Systems Programming

Remote File Memory Mapping

Brandon Birmingham & Andrew Carl Sammut

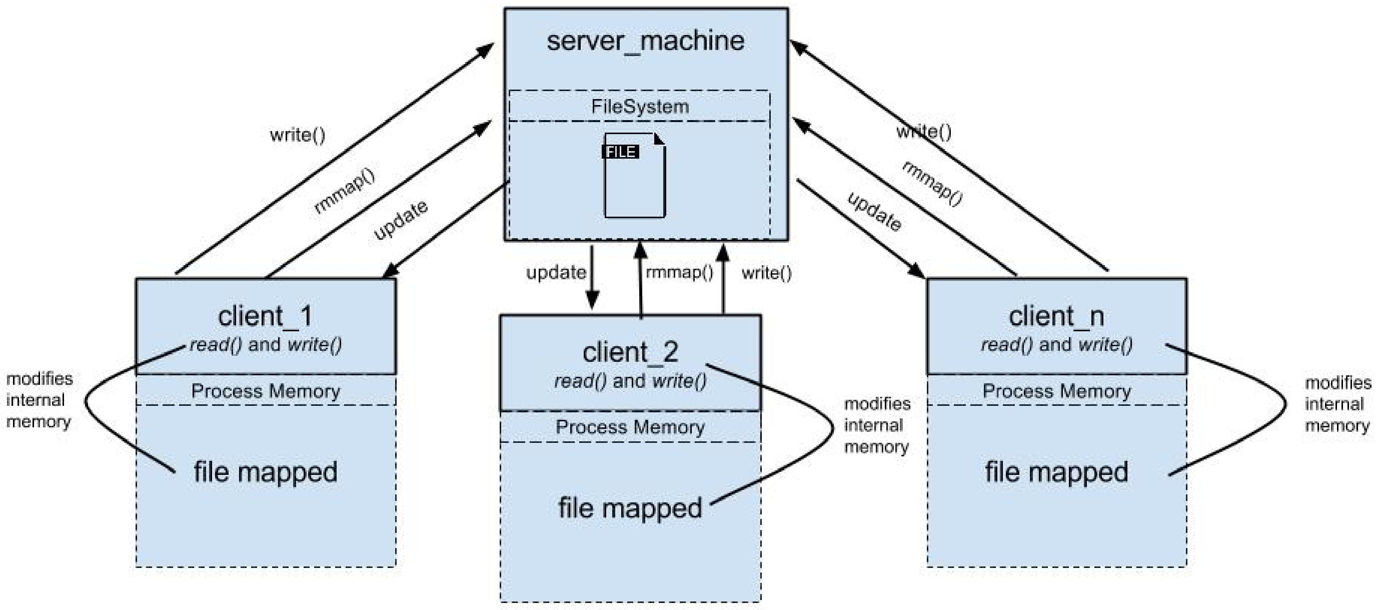
University of Malta
B.Sc. (Hons) Computing Science

Table of Contents

1. Introduction

Throughout this assignment a concurrent server-client system was implemented. Such concurrent server handles multiple clients that are all the time making requests to the server, in order to mmap the file or else writing to it. Due to mmaping the file, each client has the ability to access the file, from its own process, and each time making modifications (or reading) on such memory, which are finally propagated to the server. Different mechanisms were utilised to handle multiple clients, conflicts and communication over the network, which will be described in more detail throughout this report.

2. System Design

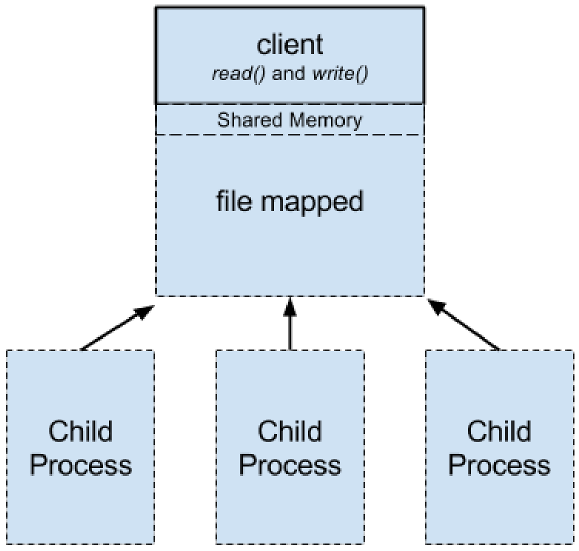


The diagram above, shows the high-level system design concept used to implement this particular system. As shown in the diagram above, the concurrent server implemented can handle multiple clients, by the protocol as shown above. Each client has its own process memory, in which all the contents of the remote file is to be mapped into. From the above pictorial representation it can be clearly seen that the client process will perform all the operations internally on its allocated memory address space. All the modifications performed by the client side, will be propagated back to the remote machine. This system was implemented by a library written in C, which effectively its main objective is to perform the mentioned functionality. The library is based upon the following functions:

