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 the IPC in the form of shared memory. Your task is to create a client/server pair to process text files by loading the text file into shared memory, allowing a second program to parse that file for lines of text containing a matching string.

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 a command-line provided search string. The client will print each line found containing the substring to the standard output stream.

Several of you complained that the second project was too easy, especially with the provided code. If you would like to challenge yourself with this project, make the following changes:

- Use a structure in the server which allows it to signal the client as soon as it begins loading lines into memory (rather than after it finishes). A circular queue would be an ideal structure. The goal is to allow the server to write lines to the queue's write while the threads read from the queue's read pointer until one pointer catches the other.
- In the client, begin writing lines to the standard output stream as soon as the first one is found by a thread. This may be modeled as a producer/consumer where the search threads are producers and the STDOUT writer is the consumer. The producers will likely be much faster than the consumer, so use multiple slots each representing a line of text containing the search string (a circular queue of strings from the producer/consumer lecture would work). Because there is only one STDOUT, multiple consumers will not provide a speedup.

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 and search string information from the `text-client`, open the file, and write the lines of the file to 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++).
2. Upon receiving a file name and path (relative to the project directory) from the client,
   (a) It writes
   "CLIENT REQUEST RECEIVED"
   to its STDLOG (use `std::clog` and `std::endl` from `iostream` if using C++).
3. Opens the shared memory.
   - After it writes
     `"\tMEMORY OPEN"`
     to its STDLOG
4. Using the path,
   - Writes `"\tOPENING: "` followed by the provided path name to the terminal's STDLOG, e.g.
     `"\tOPENING: dat/dante.txt"`
   - Opens and reads the file
   - Writes the contents of the file to the shared memory opened above
   - Closes the file and writes
     `"\tFILE CLOSED"`
     to the server's STDLOG.
     – If open fails, indicate to the client using the IPC method of your choosing
   - Closes the shared memory and writes
     `"\tMEMORY CLOSED"`
     to the server's STDLOG stream.
5. The server need not exit

Cont.

Client

The client should start with two command-line arguments—./text-client, followed by:

1. The name and path of the text file, relative to the root of the project directory which should be searched and
2. The search string for which the file will be parsed,
3. e.g.,
   `./text-client dat/dante.txt "the journey"`

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 shared memory, and count the number of lines of text found, while printing each line and its count to its standard output stream.

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

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

1. Creates a shared memory location and initializes with any necessary structures. I recommend using one or more character arrays of fixed size as a structure to hold the lines of text. Use the shared memory as a means of transmission, not storage.
2. Sends the text file name and path to the server. I recommend using the shared memory you created.
3. Copies the contents of the file, via shared memory, to local storage. I recommend writing into an `std::vector<std::string>` using `std::string(const char *)` and `std::vector::push_back`
4. Creates four threads, each of which;
   - Process $\frac{1}{4}$ of the lines of the shared memory to
   - Find any lines containing the search string
5. The client uses the results of the threads' search to write all lines of text found to its STDOUT as follows:
   - The client must begin each line found with a count starting from 1 and must use a tab character (\t) after the count, e.g.,

     ```
     1 \t Midway upon the journey of our life.
     2 \t Maketh the journey upon which I go.
     3 \t From her thou'lt know the journey of thy life."
     ```

     Note that the \t should be an actual tab character.
   - If the server was unable to open the file, writes
     `INVALID FILE`
     to its STDERR (use std::cerr and std::endl from iostream if using C++).
6. Destroys the shared memory location.
7. Terminates by returning 0 to indicate a nominative exit status

Cont.

Notes

Client

- Your client should create and destroy your shared memory. See `shm_open` and `shm_unlink` below.
  - If your client does not destroy the shared memory, subsequent runs will fail and you will lose a substantial amount of points.
- 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) Connect to the shared memory initialized by the client
     (d) Use shared memory or any other IPC to get file name/path from client
     (e) Transfer file via shared memory structure

Shared memory structure

You may store anything in shared memory. For example you might store a structure as follows:

`Shared Memory Struct`[1]

- `line` : `const char[line length]`
- `line length` : `const static size_t`
- `path` : `const char[path length]`
- `path length` : `const static size_t`

Assuming the client places the file path in the correct member before raising the barrier blocking the server, the server may simply read the file path from the shared memory.

[1] Notice this can be an actual `struct`, no need for privacy if you are managing synchronization externally.

Cont.

References

- Linux Programmer's Manual: shared memory
- shm_open() for creating shared memory
- Explanation of POSIX shared memory, SoftPrayog
- Linux Programmer's Manual: POSIX semaphores
- Examples of creating and using POSIX semaphores
- POSIX Threads Programming: cmu.edu, llnl.gov

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.