CS-6210-A -Advanced Operating Systems

Project 2

Nidish Nair | GT ID# 903055730 **Akshat Harit | GT ID# 903090915**

Design details

Synchronous (Blocking) Communication

API implemented - **call_service()**

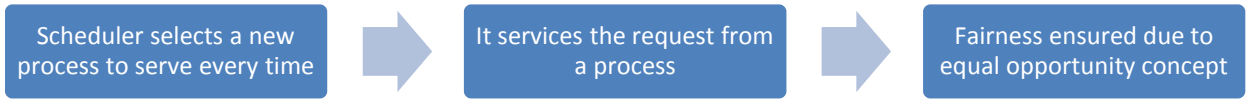
Sync calls are made by the client applications to the server via the library api which waits for the server to process it, and then informs the client to retrieve it from shared memory. Thus, requesting and receiving of the result happens within single library function call. The arguments and the result for the requested service are stored in/read from **shared memory**, so all the caller has to do is initialize the argument value and the result pointer. If the call is unsuccessful, then -1 is returned, else 1 is returned. The client application would use this returned info to make sure it gets what it needs.

Quality of Service Design

Our basic design idea behind ensuring fairness via QoS is this - to ensure that a single process does not get serviced continuously at the cost of other waiting processes. Also we realize that there is no definite way for the server to predict how many requests a process might make and for how long.

Therefore, we decided to implement a **Round Robin** scheduler in our server which will select one request to be executed per process. The ring-like queue from which these requests are picked up need not have any kind of ordering, because the scheduler that picks them up, does so in a fair manner. So, for example, if process A made 5 requests, B made 3 requests and C made 2 requests from the server, the order in which the scheduler would schedule the requests from our queue would be - **ABCABCABAA**.

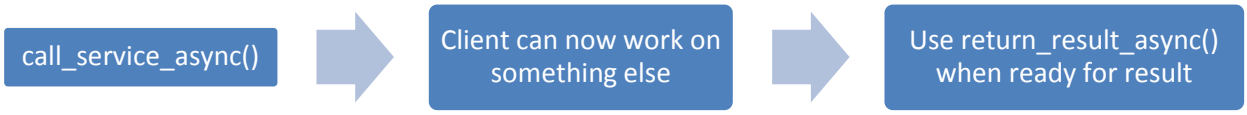
Please note that we have also enforced an **upper limit** on the number of client processes that can connect to the server. Once that limit has been reached, new clients will have to wait for an already connected process to disconnect from and free the server.



Asynchronous Communication

API implemented - **call_service_async(), return_result_async()**

In an async call, the library api just puts its data on the ring queue of the server (if there is a vacancy) and returns - it doesn't wait for the server to process its request. The client may then invoke another api call to retrieve the result when it's ready. If the server has finished processing, the result would be available for reading. Otherwise the process may continue waiting or do some other work. Similar to the sync request, a -1 is returned if the server is too busy to handle the request.



Sample Application

We designed a sample application which can make either synchronous or asynchronous calls. The service being requested from the server is the computation of the **nth Fibonacci number**. The user is prompted to select which kind of communication the client should use to communicate to the server - sync or async. Subsequently, the requests are fired off, results received, and the results are displayed.

To simulate multiple clients, our application forks off several client processes which in turn will make one or more requests to the server.

In a manner similar to the way machines communicate over a network, a **communication channel** is first initialized when each client attaches to the shared memory space created by the server. After that a client makes request(s) to the server randomly. When the client process is done with all its requests it will close its communication channel with the server, which means that it will **detach itself** from the shared memory space.

All of the above mentioned functionality is implemented by using the APIs that we designed as a part our library. Please refer the API section for more details about the APIs.

Server Application Design

The server after initializing the shared memory component enters into an infinite loop and constantly loops over a ring queue structure which will contain a client request to service. Based on the status flags that are set when the client enters its request, the server identifies the process request to fulfil. The statuses of every position in the ring queue are as follows –

- Status **0** – The position is free for receiving a new client request and data.