1. **rmmap(fileloc_t location, off_t offset)*

This particular function is the basis of this implementation. It's main objective is to efficiently map a remote file located on a particular machine on the calling process. As it is clearly shown from the above prototype, the function locates the specified file via the *location* argument which it contains the remote machine's IP address, port number and the actual file path of the file that needs to be mapped. Once a connection is established between the client and the server machines the file contents can be transmitted via the TCP/IP connection. In order to handle this transmission between over the interconnected machines the socket interprocess communication was used.

As soon as the server receives the request from a particular machine to map a particular file, the server starts to send the file contents in chunks of 1024 bytes. This was necessary in order to handle large file transmission. Once the server starts sending the contents of the remote file, the client is designed to allocate the necessary memory on its memory address space.



In order to maximise the efficiency and to make the system more robust for handling multiple processes from a particular machine the concept of the shared memory Interprocess Communication was used. This means that, when multiple processes from a particular client machine needs to take control of a particular file that resides on the server a shared memory segment is initialised on the respective client memory after all the allocation of the remote file is performed on the client side. This means that all the process that will be running on the same machine will have access to the same chunk of memory and thus all the modifications will be visible from all the processes. Having said that, all the modifications performed by all the processes will be propagated to the server from a single shared memory. This was particularly achieved by simply sending the respective key associated with the remote file to the client process in order to obtain a shared memory segment depending on the remote file that was requested to be mapped. The idea of allocating a shared memory between processes running on the same machine can be described by the following code snippet:

```
//Creating/Obtaining the shared segment and set read/write permissions
if((sh_mem_id = shmget(key, strlen(mapped_file), IPC_CREAT | 0666)) < 0)
        perror("shared memory segment was not allocated");

//Attachment of segment with the malloc'd data space of the received file
if((shm = shmat(sh_mem_id, NULL, 0)) != (void *)-1)
        strcpy(shm, mapped_file);
```

2. *int rmunmap(void *addr);*

Once remote file is successfully mapped on the process's address space, the respective process can efficiently deallocates the mapped region. Moreover, this function handles the deallocation of the shared memory that can be attached to multiple processes running on the client machine each of which are mapping the same remote file from a specific server.

3. *ssize_t mread(void *addr, off_t offset, void *buff, size_t count);*

The client process has also the ability to read contents from its mapped region and allocates the read data into a specified buffer. The main objective of this function is to directly read contents from the allocated memory rather than reading multiple times directly from the remote machine.

4. *ssize_t mwrite(void *addr, off_t offset, void *buff, size_t count);*

This particular function provides a mechanism by which the process directly writes to the allocated mapped region of the remote file. When the process needs to modify the mapped contents a particular buffer is passed to this function in order to be written directly on the process address space exactly from the specified offset. Once the memory is updated by the respective process, the modifications are propagated to the remote file by writing to the the opened connection between the client and server. In order to successfully map all the contents to the remote machine particular flags were utilised in order to handle files that are greater the the default file's size segment (i.e 1024 bytes). In order for the server to be able to handle the synchronisation of large file, the following type of flag was used: *U$<cycles>$<file_contents>*. This means that, once the server matches the *U* flag the server will be prepared to update the remote file contents by the *file_contents* buffer stream. Moreover the *cycles* flag will instruct the server to waits for that particular number of buffers that will make the entire file contents. This is because the server handles 1024 bytes at a time. In order to effectively write to the contents received by the server, file locking mechanism was used in order to eliminate race conditions between multiple process trying to write at the same file location. Once a write is performed on the the remote file an update mechanism is performed in order to update all the clients that are will be mapping the same remote file. This is clearly explained in the following sub section of the documentation.

3. Update Mechanisim

For updating, an special mechanism for identifying individual clients (machines) was needed. Firstly, the unique socket descriptors were considered. However, since different clients use a different environment, then the same socket descriptor numbers were being used. This is not a problem for the system, as due to running concurrently, it will only connect to the client being handled by that specific child process.

