CS 6210: Advanced Operating Systems Mar 16, 2023

Project 2 Report

Instructor: Ada Gavrilovska *Aditya Vikram*

1 Abstract

This report provides an overview of the design and implementation of the TinyFile service and library, which parallelizes file compression while following the Dom0 paradigm in Xen. The experimental results and the affects of various design choices are also discussed in the report.

2 Introduction

The goal of the project is to create a TinyFile service which acts like Domain0 in Xen ([3]). It has access to the resource required by other domains, which it grants via shared memory and IPC calls. Here the resource is a file compressor, which is integrated inside the TinyFile service. Client apps want access to the file compressor, but they can only do so via IPC with the TinyFile service. Clients are only allowed to use limited shared memory and message queues to interact with the service. We also provide a library called libTinyFile, which exposes the required functionality for interacting with TinyFile service and using its compression algorithm.

There are two common standards for IPC in Unix-like systems: POSIX and System-V; the former is built on top of the older System-V interfaces and implementations. However the availability and the extent of their implementations vary across Unix-like OSes. Since I was developing on OS X which doesn't have part of the POSIX semaphore and shared memory implementations, I chose to go with the System V model.

Sequence diagram in Figure 1 is presented for a rough idea of the flow of logic. Let's go through the design of each process and its modules briefly.

3 Design

3.a Service

The TinyFile service is built as a standalone executable run from the command line. We provide the number of shared memory (SHM) segments that it can allocate and the size of each SHM segment as command line args to the service. When the service is started, a `TFClientRegistry` object is created. This object is responsible for most IPC functionalities of the service and lives through the lifetime of the service. The initialization of `TFClientRegistry` object creates a System-V message queue with the service's directory as the `ftok()` key. It also allocates `n_sms` number of System-V SHM segments each of size `sms_size`. Along with this, it also initializes three System-V semaphore sets for SHM synchronization between the service and all its clients. This will be discussed later in detail. After this initialization, two things happen:

- A listener thread is started, which waits for messages from clients on the service's queue using `msgrcv()`. This thread is listening to the registration (and deregistration) message from new (existing) clients on the queue. A registration message contains the pid of the client, which we use as a index for the service's data management. The message also contains a private System-V message queue-id, to which the service sends a response upon successful (or failed) registration. This response message contains more metadata for the SHM communication to be done, and the message queue communication stops here (until when the client wants to deregister itself). The metadata for SHM communication includes the SHM segment ids, sizes, count and the semaphore ids.
- `n_sms` number of SMS observer threads are started, where each of these threads is responsible for managing its corresponding SHM segment. Managing here means waiting for a client to write to that

SHM segment, then reading the request data, processing it accordingly and writing back to the segment for the client to read. The synchronization for this is discussed later in section SHM Synchronization. Here we have a design choice to pick the number of such SHM observer threads. While we don't want to spawn arbitrarily large number of such threads, the problem statement here aims for very small values of `n_sms` (1, 3 and 5). Furthermore, in the implementation, `n_sms` is a `uint8_t` type, so it can't be more than 256 anyways. With modern operating systems, hundreds of threads are also manageable in the worst case, but the design choice to have `n_sms` number of such threads is mainly based on the small range of `n_sms` values (≤ 10). If we were solving for more number of SHM segments, a more valid approach would be to use timed observers, where we have a lesser number of fixed threads but each thread manages multiple SHM segments, rotating periodically in a round-robin manner. This implementation would be more involved with synchronization, so I went with the simpler approach for our small use-case.

