

# Advanced Operating Systems Project 2

Aniket Sakhardande (903200874) and Krishna Kumar Ramachandran (903195735)

## Introduction

The purpose of this project is to develop a service to handle requests, which is analogous to the Dom0 domain in Xen. The implementation is a client-server model, whereby the server provides compression service to requests by applications through its API. This model is implemented as a Library service and can be accessed by any application.

The service is implemented in a way that it can handle multiple requests from a single client or from multiple clients. The service allows the choice between synchronous and asynchronous compression options. Message queues and locks are used to facilitate a multi-process environment.

## API

The API is basically a client library which has client.c and client.h files. The user gets the liberty to either call the Synchronous compression function or the Asynchronous compression functions that are defined in the client.c file. The user would run their programs by providing the names of the files (or even a single file) that are to be compressed as command line arguments. The API program will simply call the synchronous or asynchronous modes of compression depending on his needs.

In the **synchronous function call**, the user client blocks till the compressed files are ready to be retrieved. But, in the **asynchronous function call**, the user adds all his files to the queue and continues to work without waiting. He can retrieve the files at a later time by calling the compressed\_files function defined in the API. More details about the two files are provided below. Snappy is integrated to provide the required compression mechanism.

### Client.h file

This is a header file which is used to include all the header functions and the structures defined for use on the client side. It also contains function prototypes for the functions to be used for synchronous/asynchronous compression.

### **Client.c file**

This file contains the entire client side workflow. As defined in the client.h file, this file has three functions: one for synchronous compression (synch\_compress) and two for asynchronous compression (asynch\_compress function for sending the files for compression and compressed\_files function to retrieve the compressed files).

## **TinyFile Service Internals**

Our TinyFile service is a shared memory based file compression service that provides compression in response to client file requests. The clients submit their requests to the TinyFile service (called server.c in our program) over a request message queue which is set up by the service at startup. There is another acknowledgement queue for each of the files requested by the client, so that the clients are notified about the compressed files. All sorts of signalling are sent and received via the acknowledgement queues.

There is a shared memory segment allocated by the server which has the shared memory size, maximum input file size and also the mutex related parameters (pthread mutex library). This is to be read by each of the clients to check if their file request can be processed. The shared memory is updated once when the client requests a file for compression, and once when the client retrieves the compressed file. This shared variable is protected by a mutex to avoid race conditions during shared memory updates.

The files are passed from the clients to the service through shared memory. The shared memory id is a unique file id generated from the file name and path by using the ftok() function. The acknowledgement queue id's specific to files are also generated using the same file name but different project id argument of the ftok() function. This information is exchanged via the request message queue. More details of the server.c file is shown below.

