S-38.3600 AS2 Program Design &User Guide 244536_YunfengHe

The Design of the Data Transfer client and server is also illustrated in the Design Chart.

Program Design(Data Transfer)

Design considerations

solution A: not implemented

Server create one well-known FIFO and servers all purpose from listening to request to data transfer.

Use semaphore and signal to synchronize the server and client.

Problems: data headers to parse; race condition; most importantly not so parallel!

solution B: implemented

Use multiple FIFOS. Server's well-known FIFO for listening and accept request from clients. Client-specific FIFOs are used for Data Transfer.

Easier to implement; Can easily make use of per-client thread at server side; Fulfills assignment requirements

Solution C: not implemented

Use Unix domain sockets. write a server-client which acts just like ordinary http server and clients.

problem: easy to implement, lose the chance to practice FIFO usage.

Implemented Program Functions:(My program employs solution B)

DTserver :

- listening to request using well-known FIFO. Done
- Creating new thread for copying file(parallel task). Done
- Thread synchronization(implemented using mutex and signal). Done
- Semaphore for access control(writing access of the listening FIFO, work with the client). Done
- Data Transfer among processes through client specific FIFO. Done
- SIGPIPE handling. Default action and not modified (just let the client terminate)
- Server timeout. (Select() timeout is used so that I can check memory leak situation after the server return. Otherwise I can only kill the server. However select() timeout has helped created most of the surprises! takes me some effect to solve) Done
- Daemon server(changing the working Dir to `/tmp/DTserver` defined in DTlib.h). Done
- FILE locking. (check the **quote** part, also explained in my learning diary) Not done
- Mutex used to protect LOGFILE access. Done
- Temporary FIFOs and foler clean up. Done

DTclient:

- client-specific FIFO creation. Done
- Access control using Semaphore. (working with DTserver) Done

- Client timeout.(try to get semaphore to send request) Done
- Server-client synchronization through signal. (both for semaphore release and Data transfer) Done
- Data Transfer among unrelated processes, through client-specific FIFO. Done

  DTclient_helper: (maybe I can also write a shell script and run the client multiple times.)
- Facilitate the test of my DTserver and DTclient. Done
- Forking new processes. Done
- Code replacement to create parallel working instances of DTclient. Done

Design Details

The Design of the Data Transfer client and server is also illustrated in the Design Chart.

At the Server side:(DTserver)

- The server receives argv[1] as a timeout in minutes ./DTserver 10 gives 10 minutes of listening time.
- First the server process turns itself into a daemon, the daemon change the working directory into /tmp/DTserver. (All copied files and the LOGFILE will be located in the working directory of the DTserver.)
- The server receives argv[1] as a timeout in minutes ./DTserver 10 gives 10 minutes of listening time.
- Then the server makes a well-known listening FIFO whose pathname is defined as a Macro in DTlib.h . After opening the FIFO as RDWR, the server calls select to listen on the FIFO. Select()'s build in timer is set as a timeout of the server.(just try to play with select a bit). Also, the server makes and opens a well-known Semaphore, whose max holder value is set to one.
- Clients send their pids as requests to the server, after they successfully acquire a semaphore. When the client get the semaphore, they can write to the listening FIFO. However, for synchronization purpose, the semaphore a client hold is not released immediately after sending the request. After reading from the listening FIFO and get the pid of this new client, the server would then send SIGUSR1 to signal the client to release the semaphore. In this way, there's only one client request in the FIFO before the server read it, this certainly can prevent FIFO buffer overflow and makes it much easier for the server to decide how much to read at once.
- One thing important is that, this mechanism won't affect the speed of the entire program since the bottle neck is at the server side. The looping speed is how fast the server can start up new thread and do copying. whether or not there's more than one request in the FIFO buffer, the speed of starting up new thread to serve them is the same.
- When select() successfully returns, the pid of the new client is saved in a read buffer by the server. The server then start up a new thread routine and pass the buffer address to the new thread. Before the new thread's finish copying the content of the read buffer, the read buffer shall be protected. Because if the main thread read another request into the buffer when the new thread copy it, the content will be corrupted. Instead of complex thread synchronization APIs, a signal is used to perform the synchronization. after starting up a new thread, the main thread of the server would just hang their waiting for the signal from the new thread. As soon as the new thread finish copying the content of the read buffer(which hold the request), it then issues this signal. When the signal is caught, by the same process of course, the main thread would then tell the client to release the