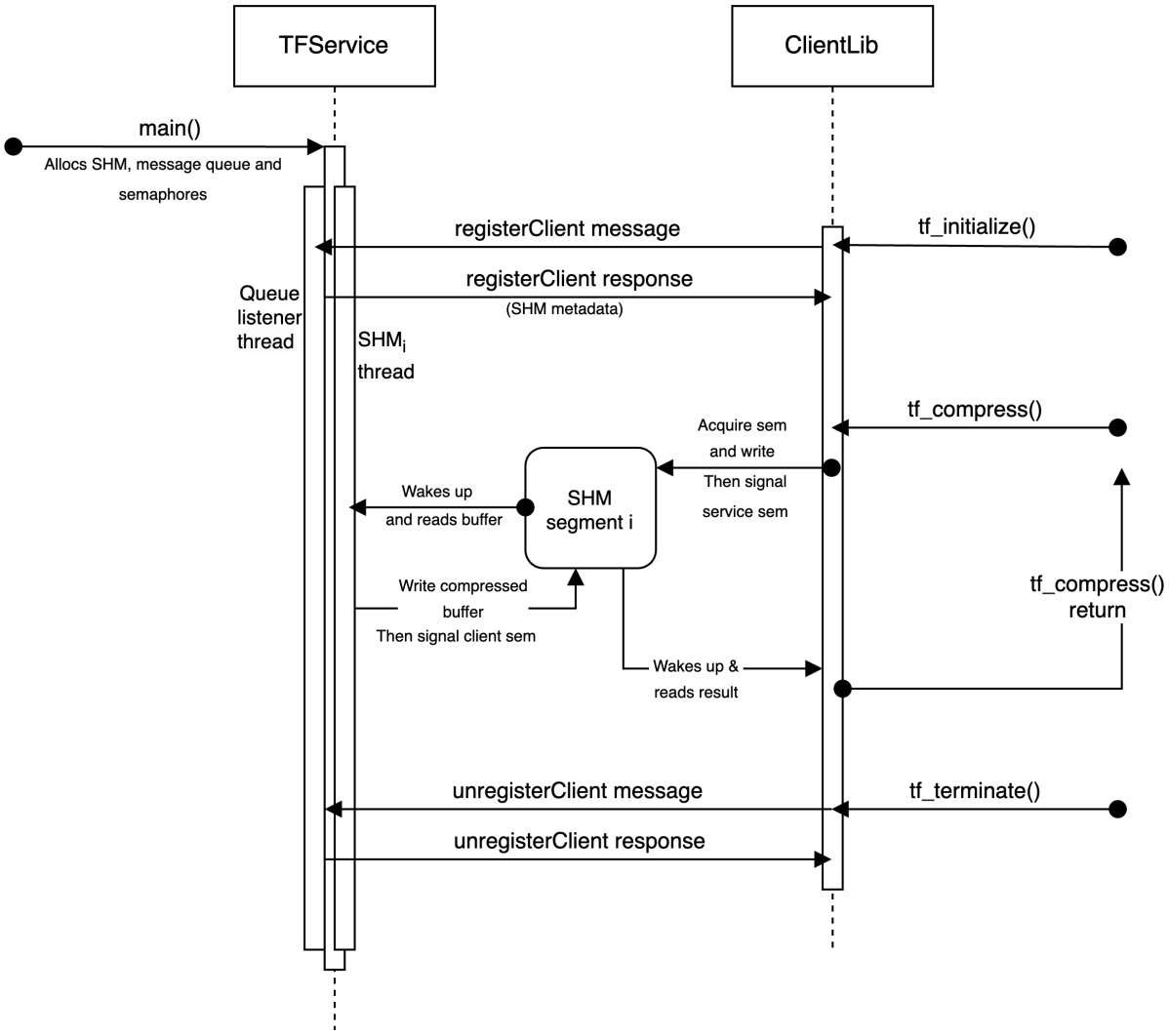

Figure 1: A sequence diagram for TinyFile components

3.b Client library

The client library for TinyFile implements the required functionality to initiate communication with the service, and provide APIs for the client apps to compress data via SHM segments. Following are the API

prototypes exposed from the public header `TinyFile.h` of the library:

- `tf_initialize()`: This is the entrypoint for the library where we perform all the setup. A `TFService` object is created which lives through the client app's lifetime, although it barely uses any memory after `tf_terminate()` is called. On initialization, the `TFService` object knows how to establish communication with the service via service's message-queue. So it creates a private message queue (for service's response) and sends a registration message with the pid and private queue's id to the service's queue. The service does its thing and responds with the SHM communication metadata that is used to initialize `TFClientSHMManager` object, that does the SHM heavy-lifting for the client library. `TFClientSHMManager` class contains the logic for attaching the service's SHM segments in client's address space and reading/writing to the SHM as and when required.

