Your submission archive (only .zip or .tar.gz archives accepted) must include:

1. A **makefile** with the following rules callable from the project directory:
   (a) `make text-server` to build **text-server**
   (b) `make text-client` to build **text-client**
   (c) `make clean` to delete any executable or intermediary build files

   There is a 25% penalty for any deviation from this format. Without this makefile or something very similar, your project will receive a 0.
2. Correct C\C++ source code for an application implementing a server with behavior described below for **text-server**. The file(s) can be named anything you choose, but must be correctly broken into header(.h) and source(.cc) files.
3. Correct C\C++ source code for an application implementing a client with behavior described below for **text-client**. The file(s) can be named anything you choose, but must be correctly broken into header(.h) and source(.cc) files.
4. A README.md file describing your project. It should list and describe:
   (a) The included files,
   (b) Their relationships (header/source/etc), and
   (c) The classes/functionality each file group provides.

Note that all files above must be your original files. Submissions will be checked for similarity with all other submissions from both sections.

Overview

This project explores usage of IPC and file I/O in the form of memory-mapped files. Your task is to create a client/server pair to process text files by loading the text file into file-backed shared memory, allowing a second program to parse that file changing every character from lower case to upper case.

Your task is to create a client which uses the POSIX IPC method you choose to send a command-line provided file name to the server. The server will load the contents of that file into shared memory. The client will use threads to search the shared memory for lower-case characters and replace them with upper-case ones.

Server

The server is started without argument, i.e.,
`./text-server`
If the server does not execute as indicated, the project will receive **0 points**.

The server is used to accept the file path from the `text-client`, open the file, and the file into the shared memory.

You may hard-code the name of the shared memory provided by the client.

The server must perform as follows AND YOU MUST DOCUMENT IN YOUR CODE WHERE EACH STEP TAKES PLACE:

1. At start, writes

   `"SERVER STARTED"`

   to its STDOUT (use `std::cout` and `std::endl` from `iostream` if using C++).

   - You may hard-code names necessary for IPC and synchronization.

2. Upon receiving a file name and path (relative to the project directory) from the client (using the IPC method of your choice),

   (a) It writes

   `"CLIENT REQUEST RECEIVED"`

   to its STDLOG (use `std::clog` and `std::endl` from `iostream` if using C++; STDERR if C).

   (b) Using the path,

   i. Writes `"\tOPENING: "` followed by the provided path name to the terminal's STDLOG, e.g.

   `"\tOPENING: dat/dante.txt"`

   ii. Opens the file to receive a file descriptor
   - If open fails, indicate to the client using the IPC method of your choosing

   iii. Uses the file descriptor to map the file to shared memory
   - After mapping it writes

     `"\tFILE MAPPED TO SHARED MEMORY"`

     to its STDLOG and
   - Closes the file using file_descriptor, writing

     `"\tFILE CLOSED"`

     to the server's STDLOG.

3. Resumes waiting for future client contact

   - The server need not exit

Cont.

Client

The client must start with one command-line argument—./text-client, followed by:

1. The name and path of the text file (relative to the root of the project directory) which should be modified
2. e.g.,
   `./text-client dat/dante.txt`

If the client does not execute as described above, your project will receive **0 points**.

The client is used to pass the file name and path to the `text-server`, search the lines of text transferred via file-backed shared memory, and replace any lower-case character with an upper-case one.

You may hard-code names necessary for IPC and synchronization.

The client must behave as follows AND YOU MUST DOCUMENT IN YOUR CODE WHERE EACH STEP TAKES PLACE:

1. Using your choice of IPC passes filename and path to server
2. Creates four threads, each of which;
   - Process $\frac{1}{4}$ of the characters of the file-backed shared memory, replacing any lower-case characters with upper-case ones.
3. If the server was unable to open the file, writes

   ```
   INVALID FILE
   ```

   to its STDERR and exits returning 1
4. Terminates by returning 0 to indicate a nominative exit status

Cont.

Notes

Client

- Consider the barrier problem for signalling the server from the client. A named POSIX semaphore allows processes access to multi-process synchronization.
  - You might hard-code the named semaphore into your client or have your server pass its name with IPC.
  - Your client likely will not create or destroy named semaphores. It will connect to semaphores created by your server.

Server

Your server should create and destroy any named semaphores used to synchronize the server and client.

- Given that your server will be killed with `CTRL` + t, you will only be able to destroy a named semaphore by setting up a signal handler using `void (*signal(int sig, void (*func)(int))(int)` to destroy the semaphore.
- Your second option is to simply attempt to destroy any semaphore before you create it.
  - Note that your expectation is an error `EINVAL (sem is not a valid semaphore)` if the semaphore does not currently exist. THAT IS A GOOD THING, in this case. Continue past this error and initialize the semaphore with the named used to destroy it.
- The start process for your server will like proceed as follows:
  1. Create any necessary semaphores—one of which should be the barrier
  2. Do the following "forever":
     (a) Lock the barrier semaphore
     (b) Attempt to acquire the locked barrier semaphore—effectively blocking until the client unlocks it
     (c) Use shared memory or any other IPC to get file name/path from client
     (d) Transfer file via file-backed shared memory structure

References

- I have included a file providing an example of file-backed memory mapping. It is not all you require, but it does provide much.
- It can be built with

```
gcc file_backed_mmap.c -lrt -o file-backed-mmap
```

- It can be invoked by

```
./file-backed-mmap dat/test.txt whatever
```

  - Ensure you view the file before and after invoking proc
  - Where do the odd characters come from at end of file?

Cont.

Grading

Grading is based on the performance of both the client and server. Without both working to some extent you will receive 0 POINTS. There are no points for "coding effort" when code does not compile and run.

The portions of the project are weighted as follows:

1. **makefile**: 10%
2. **text-server**: 40%
3. **text-client**: 40%
4. **README.md**: 10%

Test Files

You are provided three test files of varying length to test your code; check the dat directory in the provided directory:

- Anna Karenina. Leo Tolstoy, 1870. `dat/anna_karenina.txt`, 1.9MB.
- Dante's Inferno. Dante Alighieri, 1472. `dat/dante.txt`, 218KB
- Lorem Ipsum text. Generated 2022. `dat/lorem_ipsum.txt`, 578B.

The End.