- Status **1** – The position has been filled with input data of some client, and is awaiting execution by the server.
- Status **2** – The position has been updated with the result of the request and is now ready to be read by the client

For fairness under the QoS mechanisms, each client registers its pid in a shared memory global array. The fairness mechanism is enforced by the server by ensuring that each process's request is scheduled and serviced one after the other, as explained above in the QoS design.
After the scheduling has been done, the server reads the argument and calls the requisite function. Currently the design is sufficiently portable that for functions returning integer with integer arguments, a trivial one line change would allow one to substitute any function instead of the given Fibonacci function.
After the function has finished processing, the status flag is set to '2' so that the client process knows that the request has been processed. The server then selects another process' request for scheduling from the ring.

Implementation Details

API Functions that are exposed through the library

initialize_main_comm_client()

main_comm* initialize_main_comm_client(pid_t)
This function first uses ftok to get the shared key. It attaches the calling process to the shared memory segment. It returns the pointer to the shared memory segment. It also registers the pid of the calling process in the shared pid array so the server can do round robin scheduling.

call_service_async()

handler call_service_async(main_comm * main_comm_ptr, int arg, pid_t pid)
This is the function for asynchronous call.

call_service()

int call_service(main_comm * main_comm_ptr, int arg, int * result, pid_t pid)
This function provides for synchronous call.

close_main_comm_client()

void close_main_comm_client(main_comm * main_comm_ptr, pid_t pid)
This function detaches the client process from the shared memory segment.

return_result_async()

int return_result_async(main_comm * main_comm_ptr, pid_t pid, int arg, int queue_pos)
This function is used to get the result after an asynchronous call is made.

Server side functions

fibonacci function – fib()

The server provides a function for nth Fibonacci number. To provide good time metrics, the function has been coded with recursion, without dynamic programming like constructs. This ensures that the function call will involve a non-trivial amount of processing

initialze_main_comm()

This function creates the shared memory segment and then attaches to it.

schedule()

This function picks the next request to service in the ring-queue structure.

sigproc()

As the server keeps running constantly a signal handler is written such that on receiving SIGINT(Ctrl+C) signal, it cleans up the memory before exiting.

Data structures in the program

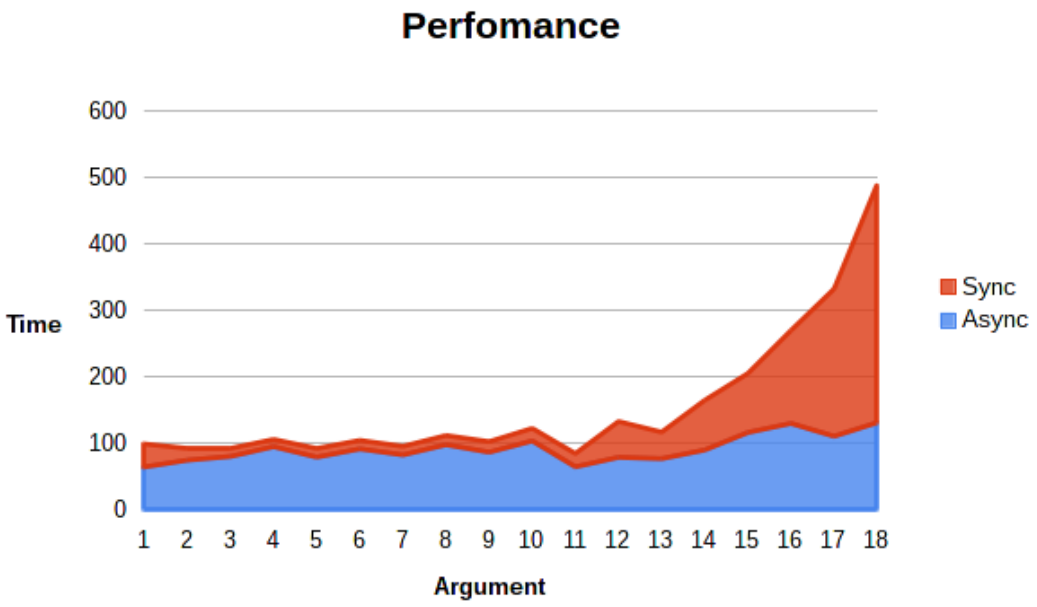
shm struct

This is a per call structures. It contains the status flag, argument for the call, pid of the calling process and a container for the result.