listening FIFO semaphore and loop to read the FIFO again (This job of signaling the client has been moved to the thread_routine, to prevent problems like main thread timeout and break from loop, without signaling the client to release semaphore).

- When select() times out, the main thread breaks out of the reading loop, close the listening end of the FIFO and stop serving new coming clients. As the main thread no longer signal the client to release semaphore, new coming client won't be able to get the semaphore and would return after certain time.(this design is implemented in the code but commented out since some unidentified problems). The main thread would then sleep 30 seconds, waiting for working thread finish their job. then the main thread unlink the semaphore and do some clean up before calling pthread_exit();(pthread_exit is called so that main thread returns without shutting down working threads) Anyway, the select() timeout has made the server logic a bit more complicated; it make a clear clean up and exit more challenging.

At the client side:(DTclient)

- Client receives argv[1] as the filename of the file to be copied.(only 1 file is allowed by DTclient itself.)
- Before doing anything at all, the client first tries to make a client specific fifo, whose name is the pid of this client, under the /tmp/DTfifo directory.(server has already created the directory on start) This make sure that by the time the client send a request to the server, the client FIFO is already created.
- Then the client opens the well-known listening FIFO created by the server, as write only. Before try to send its pid as the request to the listening FIFO, the client tries to acquire the semaphore declared by the server. if it could not get the semaphore within 10 seconds(defined in DTlib.h, int try_sem(sem_t *id)), the client automatically shut it self down.
- After getting the semaphore, the client send the request(its pid) to the listening FIFO. Then waiting for the server's new working thread to send back a SIGUSR1.
- When catching SIGUSR1, the client then release the semaphore, so that another client could send its request.
- Then the client close the listening FIFO's write end; opens the client-FIFO as O_RDWR. After this the client send the filename to be copied to the server through client-FIFO. then it call a function defined as writefile() to send the file.
- A pipe's reading end is passed to writefile to check SIGUSR1 from the server, which is the synchronization signal give by the server. before open the file for reading, the writefile function would check the buffer length of the client-FIFO at run time, then perform the writing under the command(SIGUSR1) of the server's working thread.
- After copying is done, client perform some clean up and returns.

The DTclient_helper

- DTclient_helper saves misery for testing the program. it must works with DTclient as it would replace code with DTclient as it forks.
- ./DTclient_helper [file1] [file2] [file3] .....[filek] forks k more processes, who execute the DTclient binaries.
- I have written a counter so that when a forked DTclient returns, the DTclient_helper would know it, when all of those processes return, the DTclient_helper returns. However, this counter is based on signal and is not reliable. so I comments it

out."//dealing with signal is dangerous, this signal counter may not work because if more than one SIGCHLD arrives before the handler return(if only one, sigaction helps block it, if two, there's nothing sigaction can do), signal get lost anyway, there's no underlayer buffer to buffer more than one instance of the same signal in sigaction."(quoted from the comment of DTclient_helper.c)

- After the DTclient_helper has started all DTclient tasks, it calls pthread_exit to quit, so that the forked processes could run on.

User Guide(Data Transfer)

Installation

- Three executables will be generated by "Make", namely, DTserver DTclient DTclient_helper
- All three executables need DTlib.h. You can Make the executables under the same directory and make copies of DTclient_helper and DTclient to other directories where you wanna run the client_helper instances.
- DTclient_helper make use of DTclient, so make sure DTclient_helper has a copy of DTclient in its working directory
- "Make clean" to perform clean up
- compilation only encounters warning of mixed declaration and comment format.