- `tf_terminate()`: when the client app is done with the compression work, it can terminate the connection with the service using this function. It sends a deregistration message on the service's message queue and finishes all the queued compression work on the client side. The service can clear the client from its data structures when it gets the deregistration message.

- `vector<char> tf_compress(vector<char> input)`: this is the main compress function where the client app passes a buffer of characters (read from some file possibly) to compress. The library splits it into chunks that fit inside the given SHM segments and then parallelizes the compression. Since there are at most `n_sms` segments, having more than those many threads in parallel would be a waste since we can only read/write one data block from a given segment at a time. So again, following similar logic from the last section, I chose to divide the work into `n_sms` threads (at most, if the file size is at least `n_sms` $\times$ (`sms_size`-δ), δ being the metadata size for each segment). Basically we spawn `n_sms` threads with the first `n_sms` blocks of data to compress, where each thread acquires the semaphores (discussed in section SHM Synchronization) required for that SHM segment, writes the input data and waits for the service to respond with the compressed buffer. Then we wait for these `n_sms` threads to join before proceeding with the next batch of blocks, until we have the compressed result for each buffer. Finally, we serialize the result back to a buffer of chars to return to the client.

- `tf_compress_async(vector<char> input, callback)`: this is the async version of the above API, where the client passes a callback to be invoked with the result when the compression is complete. We assume this is lower priority than the sync calls, so these requests can be enqueued onto a thread pool. I have adapted a minimal `TFThreadPool` class from my previous work experience, which allows to queue tasks on a given number of underlying threads, and provides functionality to wait for all tasks in the queue to complete, or to preemptively stop the queue, where more tasks won't be picked by the threads. Since my assumption is that the async requests are lower priority, I have initialized the `TFThreadPool` object with just 1 underlying thread, but this is configurable by changing the constants in the code, if we know the hardware capabilities and know the client use-cases better. Having just 1 thread to process async requests allows the other threads to have more CPU time, so sync requests would complete faster than otherwise, with the block parallelization in the previous API. In the async task, we are still internally calling `tf_compress()` with the data and invoking the client callback with the return value. But because these are on a single thread queue, it might take longer for the callback to be called, compared to a sync call.

- `tf_wait_all_async_requests(input)` This is an additional API for the client app to wait on all its async requests, in case the app has to use those to perform some other actions or if it has to save and quit. This is a common synchronization primitive akin to a barrier, used in asynchronous settings. It relies on the async tasks' `TFThreadPool` object's functionality, we just wait on all the tasks in progress to be finished.

3.c SHM Synchronization

Our design doesn't allocate specific SHM segments to specific clients. This seemed like an unreliable design to me, because we don't know the load characteristics of our system, i.e. we don't know how many clients are connected at what time and when they might want to compress data. So the better design per me is to

allocate all the SHM segments on the service side, and give all clients the SHM ids for all these segments, so that every client can utilize every SHM segment in the best case.

Now this leads us to synchronization and security problems because clients can possibly read or overwrite each other's data. System-V semaphores come to the rescue here, we can use them to ensure valid and fast communication just with SHM (bypassing message queues which are slightly slower because of the implicit synchronization). For each SHM segment, we define three semaphores:

- `clientContentionSem`: this is used to decide which client "owns" that SHM currently, and prevents clients from reading or overwriting each other's data
- `shmServiceSem`: semaphore that the service thread managing this SHM segment waits on, and the client signals
- `shmClientSem`: semaphore that the client waits on, when it has to read the response, signalled by the service.

