ELEC-C7310 – Learning Diary 2

Data Transfer

Juho Pellinen

The final result of the project was quite successful, the program works most of the time. However, there are some issues which I would've fixed/added if I had more time:

- The client (reader program) reads only 65536 characters from the file
    - Maximum throughput is ~64 kB
    - Crashes the client if the file is larger than 65536 characters
- Unable to write log files unless the log file has been created (I had same problem with the program one)
- FIFO-transmission (both sending and receiving) had to be done with while-loop (even though there is just one transmission per client)
- There were odd issues which had to be solved using quite inefficient methods (for example copying string into another using while-loop and index)
- FIFOs used for transferring the data (named after client's pid) are not removed (even though they are unlinked and removing the metadata FIFO works)
- Sometimes another transfer gets the same PID transferred than another transfer
- More logging with the writer
- A single client could be utilized for sending multiple files at once

I started the project implementing the client and its functions (reading the file into a string and printing it to the user). Because I had created similar functions in the previous program, this was not that hard for me. However, there were some issues with the memory and reading the file, so I had to put manually the end-of-the-string character ('\0'). One thing that could be improved with client is that it could transfer multiple files once with a single instance. This however may cause problems with files (if all the transfers have the same pid), but I haven't investigated this further.

I wanted to set the client to send the data through a FIFO which has been named the client's PID. I think implementing this connection was the hardest part of this program. For many days I tried to figure out a method that I could transfer the pid and the data in the same FIFO. After asking help for this problem, I realized that I should've thought "outside-the-box" (by creating a separate FIFO) instead of thinking that my own thought (one FIFO for each data transfer including the pid) would've been good. I managed to solve this problem with the help of course staff (thank you once again!). I decided that the best option would be using the client's pid as the name of the FIFO used for transferring data. For transferring the pid from client to writer I decided to create a separate FIFO (metadataFIFO, located in /tmp/metadata by default). The FIFO implementation was not hard, especially with the help of good demonstration codes given in this course!

The implementation of writer was more interesting and complicated thing to do. One client instance was enough for one file, but a writer has to perform with multiple transmissions incoming all the time. That's why threading the writer process was important. On the start the writer starts listening to the metadataFIFO for incoming metadata transmissions. After that a new thread will be initialized with the received pid information from the client. The new thread listens the FIFO named after the pid (located in /tmp/<pid>). Threading was a bit hard, but I think that I understood threading better with C than I did with Python.

After receiving the data from the pid FIFO, the thread creates a custom structure defined to containing the pid and the data. This was done because threading a function requires only a single void parameter. That's why I couldn't create separate fields for data and pid. The created structure will be used with function *writeToFile.* As the name says, this function is used for writing data into files. Because writing the file requires disk IO, files should be written one at the time. Also, the function will add a log entry into the log file, which will be used by each thread, so it can be accessed by thread at the time to prevent file corruption.

The *writeToFile* function extracts the pid and the data from the void structure, and they will be set into different values. The names of created files will contain unix timestamp and the pid of the client (the structure of the name is <timestamp>_<pid>). That's because the accuracy of unix timestamp is one second. Therefore, there will be problems with multiple files will be transferred at the same time. Also, using of pids only wouldn't work perfectly either, because as mentioned in the beginning of this diary, for some reason multiple data entries can have the same pid even though they're sent separately. Having the same name is problematic because there cannot be files with identical names in the same folder.

writeToFile also handles writing the log file as mentioned before. The thread will be mutex locked during the file creation and writing the data into the file. After this, the mutex will be unlocked and the next thread can access writing the file. The file will be written in the directory where the program was executed.

I think I learnt a lot during this process, more than I did with the first assignment. That's because the while the first assignment contained many familiar functions for me (for instance: reading and writing files), this assignment focused on more unfamiliar functions and things, like threading. At first, I thought that this will be hard for me because asynchronous functions in Python were quite hard for me to understand. But like I mentioned earlier, in C threading was even easier than in Python, at least in my opinion. On purpose, threading in writer is highlighted in this program because if the user wants to transfer a large file (over 1000 bytes), the system will sleep for five seconds after transferring the data. During this five seconds, other, smaller data can be transferred. Usage of mutex functions was interesting, too.

Threading (and mutex) was not the only interesting part of this program. FIFO is an interesting concept and utilizing it in a program was interesting and instructive for me. The structure and the type of a FIFO-pipe is pretty interesting in my opinion, because if one client receives the data sent through FIFO, other clients don't receive it.

The project also contains a library for logging related functions. The function *writeLog* requires text entry wanted to log and the identification of the log (whether it's client or writer log). This function very simple, it opens the wanted file and logs the entry there. As mentioned before, each writer thread uses the same logging file and that's why it needs to be locked.

The code was tested on both Windows Subsystem for Linux (using Ubuntu 20.04.3 LTS) and on a virtual computer running Ubuntu 20.04 LTS. There was no unittesting or other automatized testing, but I tested the program running client multiple times. During the development, the issue with using pids or timestamps was found, and it was fixed. Unittesting would be an improvement. I created a simple "stresstester" script, which had 256 entries of testing files, and the writer worked fine with WSL, neither crashes nor "duplicates" were found. Also, there were some interesting phenomena between WSL and a "real" Ubuntu: I had missed a *mkfifo* from client.c and tested it using WSL > everything worked fine. However, when trying to repeat this with an Ubuntu VM, didn't work (because of the missing *mkfifo*). Also, WSL appears to create a file with 0664-permissions, but Ubuntu VM set permissions to 0000 as default and they had to be changed afterwards (program does that by itself).

Overall, I'm very satisfied with the project. Even though this is not perfect, it works fine, and I have learnt using many new methods and functions in C / POSIX. There are some issues as mentioned in the beginning of the diary, but the basic functionality works fine without any problems. I am pretty sure that most of the listed problems could've been fixed if I had more time for this program. However, there are always some things which could always be better.