Socket Programming

Getting Set Up

1. As usual, start by going to the homework page, read the homework 5 description, and check out your repository with the provided link.
2. If you're using:
    1. `systemsX` to work on this homework, you'll need a unique port number to run your webserver, which is mentioned in spreadsheet at the bottom of the homework 5 description.
    2. Docker container, you can use any port number in your container, but the default forwarded port metioned in the `devcontainer.json` is `5000`.
3. In this homework, you will be developing a webserver. This requires you to understand the basics of socket programming, and HTTP. In this lab session, we will be discussing the network programming aspects of the homework.

Socket Programming

1. There is an excellent guide to network programming at http://beej.us/guide/bgnet/. You should definitely read this guide. Go read it now.
2. Open `homework5.c` and locate the main function. We will go through this line by line and discuss the network functions it uses. Please read the actual main function as well as my description here - main contains many helpful comments describing the details of what we are doing.
3. The first thing we do is get the port number from the command line arguments, and convert it from a string to an integer using the atoi function. (Use "man atoi" to find out more about this function.)

```
int port = atoi(argv[1]);
```

4. Next we need to create a socket for clients to connect to our webserver. We pass in parameters specifying we want an IPv6 socket, and that it should be streaming (i.e. TCP).

```
int server_sock = socket(AF_INET6, SOCK_STREAM, 0);
```

5. Now, we need to set the socket options to allow reusing the port instantly after the socket is closed. To disable the wait time for rebinding the socket, we set the following socket options:

```
retval = setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, &reuse_true, sizeof(reuse_true));
```

   Visit the man page of `setsockopt` to understand what each of the function parameters mean.

6. Next, we bind our socket to a port. Here we're saying our socket is IPv6, and that it should use our port number.

```
struct sockaddr_in6 addr;   // internet socket address data structure
addr.sin6_family = AF_INET6;
addr.sin6_port = htons(port); // byte order is significant
addr.sin6_addr = in6addr_any; // listen to all interfaces
retval = bind(server_sock, (struct sockaddr*)&addr, sizeof(addr));
```

7. Now we have to actual listen to our socket, in case anyone tries to connect to us. We do this like so:

```
retval = listen(server_sock, BACKLOG);
```

8. If anyone connects to us, we're going to accept the connection, and create a new socket to talk to them on. We do this so we can continue listening for connections on our existing socket. This means our existing socket, which were listening on, keeps its port number, and people can still connect to us using that port number.

```
sock = accept(server_sock, (struct sockaddr*) &remote_addr, &socklen);
```

9. At the end of the program, we also need to close the file descriptor of the socket to allow other sockets to bind to the address later. If the socket keeps on listening, the system will keep continuing to accept connections on this the socket. To close the socket, you need to do is uncomment the following line:

```
close(server_sock);
```

10. Answer the questions on socket programming on Gradescope.
11. Now in our code we call a function that will actually use the socket. Before we look at that function, let's learn a little about the HTTP protocol.

HTTP Protocol

1. The client will start by sending us a request that will look like

```
GET <filename> HTTP/1.0
\r\n\r\n
```

2. For our purposes, the things we need to be able to do are to tell when the request has ended (which we can do by look for the control sequence it will always end with), and being able to find the file name. Because we are super nice, we have provided you with the `parseRequest` function which will take in an HTTP request and return a filename.
3. Assuming the server has the file, it will respond the following header, followed by the bytes which make up the file.

```
HTTP/1.0 200 OK\r\n
Content-type: text/html; charset=UTF-8\r\n\r\n
```

4. `text/html` will need to be replaced with the appropriate Content-Type for non HTML files.

