Your submission archive (only .zip accepted) must include:

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

   There is a 30% 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 **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 **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, aside from those found in the class repository, 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 files containing large numbers of equations by loading the text file into shared memory, allowing a second program to parse that data and sum all lines from the file.

You must 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 perform the arithmetic equations, line-by-line, then sum the results. The client will print each subtotal and final total to the command line.

Server

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

The server is used to accept the file name and path from the `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 STEPS 2–4 TAKE 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 from the client,
   (a) It writes
   "CLIENT REQUEST RECEIVED"
   to its STDOUT (use `std::cout` and `std::endl` from `iostream` if using C++).
3. Opens the shared memory.
   - After it writes
     `"\tMEMORY OPEN"`
     to its STDOUT
4. Using the path,
   - Writes `"\tOPENING: "` followed by the provided path name to the terminal's STDOUT, e.g.
     `"\tOPENING: dat/equations_1.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 STDOUT.
     – 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 the command-line arguments—./client, followed by:

1. The name and path of the text file, along with
2. The number of lines in the file, e.g.,

```
./client dat/equations_691.txt 100
```

If the client does not execute as described above, the client and behavior portions will receive **0 points**.

The client is used to pass the path, filename, and line count to the `server`, visit each line of text the server stores in the shared memory, calculate the sum of each line, and accumulate (add) all sums.

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 and writes:
   "`SHARED MEMORY ALLOCATED: 12345 BYTES`"
   I recommend using one or more character arrays of fixed size as a structure to hold the lines of text.
2. Sends the text file name and path to the server. You may use the shared memory you created.
3. Creates four threads, each of which;
   - Process $\frac{1}{4}$ of the lines of the shared memory,
   - Converts each line into an equation, calculates and sums the results, and
   - Provides its final total to the controlling process.

   After creating the threads, writes
   "`THREADS CREATED`"
   to STDOUT (use std::cout and std::endl from iostream if using C++).
   You must use POSIX threads (pthreads). Anyone found to be using the C++ thread library will be reported as violating Academic Integrity.
4. The client uses the results of the threads to note the number of lines processed by each thread, the total calculated by each thread, and finally the sum.
   - The client must report as follows

     ```
     THREAD 0:  250 LINES, -13426.6
     THREAD 1:  250 LINES, 2738
     THREAD 2:  250 LINES, 8993.93
     THREAD 3:  250 LINES, -459.8
     SUM: -2154.47
     ```

     To its STDOUT.
   - 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++).
5. Destroys the shared memory location.
6. 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.
- To coordinate files larger than you can accommodate in your initial allocation, you might look at the communication strategy used by the producer/consumer model.

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]

- `buffers[4][kBuffLen] : char[][]`
- `lengths[4] : std::size_t[]`

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

Cont.

Consider the size of kBuffLen carefully. You know that the minimum size of allocation is some multiple of a page (4KB). You should select a value large enough to handle the common case, but not so large as to cause runtime issues. You are provided an example of the largest file you should be able to handle (`dat/equations_50000.txt`).

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 for correctness. There are no points for "coding effort" when code does not compile and run.

The portions of the project are weighted as follows:

1. **documentation**: 10%
   - Zero Lint errors: 5%
   - Correct README.md: 5%
2. **makefile**: 10%
   - All or nothing.
   - Without correct makefile, only documentation can be earned.
3. **server**: 25%
   - Source code builds executable.
   - Executable runs without error (note it does not have to actually process anything, just execute).
4. **client**: 25%
   - Source code builds executable.
   - Executable runs without error (note it does not have to actually process anything, just execute).
5. **correctness**: 30%
   - Program performs correctly with 1000 line file: 10%
   - Program performs correctly with 50000 line file: 10%
   - Program performs correctly with a [299-799] line file: 10%
   - These will be new files used for grading, not the same ones given for testing.
   - If program fails due to previous run (incorrect shared memory or semaphore management), then no points are awarded.

Cont.

Test Files

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

- `equations_1000.txt`
- `equations_50000.txt`
- `equations_691.txt`

The End.