

# Homework #1

B05902120 / Yu-Ting, TSENG

Oct 10, 2017

## 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`.

## 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 server. 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.

## Sample Execution

- Read-server side (suppose in `linux1.csie.ntu.edu.tw`):

```
User: ./read_server 4000
```

```
Console: starting on linux1, port 4000, fd 3, maxconn 1024...
```

- Write-server side (suppose in `linux1.csie.ntu.edu.tw`)

```
User: ./write_server 4001
```

```
Console: starting on linux1, port 4001, fd 3, maxconn 1024...
```

- Client side (connecting to the read server)

```
User: telnet linux1.csie.ntu.edu.tw 4000
```

```
Console: Trying 140.112.30.32...
Console: Connented to 140.112.30.32.
Console: Escape character is '^]'.
User: sp_sample_file.txt
Console: ACCEPT
Console: sp_sample_content
Console: Connection closed by foreign host.
```

- Client side (connecting to the write server)

```
User: telnet linux1.csie.ntu.edu.tw 4001
Console: Trying 140.112.30.32...
Console: Connected to 140.112.30.32.
Console: Escape character is '^]'.
User: sp_sample_created_file.txt
Console: ACCEPT
User: sp_sample_created_content
User: ^]
User: telnet > quit
Console: 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.
```

## 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]
[response content] (when reading, allows multiple lines)
```

where [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, `[request content]` 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.

## 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 `read_server`. (1 point)
  - A read request A connects to server but hasn't send the request.
  - Then a read request B connects to server and sends the request.
  - Server should be able to respond to Read request B.
4. Two requests issued to read server. (1 point)
  - A write request A connects to server with `filename` but hasn't send the content.
  - Then a write request B connects to server and specifies 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 server with `filename` but hasn't send the content.
  - Then a read request B connects to 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 waiting.
- Then 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.

## 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.