Because System-V makes semaphore available in sets, we only have to initialize and maintain three semaphore set id's on service and client side. Each of these sets has `n_sms` number of semaphores internally, one per SHM segment, but the API makes it easier to manage and operate on the semaphore sets directly. Next we briefly describe the flow of one block from Client → Service → Client:

- A client thread C1 managing SHM segment i wants to compress a block of data. It decreases `clientContentionSem` for segment i. If it is able to decrement the semaphore, then this thread gets exclusive access to the segment from clients' side. Now C1 decrements `shmClientSem` for segment i, which would succeed in the normal case because the previous client would've reset it after usage.
- C1 writes its data to the SHM and signals `shmServiceSem` for the segment i, so that the service thread S1 (which manages the SHM from service's side) can wake up. C1 also decrements `shmClientSem` again, which starts a wait for the service's response.
- S1 wakes up, reads the data from SHM segment, compresses it using snappy-c and writes back the response back into the same SHM segment. Then it signals `shmClientSem` for segment i and completes a loop of S1, so that the next loop will again decrement `shmServiceSem` and wait for a new client thread to write.
- C1 wakes up when S1 signals `shmClientSem` and reads the response from the SHM segment into result buffer somewhere. Then increments `shmServiceSem` one more time so that the next client thread can acquire it. Finally, it increments `clientContentionSem` for segment i, which possibly wakes up the next client thread waiting for SHM segment i.

4 Implementation

I've implemented the TinyFile modules discussed above in C++, where STL containers and algorithms helped a lot in reducing boilerplate code. System V APIs are used for all IPC, shared memory allocation and synchronization among the service and all clients. For concurrency within the service and client (via lib) processes, `std::thread`s from C++ STL are used, along with `std::mutex` for locks and `std::condition_variable`s for synchronization in `TFThreadPool`. For the shared memory, I used the first two bytes (`uint16_t`) to store the buffer size and the remaining space for the buffer. `uint16_t` seemed like a logical choice because our range of `sms_size` is up to 8192 for the prescribed experiments, and `uint16_t` can hold up to 65536, without losing out more memory per the vital SHM space.

Snappy-c [4] has been integrated in the TinyFile service for its compression algorithm. Because snappy was developed with a performance goal in mind, it doesn't achieve the same levels of compression as standard libraries like zlib. In fact, the max possible size of the compressed buffer might be greater than the original by a few bytes: `max_compressed_buff_len` $= 32 + \frac{7}{6}$ `input_buff_len`. I've kept this in mind while passing buffers in shared memory, so that the result buffer doesn't overflow. While this maintains correctness, it loses out on some shared memory space because we care for the worst case. As a result, a slight performance hit is expected because we likely have to compress more number of blocks over SHM.

`std::chrono` library has been used to measure Client-side Service Time (CST) in microseconds. A couple of Stack overflow posts helped in some aspects of the `TFThreadPool` implementation ([1]) and a utility vector-join function ([2]) to reduce code.

4.a Client App

The client app called `TFClient` is also written in C++. It includes the public header `TinyFile.h` of client library and statically links its source code. The library either takes a file path (`--file`) as argument or a text file (`--files`) with list of file paths, as prescribed in the project instruction. The file buffer is read and depending on the `--state` arg, we either call the sync API or the aysnc API (where the client app executes a while loop to simulate some random long task after calling the async API). The resulting compressed buffer is saved in an output directory that can also be configured using command line args (more details in the README). Finally, there is also a `--no_service` argument to ignore the service altogether and directly use snappy-c in the client app to compress, using the same `sms_size` blocks. This is only meant for verifying the correctness of the TinyFile service and library up to some extent.

Next we discuss the results and observations.

5 Results

The results are based on my M2 MacBook Air with an arm64 processor, 8GB RAM. I had to update some System-V constants to get the required values of `n_sms` and `sms_size` to work due to some MacOS restrictions. Following are the ranges of SHM sizes and counts for which I tested both async and sync TinyFile APIs:

- `n_sms`: 1, 3, 5
- `sms_size`: 128, 256, 512, 1024, 2048, 4096, 8192

Due to the snappy-c corner case described in previous section, 32 bytes was infeasible and I chose to skip the 64 bytes case as the available buffer space was very small after accounting for the snappy overheads.

Right off the bat, we can see in the left plot in 2 that large files have longer CST times. This in fact holds true for all scenarios throughout the experiments. It is expected because we are limited by the size of SHMs and each file has to be split into blocks, so larger the file, more the number of compression and SHM operations required. It can also be seen in Figure 2 that as the size of SHM segments increase, the difference between CST for the varying file sizes drops. This is because with larger SHM sizes, the number of SHM read, writes and synchronizations drops, so that isn't the bottleneck for CST anymore.