Usage

DTserver:

- In terminal, change to the directory of DTserver binary and give command ./DTserver [argv[1]] to start the daemon server. where argv[1] is a number representing the server's listening timeout***, in minutes.***
- All copied files and the LOGFILE is located in **/tmp/DTserver.**(the working directory of the daemon, defined in DTlib.h) Temporary FIFOs are located in /tmp/DTfifo.

DTclient:

- ./DTclient [filename] to start the client.

DTclient_helper:

- make sure that DTclient_helper has DTclient in it's working directory. in the terminal, give command like ./DTclient_helper [file1] [file2]
- [file3]...[fileK] to start multiple parallel instances of DTclient.
- Trash files named trash1 trash2... are given in the folder for testing purpose. they are large enough as text files.

Testing method:

- start the DTserver by ***valgrind ./DTserver 10*** (this gives 10 minutes of server timeout)
- copy DTclient and DTclient_helper into two different working directories, open two terminal and use DTclient_helper to send request. for example: ***valgrind ./DTclient_helper trash1 trash2 trash3 trash4 trash5 trash6 ....*** after giving command at both terminal, press enter quickly one after another

- check ***/tmp/DTserver*** to see if the files are copied correctly. then use ps -e to see if there's any hanging client. Also can check the LOGFILE to see if the server shut down correctly after 5 and half minute(server stop listening after 5 minutes and wait another 30 seconds before return.)
- while copying, you can always check ***/tmp/DTfifo*** to see FIFOs being made and unlinked. when server returns, it tries to clean all fifos and the /tmp/DTfifo directory.
- when valgrind returns, see the summary.
- Do stress test, give command one after another.
- Give the server enough time to run please.

Testing result:

- memory leak: not possible at both server and client side
- stress test: ok, robust.
- possible problems: When server timeout while client sending request, there could be unexpected blocking. Although I have already spend a lot energy and solved many blocking problems.
- Select() timer has made things fairly complicated, I can always modify the code and give select() a NULL argument to disable the timer. Sorry I haven't implement a user interface for this! not so necessary as you wanna valgrind the DTserver anyway.

Quote from my S-38.3600_As2_LearningDiary_244536_YunfengHe

File lock problem:

Fcntl(); provide a way to put read or write lock on a file(fd). this is useful when more than one process wanna write to the same file. My Data Transfer program don't really use the file lock, since I consider fwrite() quicker. I made a test of two instances of my client writing the same file for the server to copy(a 102MB text file), and the copied file is not corrupted. This is because, the two files opened by the two clients under different folders are exactly the same. this means at the server two threads would write the same thing at the same offset of the file(regardless of who write first, it doesn't matter, it's like draw the same line at the place on a paper twice). if the two files with the same name are not the same in the first place, then the COPY file will be corrupted. Anyway, trying to make a copy of two files with the same name at the same time is really silly anyway. that's why, File lock is only advisory lock and not all the time needed!

Feedback please(about memory leakage):

When a working thread (not the main thread) of the server done its job, return is called instead of pthread_exit, so that there's no memory leakage. When I used pthread_exit, I encounter memory leakage, what's the problem??

The Memory leak problem in detail

when server time out and returns, valgrind tells me there's mem leak, by malloc and calloc, but I really cannot tell what's wrong in the code, I freed allocated memorys, and malloc is thread safe when compiling with pthread. feedback please!!!(mem-leak won't happen if there's only one thread working. multiple client caused this somehow. always 5 blocks and 936 bytes lost.) **This problem is solved.** When thread is done with it's copy job, I wrote pthread_exit(NULL), instead of calling return. This somehow create problems

as pthread_exit() do not do any clean up at all. I need only to use pthread_exit() when the main thread return, to prevent from the timout mechanism shutting down other working thread prematurely. In the thread routine, use return NULL to return.