Although child processes, share their parent's contents, different child processes do not propagate their changes between each other. Therefore, a shared memory containing an **array of process identifiers** was used to distinguish between child processes. The memory shared array declared is shown below :

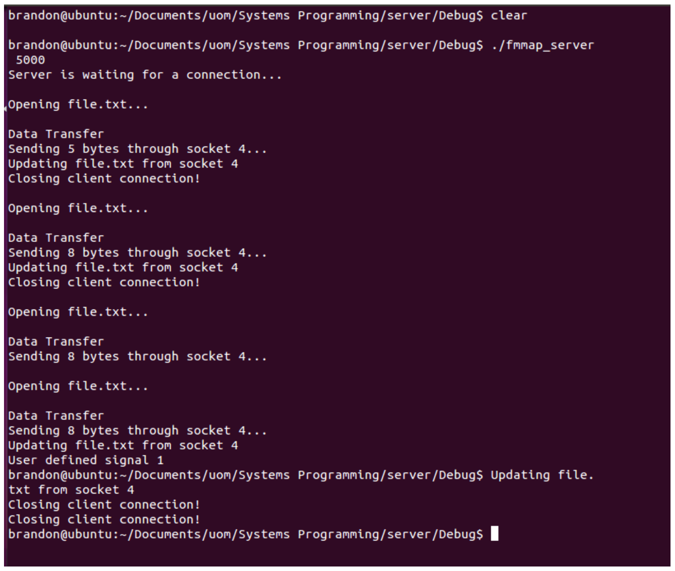
```
char *pids_mem = mmap(NULL, MAX_CLIENTS * sizeof(char), PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
```

As soon as the connection between the client-server is created, the pid of the child process handling the connection is stored in such array. Each time, a client updates the file on the server, a signal to each of the process identifiers found in the memory shared array, is sent by using the system call : kill(pid, SIGUSR1). A signal handler was implemented, in order to receive the signal “SIGUSR1”, at each child process. As soon as it receives the signal, a special encoding sequence “$$!*!$$” is sent to the client, by using the write() system call and the socket descriptor found in the child process.

At the client side, the client receives the encoding sequence inside one of its buffers using the read() system call. The buffer is checked for such sequence before every read(), write() and mmap() system calls in order to always to have the mapped file synchronized with the actual file before performing operations on it. As soon as the sequence is captured, the original mapped file of the client, is remapped in order to obtain the updated version of the file. After obtaining the updated file, the client can restart to perform its normal operations.

4. Test cases

Numerous tests were performed on our system, to test that the system is functioning correctly. Using two different PCs, on one PC a server and a client (in debug mode) were run and on the other another client (in debug mode) was run. Both clients were run in debug mode, in order to simulate that the locking and updating mechanism were working as they should.



```
brandon@ubuntu:~/Documents/uom/Systems Programming/server/Debug$ clear

brandon@ubuntu:~/Documents/uom/Systems Programming/server/Debug$ ./fmmap_server
 5000
Server is waiting for a connection...

Opening file.txt...

Data Transfer
Sending 5 bytes through socket 4...
Updating file.txt from socket 4
Closing client connection!

Opening file.txt...

Data Transfer
Sending 8 bytes through socket 4...
Updating file.txt from socket 4
Closing client connection!

Opening file.txt...

Data Transfer
Sending 8 bytes through socket 4...

Opening file.txt...

Data Transfer
Sending 8 bytes through socket 4...
Updating file.txt from socket 4
User defined signal 1
brandon@ubuntu:~/Documents/uom/Systems Programming/server/Debug$ Updating file.
txt from socket 4
Closing client connection!
Closing client connection!
brandon@ubuntu:~/Documents/uom/Systems Programming/server/Debug$ 
```

5. System Defects

Our system currently, is only able to handle changes on a single remote file found on the server. However, changes in multiple files can be handled by using a table where the process id of each child process (handling a client) is stored, along with the file the client is mapping to. A dynamic linked list can be implemented as a data structure of the table. If client A mapping file.txt, updates the file, all other clients which are mapping the same file (found from the table) should also be updated, by using the same mechanism we implemented.

Also, **very** large files are not handled correctly due to lack of optimisations.

Another known defect occurs in the case when two clients write to the file at “exactly” the same time. In this case, the updated version on the remote file depends on the update of the last process. The reason behind this is that first client that should write takes the lock and updates the file. Then, the other client will not be able to receive the update of the file, and therefore updates the old version of the file.

6. Conclusion

From the above description it can be seen that the main functionality and the principal objective of the remote file mapping system was achieved with minimal defects and constraints. Obviously, the implemented system is not fully perfect but further improvements can be made in order to increase its efficiency and robustness.

NB: In order for the system to work the file should be placed relative to the server directory.