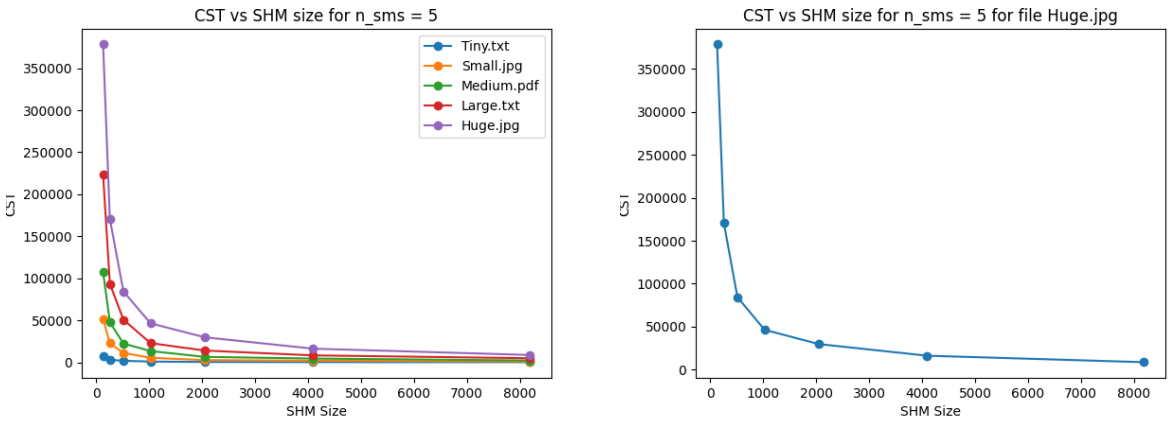

Figure 2: Variation of CST (μs) with SHM size for (a)left: varying file types and sizes; (b)a fixed big file

If we focus on one file Huge.jpg, we can clearly see the exponential effect of SHM size on the CST. CST rapidly drops off from around 380ms to 75ms as SHM size grows from 128 up to 512 bytes, but after 1024 bytes, the gain in performance becomes linear and eventually stagnates. So it would be a good choice to either go with 4096 or 8192 bytes per SHM in this client model.

The performance gains from adding more SHM segments is not that rapid though. As seen in 3, for SHM size of 1024 bytes, the drop in CST is significant from 1 segment to 3 segments, but reduces from 3 to 5 segments. This is likely caused by our design where both service and client processes (at least while the API is called) have one thread per SHM segment. With increasing number of threads, we have the overhead of creating and joining threads as well as context switches, which would reduce the gain we get from parallelizing over more SHM segments. This is the classical design choice in concurrency where we can use experiments like this to find the sweet spot with good performance while keeping the system overheads in control. I'd argue that 5 segments is still a good choice here because we get around 20% drop in CST with two more threads, which should be manageable in modern hardware and OSes.