### **Server.c file**

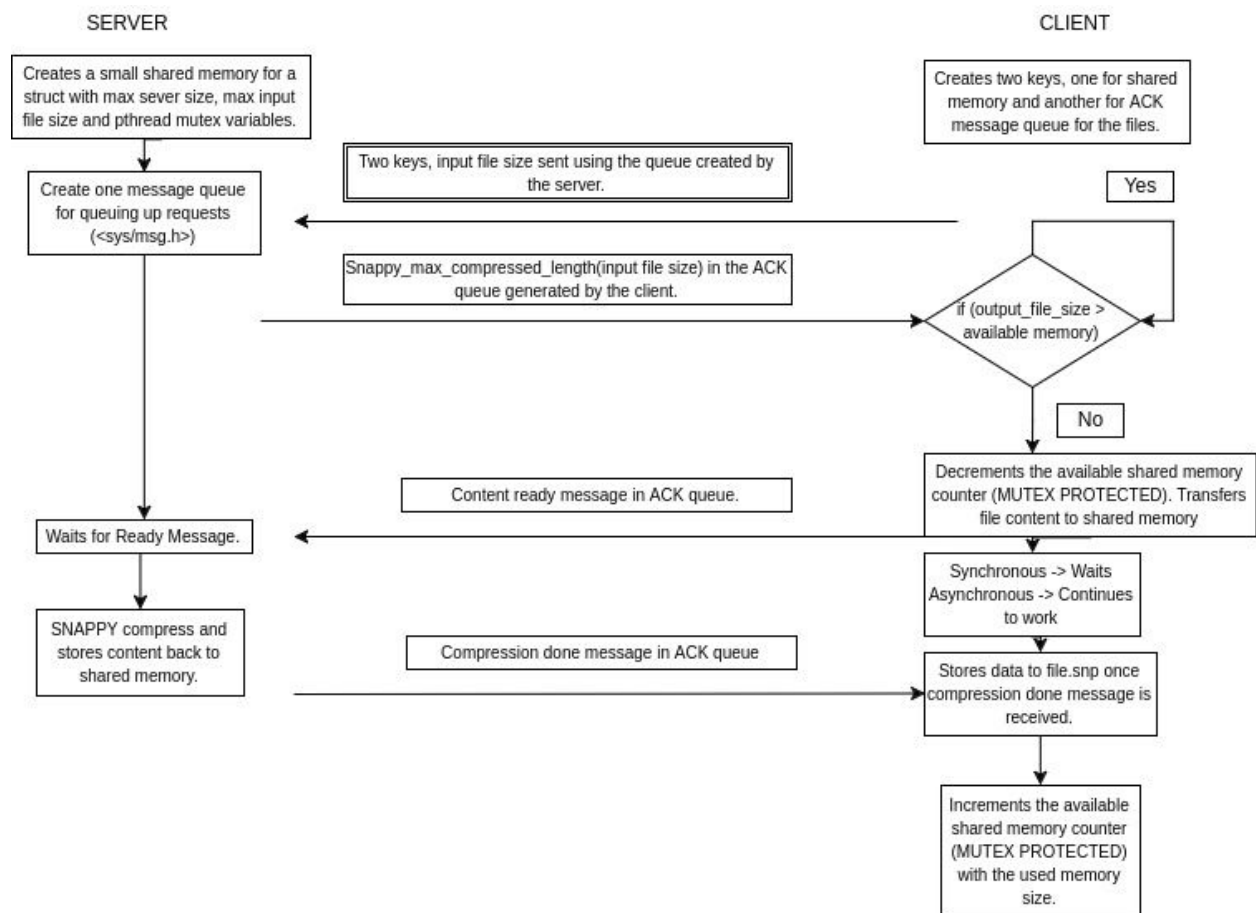
This file contains the workflow for the server. It includes all the headers and structures defined in client.h and the server functionality. The server acts as a daemon process (implemented in a while loop) which keeps running, and responds only when the client

has made compression requests. More information about the work flow is given later in the report.

## Quality of Service requirements:

Each of the clients were assigned a unique id. A parameter called `client_id` was created in the `message_buffer` structure. Unique\_id to a client was generated using `ftok()` of the first file. `Client_id` variable was incremented in the server everytime a client sends a request for the first time and was copied to the message buffer structure. An array with two pointers to track the active clients is maintained at the server side. Then the server serves the requests in round robin fashion from head to tail pointers. However, we faced several implementation issues in this approach and eventually ran out of time, hence this part of the code was not implemented

