System Programming 2013 (CC)
Assignment 2
Multiplexing Web Server

Due: 23:59, 10/31, 2013 (UTC+8)

I. Problem Description:

Athletic or not, people nowadays surf much every day, at least on the Web. According to a research of Yahoo! Taiwan, their homepage is visited more than 300 million times every day, yielding more than 3,000 requests per second. It is therefore very important to handle the issues in multiplexing during the design of a Web server.

In this assignment, you are asked to enhance a simple Web server with the capability of multiplexing. It means multiple requests will need to be served at the same time.
You need to utilize several system calls and functions, which may include `dup()`, `dup2()`, `select()` and so on to complete this assignment.

You could find the source code of the simple Web server on the course website.

Run the compiled binary server as:

```
$ server port_num log_file
```

The client would access the `server` with port *`port_num`* while the `server` would store its working log inside *`log_file`* after the `server` is up.

How Does the Simple Web Server Run?

Roughly speaking, alteration of existing codes beyond `main()` function is unneeded, while reading them may help you understand better the intent of each function.

The simple Web server initializes itself with a socket descriptor retrieved from the system after started and listens to the designated port. The server then allocates a set of *`http_request`* objects to manage all possible file descriptors, an object per descriptor, as clients will communicate (send requests and receive responses) with the `server` through **socket descriptors**. It then begins to `accept` connections and use the descriptor returned to communicate with its clients. The request processing function, `read_header_and_file`, handles almost every single task for a request: it parses the request header and reads the

file requested into the *buf* of the *http_request* object associated with that request. For the last step it just writes out the buffered data to the clients. It's just that simple!

Not That Simple, Perhaps

There are actually many details in the implementation of all these functions.

In this assignment you should focus on manipulation of file (and socket) descriptors and non-blocking I/O to accomplish multiplexing as other requirements. The two data structures: *http_server* and *http_request* would be the most important part now as they encapsulate most of the information you need, for example *buf*, *status*, *conn_fd*, and *listen_fd*.

Most of the modification will locate inside the big `while` loop, but you may want to add your functions to clarify your code. If so, please **put them at the very end of your source** and mention what you' ve added in your report.

II. Tasks and Scoring:

There are 4 subtasks, sequentially from 1 to 4, in this assignment. By finishing all subtasks you earn the full 15 points.

1. Output the header and data (`small` file) in *http_request.buf* to the client (4 pts)
   - You should allow multiple requests for small files at the same time.
   - Use `write` to communicate with the client.
   - Client should `GET /small HTTP/1.1` correctly.
   - You may notice that the `large` file would be cut. That is fine in this subtask. You will solve this in the next subtask.
2. Output the header and data (`large` file) in *http_request.buf* to the client (6 pts)
   - Client should `GET /large HTTP/1.1` correctly.
   - The simple Web server malfunctions when requested large files. You should allow multiple requests for large files at the same time.

3. Your `server` will output runtime information such as connecting clients and file requested to the *`log_file`* by calling `printf()` and `dup()/dup2()` (1 pt)
   - Read the next section for the contents and formats.
   - `fprintf`, `freopen`, and similar functions other than `dup/dup2` are prohibited.4
4. Report (including problem description, solution and discussions) (4 pts)
   - Your server will process requests simultaneously (perform multiplexing). How do you achieve the goal? Explain the structure of your code and how does it satisfy the requirements.

★ Bonus would be given for extra improvements and fixes of severe bugs.

III. Execution Input, Output, and Testing:

Server Side Input

You only need to start it up. No further input is required.

```
$ server port_num log_file↵
```

Server Side Output

You should output runtime info into the *`log_file`*. There is no restrictions on content and format, but you may want to output **hosts**, **ports**, **file descriptors**, **timeouts**, **read/write ready**, **read/write completion**, and **timestamps**, as well as **path**, **log file name** and **port listened**. You may also output the information to the screen if you want to.

Client Side Input & Testing

Use `telnet` to connect to your `server` and send `GET` request afterward.

```
$ telnet host port_num↵
GET /file_name HTTP/1.1↵
↵
```

You could use browsers instead, with the following URL:

```
http://host:port_num/file_name
```

The `host` here depends on which machine you use. For instance, if you runs the `server` on `bsd3`, it would be `bsd3.csie.ntu.edu.tw`.

This could be used for testing your program.

Client Side Output

The output will consist of HTTP header and the file content. For example:

```
HTTP/1.1 200 OK
Server: SP TOY
Date: Wed, 17 Mar 2010 10:03:30 GMT
Content-Length: 32
Connection: close

small
```

is the response as an example.

IV. 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 1,023 are usually privileged and you should prevent using them. The range from 3,000 to 30,000 is suggested.

V. Environment:

1. Use one of the machines, `linux1` – `linux15` and `bsd1` – `bsd6`, as your host.
   For the “odd” numbered class members, please use machines odd-numbered, e.g. `linux1`, `linux3`, …; for the “even” numbered class members, please use the other machines.
2. Name your binary code `server`.
3. Put the `small` and `large` files under the same path as `server`’s to test whether your `server` runs correctly.

VI. Submission:

Your assignment should be submitted to the ceiba system by the due. These three files should be included exclusively:

1. `Makefile`
2. Report.pdf
3. `sp_hw1_httpd.c`

The `Makefile` should compile your source into an executable file named `server`.

Put your student ID (with lower-case letters) and name in a comment at the first line of your source code, `sp_hw1_httpd.c`, with the following format:

```
/* STUDENT_ID NAME */↵
```

These files should be put inside a folder named with your student ID and `tar`ed before submission. The commands below will do the treak.

```
$ mkdir STUDENT_ID↵
$ cp Makefile Report.pdf sp_hw1_httpd.c STUDENT_ID↵
$ tar -zcvf [SPHW1]STUDENT_ID.tar.gz STUDNET_ID↵
$ rm -r STUDENT_ID/↵
```

You may want to peek your tarball with this command:

```
$ tar -zcvf [SPHW1]STUDENT_ID.tar.gz STUDNET_ID↵
```

A Sample Student

For example, a student b97902000 林玲綾 does these:

```
$ mkdir b97902000↵
$ cp Makefile Report.pdf sp_hw1_httpd.c b97902000↵
$ tar -zcvf [SPHW1]b97902000.tar.gz b97902000↵
$ rm -r b97902000/↵
```

And list her file with (… Oops, late submission.)

```
$ tar -tvf [SPHW1]b97902000.tar.gz↵
drwxr-xr-x 0 b97000 student   0 Mar 31 b97902000/
-rw-r--r-- 0 b97000 student 932 Mar 17 b97902000/sp_hw1_httpd.c
-rw-r--r-- 0 b97000 student  68 Mar 17 b97902000/Report.pdf
-rw-r--r-- 0 b97000 student  19 Mar 17 b97902000/Makefile
```

And the first two lines of her `sp_hw1_httpd.c` would be

```
/* b97902000 林玲綾 */
#include <unistd.h>
```

Plagiarism

Plagiarism is strict prohibited.

Late punishment

While your credits will be deducted for delaying (5% per day), a late submission will still be much better than absence.