2016 System Programming Assignment 1

Due: 23:59 Tue, October 25, 2016

1 Problem description

It's very important to tackle the issue in multiplexing. In this assignment, you are asked to make a read/write server support many clients to read and write file contents at the same time. You need to utilize several system calls and functions, including `fcntl()`, `select()`, `accept()`, and so on, to complete this assignment.

The sample source code is on CEIBA. You don't need to write the whole code from the very beginning, since we have already handled most of the parts. The code we give you has been able to run a very simple read/write server. The only thing you need to do is to modify the code in order to implement I/O multiplexing, dealing with many requests at the same time, rather than being blocked by only one request.

Also the file control should be carefully handled, such as file lock. You are requested to protect the file from other's overwriting when the file is being read, and similarly you should prevent the file from other's reading when the file is being written. However, two read requests on the file at the same time is okay, since that does not cause any synchronization problem.

Roughly speaking, alteration of existing codes after the line `======================` is unneeded, while reading them may help you understand better the intent of each function. For more details of the existing code, you should refer to the slide. You may want to add functions to clarify your code. If so, please report what you've added in the `README.txt`.

2 How to run the sample program

- Compile
  You first need to compile server.c in order to produce execution files. If you take a look on `Makefile`, you will see the compiling it directly produces `write_server`, while compiling it with `-D READ_SERVER` produces `read_server`. For the first try, simply type this command:
  $ make

- How does the simple servers run?
  You first need to start up the servers. Let's take the read server for an example.
  $ ./read_server port_num
  where `port_num` is the port that you want assign to the server, say, 4000, or other port number unused. Similarly, `write_server` can be executed in the same way.

- Client-side input and testing
  Use `telnet` to connect to your server. Take the read server as the target for this example.
  $ telnet host port_num
  where `host` is where the server starts, say, `linux1.csie.ntu.edu.tw`. The argument, `port_num`, should be the same as the ones where you starts the read server. And then you can send your target filename, and one enter to indicate the end of the input. For example, if there exists a file named `sp_sample_file.txt`, which contains the only string `"sp_sample_content"`, the command should be like

  `sp_sample_file.txt`

  The read server shall return the content of your specified file. That is, the client should receive

  `sp_sample_content`

  On the other hand, if you connect to a write server, the filename you send to the server will be the name of a newly created file, and after that you are able to send arbitrary length of messages, which will all be written into that file, until the connection is closed.

3 Sample execution

blue texts indicates input by user.

- Read-server side (suppose in linux1.csie.ntu.edu.tw)
  $ ./read_server 4000
  ```
  starting on linux1, port 4000, fd 3, maxconn 1024...
  ```

- Write-server side (suppose in linux1.csie.ntu.edu.tw)
  $ ./write_server 4001
  ```
  starting on linux1, port 4001, fd 3, maxconn 1024...
  ```

- Client side (connecting to the read server)
  $ telnet linux1.csie.ntu.edu.tw 4000
  ```
  Trying 140.112.30.32...
  Connected to 140.112.30.32.
  Escape character is '^]'.
  ```
  sp_sample_file.txt
  ```
  ACCEPT
  sp_sample_content
  Connection closed by foreign host.
  ```

- Client side (connecting to the write server)
  $ telnet linux1.csie.ntu.edu.tw 4001
  ```
  Trying 140.112.30.32...
  Connected to 140.112.30.32.
  Escape character is '^]'.
  ```
  sp_sample_created_file.txt
  ```
  ACCEPT
  ```
  sp_sample_created_content
  ^]
  telnet> **quit**
  ```
  Connection closed by foreign host.
  ```

In order to disconnect with the write server, you have to type ^] (ctrl+]) first, and enter quit. After disconnection, you should find a newly created sp_sample_created_file.txt in the same directory containing the string

`sp_sample_created_content`

4 Format for request and response

The request to both read_server and write_server would be in the following format:
`[filename]`↲
`[request content] (when writing, allows multiple lines)`
The response for both read_server and write_server should be in the following format:
`[status]`↲
`[request content] (when reading, allows multiple lines)`

where ↲ is a newline character, namely ascii character '\012'. `[filename]` would be an ascii string without a newline, and could have length at most 30. `[status]` should be `ACCEPT` if the request is succeeded, or `REJECT` if it isn't. (e.g., the file is locked by another request.)