## Work-Flow chart



## Work-Flow

1. Server.c is run in the server side. It takes two arguments. One is the total number of segments and the other is the maximum memory allocated for each segment.
2. Server program creates a shared memory to share the server size, maximum input file size that can be given and mutex related parameters. These parameters belong to a structure.
3. The server program also creates a message queue in which the client side requests lined up.
4. Client on receiving the parameters sent, checks if the file size is smaller than the maximum input file size, if not, it breaks out with an error message.
5. Then the client creates the two keys as mentioned earlier in the client.c file explanation.
6. Using the message queue created by the server, it sends the two keys, one for the shared memory and another for the acknowledgement queue. The client also sends the input file size in this queue. These parameters are basically stored in a data structure which is sent.
7. Server on receiving these details, finds the maximum compressed file size for the given input file size and sends this detail using the acknowledgement queue.
8. Client on receiving the maximum output file size, checks if the server has enough memory to process each of its files. If the server doesn't have enough memory, it waits till memory gets available. This memory availability check is done using pthread mutex locks, to make sure there are no race conditions.
9. Once the memory is available, it creates a shared memory using the key it generated earlier, the size of which is equal to the maximum output file size sent by the server. This optimization is a key process because we are not utilizing a complete memory segment for all files.
10. Once the shared memory is created, the client copies all the file contents to the memory and sends signal to the server.
11. The server on receiving the signal compresses the contents of the shared memory and stores it in a temporary location. Later, the compressed file contents are stored in the same shared memory that the client created. The allocated temporary memory is then freed. Then it notifies the client that the compressed file is ready.
12. In **synchronous function**, the client waits at this point till it gets notification from the server that the file is ready. In **Asynchronous functioning**, it continues with other operations and comes to this point at a later stage to retrieve the files.
13. The client access the shared memory and uses fputc function to copy the contents to a new file whose naming convention is "filename.snp". We are using

.snp so that the compressed file outputs can be checked using the snappy uncompress to make sure we get the original file.

## Test Cases

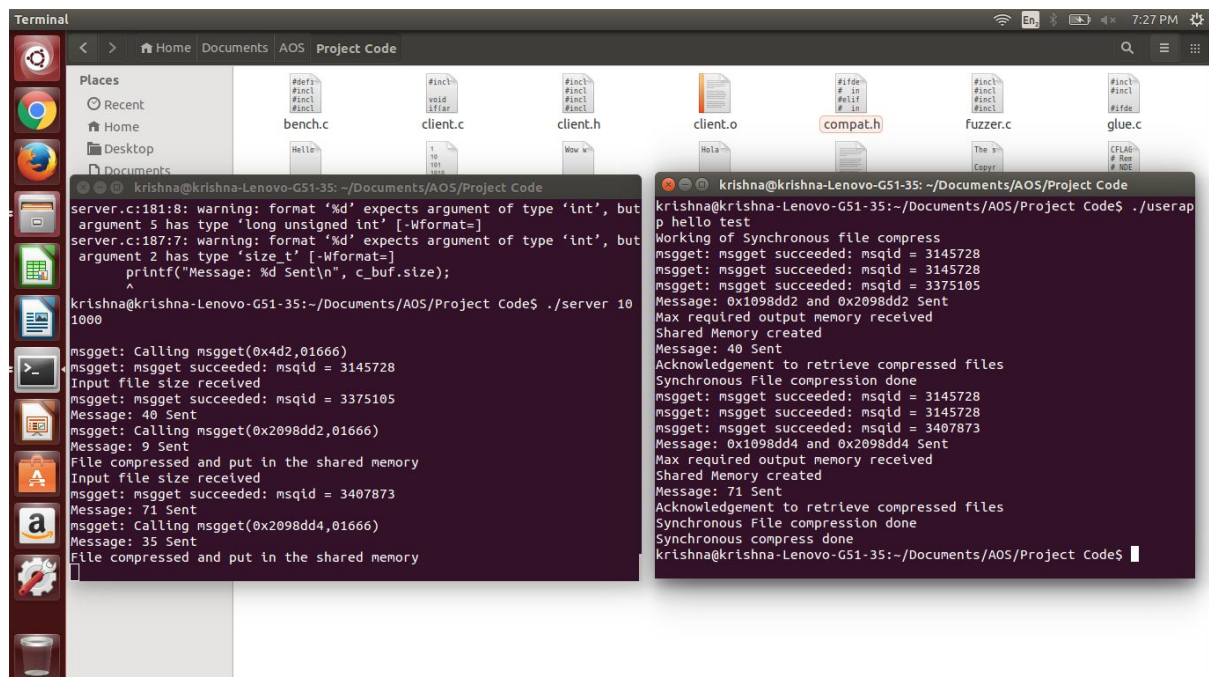
The system was tested for the following test cases:

1. Single file for both synchronous and asynchronous modes of compression
2. Multiple files using synchronous and asynchronous modes of compression
3. Single/Multiple files for multiple clients using synchronous and asynchronous modes of operation
4. The input file size exceeds the size of the maximum shared memory (Throws out an error message and terminates execution)
5. Running application without providing command line arguments (Throws appropriate error messages)

