointers to the shared memory.

5. Input files: sample.txt, sample1.txt, sample2.txt, sample3.txt, sample4.txt

6. Corresponding compressed files

7. snappy.c: For performing file compression. Called by the server.

b) results folder:
i) input_output_files folder: Contains all input and output files
ii) output_log_files folder: Contains the logs of the following experiments:

1. Async mode:
    a. 2 clients with 3 files
    b. 2 clients with 4 files
    c. 2 clients with 5 files
    d. 3 clients with 4 files
    e. 3 clients with 5 files

2. Sync mode:
    a. 1 client with 1 file
    b. 2 clients with 2 files
    c. 3 clients with 3 files

3. Sync with QOS
    a. 2 clients 2 files
    b. 2 clients with 4 files
    c. 2 clients with 5 file
    d. 3 clients with 4 files
    e. 3 clients with 5 files
    f. 3 clients with 6 files
    g. 10 clients with 4 files

Implementation Issues

Our code works well for the sync and async mode along with the sync QOS mode. We could further implement the async in QOS mode. We have tested our implementation in many test cases. It works great for most of the cases but in very few cases we get a deadlock and segmentation fault. The segmentation faults are mainly due to memory leaks and can be debugged using gdb.

Conclusion

In this project, we implemented the sync and async mode along with the QOS. In sync mode, the client has to wait to get the service done by server. This may results that the server is sitting idle. In contrast, the async mode utilizes the resources very effectively. It starts processing a new file without much delay. Moreover, each client can start writing to the output buffer. In QOS mode, all the threads are working in a collaborative way. As in each thread get a fair amount of server access. In contrast, in sync mode, each client is working independently. They one by one send their file to server gets processed.