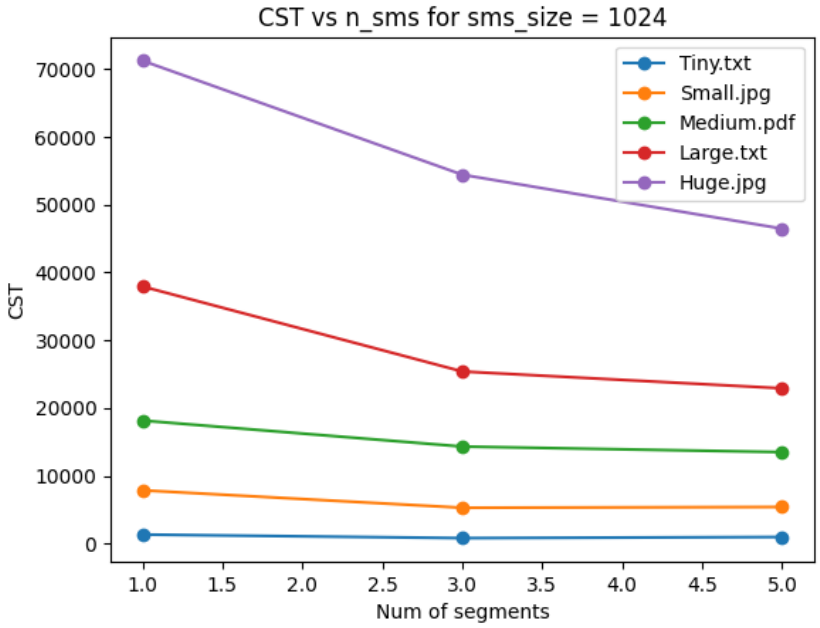

Figure 3: Variation of CST (μs) with number of SHM segments at fixed SHM size

Lastly, we evaluate sync vs async APIs. The difference is negligible in my experiments between the two APIs for varying sizes and types of files. The CSTs are very close and sometimes the async calls complete faster than some sync calls, depending on the overall system load. 4 shows two such examples with varying number of SHM segments, but the trend is same across SHM sizes and counts. On average, async calls take slightly longer, mostly because of the overhead in queuing to the thread pool and executing the task. I believe that if an experiment was done with thousands of API calls in one run of the client app, we would be able to see the difference in CSTs, because then all the async calls would get queued to a single thread. Unfortunately under the time constraints, I could not perform this experiment with a larger number of API calls.

For multiple file compressions in one run using `--files` argument, there is nothing notable to discuss from the experiments. The results scale as expected with the sync API calls. As discussed above, with a large number of files in one run, the async API call results would've been interesting to see. Some other notable but repetitive (or not very interesting) plots from the experiments are available at Drive link.

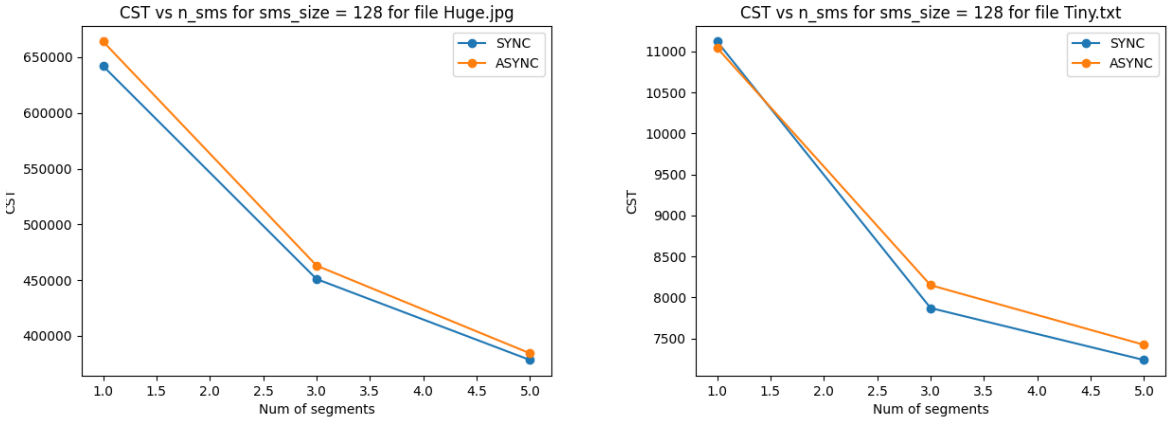

Figure 4: CST (μs) for sync vs async APIs for two different file sizes

6 Conclusion

The report summarizes the design of a Domain0 like service that uses shared-memory and synchronization semaphores as to create a highly performant communication mechanism between the clients and the service. We can expose resources to the clients in an efficient manner while having control over the maximum overall resource usage by the system, like Xen. The affects of different system parameters were also discussed and various design choices were reasoned for.

There is a bit of experimentation gap as discussed towards the end. Besides that, one implementation gap is the I assume that service is always running in the background when the client starts. I had planned to implement a notification to the client if the service quits midway so that client can gracefully handle that. That would be good exercise for the future as well, with many design choices to make.

References

[1] Efficiently waiting for all tasks in a threadpool to finish. https://stackoverflow.com/a/23899225.

[2] Merge vector of vectors into a single vector. https://stackoverflow.com/a/35291919.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, oct 2003.

[4] Andi Kleen. Snappy-c. https://github.com/andikleen/snappy-c.