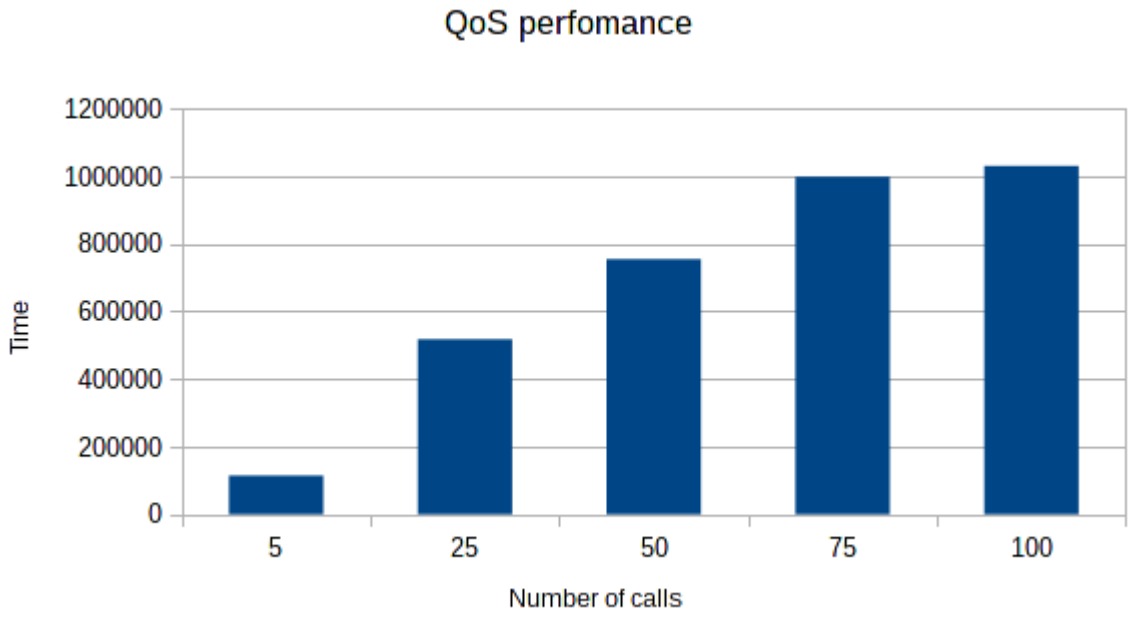
main_comm

It is the main ring structure that holds all the call for client requests. Also the server process parses this structure.

Performance Time Analysis



Here we see that for an asynchronous process, the average time for getting the Fibonacci data is almost constant. This is because the synchronous process has to wait for the server to finish processing, while the asynchronous process can do useful work in the meantime.



Here varying number requests were made by 5 processes at the same time. The order of request by process creation was 100, 75, 50, 25 and lastly 5. In spite of the fifth process being created relatively later, we see that the Round Robin scheduler has executed its requests and quality of service is maintained.

Final Notes and Observations

The granularity for QoS mechanism is at the request level. There is no concept of fairness based on the nature of the request. As such, for the pertinent case, a request for say 100th Fibonacci number is given the same service as the process requesting the 3rd Fibonacci number, even if the latter involves less processing time.

Currently functions that return integers and accept integer argument can be easily ported for the API. A more general approach would be to use void* pointers for arguments and function pointers that allow the server API to implement any function trivially.

If a client stops without cleaning up after itself, there is no mechanism to modify the ring structure so that the slot in the ring is available for some other requests. A time based mechanism could be used, but with it a detailed analysis needs to be done on how much time to wait.