To run the application, follow the steps mentioned in **Readme.txt**

## Results

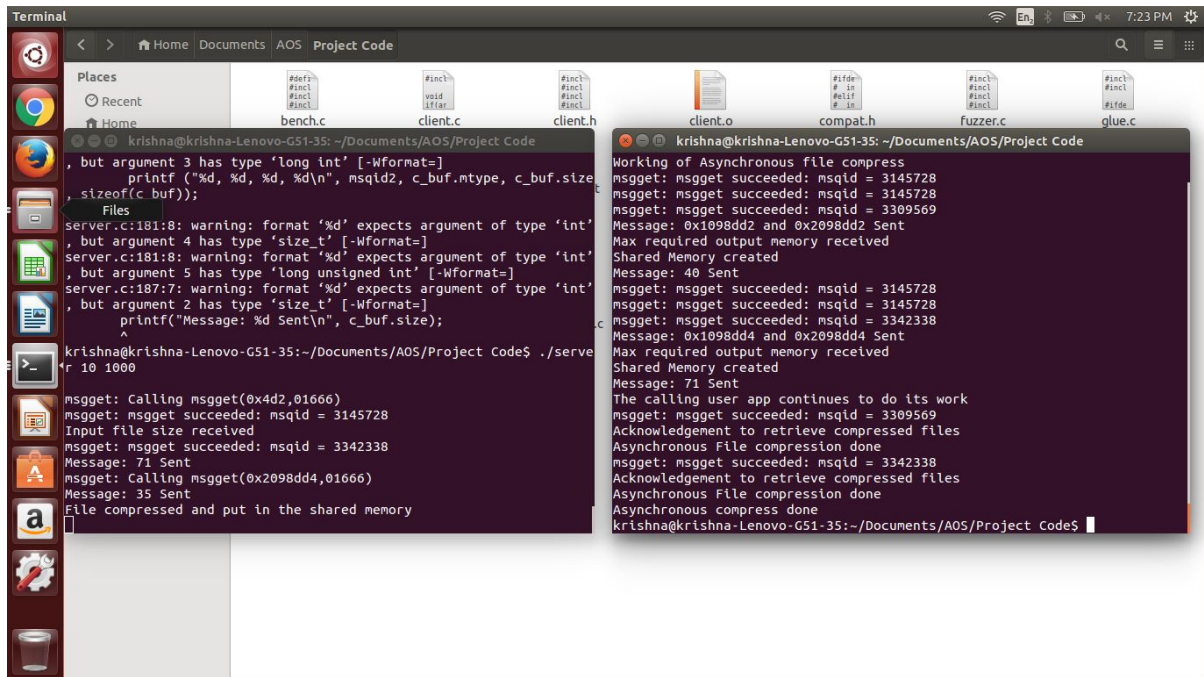
1. Synchronous Compression



```
krishna@krishna-Lenovo-G51-35: ~/Documents/AOS/Project Code
server.c:181:8: warning: format '%d' expects argument of type 'int', but
argument 5 has type 'long unsigned int' [-Wformat=]
server.c:187:7: warning: format '%d' expects argument of type 'int', but
argument 2 has type 'size_t' [-Wformat=]
printf("Message: %d Sent\n", c_buf.size);
^
krishna@krishna-Lenovo-G51-35:~/Documents/AOS/Project Code$ ./server 10
1000
msgget: Calling msgget(0x4d2,01666)
msgget: msgget succeeded: msqid = 3145728
Input file size received
msgget: msgget succeeded: msqid = 3375105
Message: 40 Sent
msgget: Calling msgget(0x2098dd2,01666)
Message: 9 Sent
File compressed and put in the shared memory
Input file size received
msgget: msgget succeeded: msqid = 3407873
Message: 71 Sent
msgget: Calling msgget(0x2098dd4,01666)
Message: 35 Sent
File compressed and put in the shared memory

krishna@krishna-Lenovo-G51-35:~/Documents/AOS/Project Code$ ./userap
p hello test
Working of Synchronous file compress
msgget: msgget succeeded: msqid = 3145728
msgget: msgget succeeded: msqid = 3145728
msgget: msgget succeeded: msqid = 3375105
Message: 0xi098dd2 and 0x2098dd2 Sent
Max required output memory received
Shared Memory created
Message: 40 Sent
Acknowledgement to retrieve compressed files
Synchronous File compression done
msgget: msgget succeeded: msqid = 3145728
msgget: msgget succeeded: msqid = 3145728
msgget: msgget succeeded: msqid = 3407873
Message: 0xi098dd4 and 0x2098dd4 Sent
Max required output memory received
Shared Memory created
Message: 71 Sent
Acknowledgement to retrieve compressed files
Synchronous File compression done
Synchronous compress done
krishna@krishna-Lenovo-G51-35:~/Documents/AOS/Project Code$
```