For the read server, `[request content]` would be empty, and `[response content]` should be the content of the file named `[filename]`. It's guaranteed that the file specified by `[filename]` for read server exists.

For the write server, `[requestcontent]` would be the content to be write to the file named `[filename]`, and have no `[response content]`. If the file already exists, you should overwrite it, or if it doesn't exist, create a new file. The file should be remained a lock when reading from/writing to the file. Try to use the system call `fcntl()` to lock the file. The file must be locked when the filename is specified, and unlocked when the connection from the client is closed.

5 Tasks and scoring

There are 6 subtasks in this assignment. By finishing all subtasks you earn the full 6 points.

1. read_server returns the file content correctly. (1 point)
    - There would be only one request at a time for this subtask.
2. write_server reads the request and writes to file correctly. (1 point)
    - There would be only one request at a time for this subtask.
3. Two requests issued to read_server. (1 point)
    - A read request A connects to read_server but hasn't send the request.
    - Then a read request B connects to read_server and sends the request.
    - Server should be able to respond to Read request B.
4. Two requests issued to write_server. (1 point)

- A write request A connects to write_server, specifies the filename but hasn't send the content.
- A write request B connects to write_server and specified the same filename.
- Server should reject request B.

5. Two requests to read_server and write_server at the same time. (1 point)
   - A write request A connects to write_server, specifies the filename but hasn't send the content.
   - A write request B connects to read_server and specifies the same filename.
   - Server should reject request B.
6. Two requests to write_server and read_server at the same time. (1 point)
   - A read request A connects to read_server and sends the request, but is still waiting for response from the server.
   - A write request B connects to write_server and specifies the same filename.
   - Server should reject request B.
7. Multiple requests to write_server and read_server at the same time. (1 point)
   - Client would issue about 100 requests in a short interval, to both servers.
   - Servers should still be working properly and not crash.

6 Notes

- As a kind of system resource, a port may be occupied by another process and you may get message like

  ```
  bind: Address already in use
  ```

  when trying to bind your server to an occupied port. Choose another port you like in this situation. Port numbering from 0 through 1023 are usually privileged and you should avoid using them. The range from 3000 to 30000 is suggested.
- The file size may be larger than the buffer size in read server. Try to **NOT** increase the buffer size, but read the file into the buffer just before sending it to client.

7 Submission

Your assignment should be submitted to the course website by the due. Or you will receive penalty. At least three files should be included:

1. `server.c (as well as all other *.c)`
2. `Makefile`
3. `README.txt`

Since we will directly run your `Makefile`, therefore you can modify the names for `*.c` files, but `Makefile` should **compile your source into two executable files named read_server and write_server.**
In `README.txt`, please briefly state how to compile your program, and anything you have done besides the basic functionality.
These files should be put inside **a folder named with your student ID(lowercase)** and you should compress the folder into a `.tar.gz` before submission. Please do not use `.rar` or other file type. The commands below will do the trick. Suppose your student ID is b05902000:
$ b05902000
$ cp Makefile README.txt *.c b05902000
$ tar zcvf SPHW1_b05902000.tar.gz b05902000
Please do **NOT** add executable files to the compressed file. Error in the submission file (such as files not in a directory named with your student ID, compiled binary not named read_server and write_server, or so on) **may cause deduction of your credits.**

8 Punishments

- Plagiarism
  Plagiarism is strictly prohibited.

- Late punishment
  5% of credits you get in this assignment 1 day.