serve_request

1. The code for `serve_request` is below. You can see that we read a request from the client, parse it, send a response, and then send the requested file to the client. Read over the code and make sure you understand exactly what it is doing, and what fixes you need to make.

```
void serve_request(int client_fd)
{
int read_fd;
int file_offset = 0;
char client_buf[4096];
char send_buf[4096];
char * filename;
char *requested_resource;
memset(client_buf, 0, 4096);
while (1)
{
    file_offset += recv(client_fd, &client_buf[file_offset], 4096, 0);
    if (strstr(client_buf, "\r\n\r\n"))
    break;
}
requested_resource = parseRequest(client_buf);
snprintf(send_buf, sizeof(send_buf), request_str, "text/html; charset=UTF-8");
send(client_fd, send_buf, strlen(send_buf), 0);

// strip everything that isn't part of *the file name* off of the
// resource request string, and add a "./" to the beginning to make
// the path relative to CWD.
filename = requested_file(requested_resource);


// is /cgi-bin/ the beginning of this request? then execute a
// subprocess with posix spawn and pass the proper arguments via
// environment variables.
if (strstr(filename, "/cgi-bin/") && strtok(filename, "?")) {
    posix_spawn_file_actions_t actions;
    pid_t pid;
    char **newenv = malloc(sizeof(char *) * 2);
    char *env = malloc(1024);
    char ** argv = malloc(sizeof(char * ) * 2);
    argv[0] = filename;
    argv[1] = NULL;
    newenv[0] = env;
    newenv[1] = NULL;

    // TODO - WRITE CGI PART

} else{
    read_fd = open(filename,O_RDONLY);

    send_file(read_fd, client_fd);
}

return;
}
```

2. Answer the questions about this code on Gradescope.
3. Notice that inside the function `requested_file`, we append a `.` to each file. This causes us to look within the current directory for our files.
4. We want the webserver to take in a directory, and serve the files within that directory. So if you run `./homework5 PORTNO WWW` it should serve files not from the current directory, but from the `WWW` directory inside of the current directory. Note that right now the webserver code ignores the second command line parameter - you should add code to the main method to fix this.
5. Notice we call a function `send_file` inside `serve_request`. The current implementation of `send_file` sends too many bytes. You need to fix this code so that it sends only the amount of the bytes that are being read. Note: The return values of `read` and `send` are important to detect the failure and send the correct amount of bytes.
6. To complete the CGI requirement, we have provided the code to populate the imporant parameters to create a new child process. You need to parse the `requested_resource`, and prepare `QUERY_STRING` environment variable based on the query parameters. Once you have done that, you can use `posix_spawn` to create the child process and return its response to the client. Note: To understand how to use the `QUERY_STRING` environment variable, take a look at the `hello-world.py` in `WWW/cgi-bin` directory.

Running and Testing Web Server Code

1. If you are running in a Docker container evironment you can run and test your program locally. However, if you're using `systemsX`, you have two options to run and test your program:
    1. Forward the assigned port when initiating your SSH session using

       ```
       ssh -N -f -L localhost:port:localhost:port netid@systemsX.cs.uic.edu
       ```

       which makes `http://localhost:port` on `systemsX` accessible on your local machine (e.g. browsers or `curl`).
    2. Test your program on `systemsX` using command-line tools such as `curl` (e.g. `curl http://localhost:port`) by either logging in two different terminal windows or by using a terminal multiplexer like `screen` or `tmux`.
2. To test the current functionality, read the comments and fix the mentioned changes in `Socket Programming` section. You can now run your web server by exectuing `./homework5 PORTNO WWW` where `PORTNO` is your port number.
3. Use `curl` and type

   ```
   curl http://localhost:PORTNO/index.html
   ```

   You should see a response containing an HTML page!

Peer evaluation rubric

You can find your pairings for the lab in this sheet. Please grade your peers out of **1 point** using this form.

Session	Task	Points
Session A	Discuss questions 1, 2, and 3	1 point
Session B	Discuss questions 4, 5, and 6	1 point

Total grade calculation

Task	Points
Turn in a Gradescope assignment	6 points
Discuss your answers	1 point
Evaluate another student	1 point
Total points	8 points