## 2. Asynchronous Compression



```
krishna@krishna-Lenovo-G51-35: ~/Documents/AOS/Project Code
server.c:181:8: warning: format '%d' expects argument of type 'int'
, but argument 3 has type 'long int' [-Wformat=]
printf ("%d, %d, %d, %d\n", msqid2, c_buf.mtype, c_buf.size
, sizeof(c_buf));
server.c:181:8: warning: format '%d' expects argument of type 'int'
, but argument 4 has type 'size_t' [-Wformat=]
server.c:181:8: warning: format '%d' expects argument of type 'int'
, but argument 5 has type 'long unsigned int' [-Wformat=]
server.c:187:7: warning: format '%d' expects argument of type 'int'
, but argument 2 has type 'size_t' [-Wformat=]
printf("Message: %d Sent\n", c_buf.size);
krishna@krishna-Lenovo-G51-35:~/Documents/AOS/Project Code$ ./serve
r 10 1000
msgget: Calling msgget(0x4d2,01666)
msgget: msgget succeeded: msqid = 3145728
Input File size received
msgget: msgget succeeded: msqid = 3342338
Message: 71 Sent
msgget: Calling msgget(0x2098dd4,01666)
Message: 35 Sent
File compressed and put in the shared memory

Working of Asynchronous file compress
msgget: msgget succeeded: msqid = 3145728
msgget: msgget succeeded: msqid = 3145728
msgget: msgget succeeded: msqid = 3309569
Message: 0x1098dd2 and 0x2098dd2 Sent
Max required output memory received
Shared Memory created
Message: 40 Sent
msgget: msgget succeeded: msqid = 3145728
msgget: msgget succeeded: msqid = 3145728
msgget: msgget succeeded: msqid = 3342338
Message: 0x1098dd4 and 0x2098dd4 Sent
Max required output memory received
Shared Memory created
Message: 71 Sent
The calling user app continues to do its work
msgget: msgget succeeded: msqid = 3309569
Acknowledgement to retrieve compressed files
Asynchronous File compression done
msgget: msgget succeeded: msqid = 3342338
Acknowledgement to retrieve compressed files
Asynchronous File compression done
Asynchronous compress done
krishna@krishna-Lenovo-G51-35:~/Documents/AOS/Project Code$
```

## Limitations

1. The shared memory and the request message queue created by the server is not freed as the server is terminated through control+c. Hence it is advisable to clear the shared memory and message queue through ipcrm command before program execution
2. In case clients terminate through errors, corresponding message queues should be freed through the msgctl function. However, there were several anomalies reported in this
3. The client should provide file names at the very start of the application run. This is more of an implementation choice than a limitation.
4. In very rare scenarios, the compression with the same files as input produces an shmget failure error. This was found to be due to the fact that the allocated shared memory was not detached during previous runs.
5. In the advos server, there were several shmget and msgrcv errors due to undestroyed message queues and shared memories.
6. The server has to be run before the client. This is because server acts as a daemon process who continuously serves client requests.

## **Conclusion**

The program thus provides an application interface that can be used efficiently by multiple users or application. The inherent client-server semantics ensures efficient operation and performance of the service