CS 361: Computer Systems **Lab Session 8 Worksheet**
Fall 2019

Getting Set Up

1. As usual, start by going to the course website, reading the homework 5 description, and checking out your repository with the provided link.
2. You will each be assigned a unique port number to run your webserver on. Talk to the TA or check the list on piazza to get your port number. Use only your assigned port number for this assignment! Using a different port number will break things both for you, and for other students.
3. In this homework, you will be developing a multi-threaded webserver. This requires you to understand the basics of both multi-threading, and socket programming, and HTTP. In this lab session, we will be discussing the network programming aspects of the homework. We will cover threading in the next lab session. If you'd like to get a head start on threading, check out the provided file `thread_example.c`.

Web Server Code

1. `homework5.c` contains a single threaded webserver which serves html files from its current working directory.
2. To test its current functionality, run `./homework5 PORTNO WWW` where `PORTNO` is your port number.
3. Open a web browser and type in "http://systems.cs.uic.edu:PORTNO/WWW/index.html" with PORTNO replaced by your portnumber. You should see a website!
4. Unfortunately homework5.c is a very limited webserver. It only looks in its current directory, it only serves HTML files, and it's single threaded. You will be fixing all of these things.
5. To complete this lab, you will need to do the following:
   (a) Have the webserver check which directory is passed in as a parameter, and serve files from the directory.
   (b) Have the webserver correctly respond to requests for non text/html files.
   (c) If the request is for a directory, check if there is an index.html file in that directory. If so, return that file.
   (d) If there is a request for a directory and it does not contain index.html, automatically generate an html file listing all the files in that directory, and linking to them.
   (e) If there is a request for a file that doesn't exist, generate an attractive 404 page.
   (f) Add multi-threading to the webserver.
6. After this lab you should have all of the information you need to do everything except adding multi-threading. You should aim to complete at least the first 3 of these tasks this week.

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 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. 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));
   ```

6. 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);
   ```

7. 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);
   ```

8. We will go over all of the details of network programming in greater detail in class this week. If you want more information now, please consult the beej networking guide.

9. Answer the questions on socket programming on Gradescope.

10. 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 <other stuff we'll talk about later> \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.

   ```
   void serve_request(int client_fd){
     int read_fd;
     int bytes_read;
     int file_offset = 0;
     char client_buf[4096];
     char send_buf[4096];
     char filename[4096];
     char * requested_file;
     memset(client_buf,0,4096);
     memset(filename,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_file = parseRequest(client_buf);
     send(client_fd,request_str,strlen(request_str),0);
     // take requested_file, add a . to beginning, open that file
     filename[0] = '.';
     strncpy(&filename[1],requested_file,4095);
   ```

```
  read_fd = open(filename,0,0);
  while(1){
    bytes_read = read(read_fd,send_buf,4096);
    if(bytes_read == 0)
      break;

    send(client_fd,send_buf,bytes_read,0);
  }
  close(read_fd);
  close(client_fd);
  return;
}
```

2. Answer the questions about this code on Gradescope.

3. Notice that we append "." 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. Try modifying the code in `serve_request` so that instead of looking within the current directory, it looks for files within the directory you pass in as a command line argument.

6. Next, work on being able to send non-text file types. This will necessitate changing the content type within the response that you send. Your textbook contains a list of different content types which may be helpful on page 949. You can also see a very complete list at http://www.freeformatter.com/mime-types-list.html.

Handling Directories

1. Clone the github repo at "https://github.com/uicsystems/directory_reading_example.git" for examples of the kind of file/directory handling you will need to do.

2. Read through `example.c`. Note how it uses the stat call to first check if a file exists, and then to check if its a directory.

3. You can read the contents of a directory using the `opendir()` and `readdir()` calls. Together they behave like an iterator, that is, you can open a (DIR *) with `opendir()` and then continue calling `readdir()`, which returns info for one file, on that (DIR *) until it returns `NULL`. Work first on identifying if a directory has an `index.html` file and returning that file, and next on generating and returning a list of its contents. We provide you with a set of strings (char * index_) which may be helpful in creating the page for directory listings.

4. Answer the questions about file handling and directories on Gradescope.

Testing

1. If you are on campus, you should be able to access your webserver in a web browser.
2. However, we block traffic from off campus to certain ports, for security. There are two ways you can test your code while off campus:
   (a) Use the VPN to connect to campus. This will cause our network to believe your computer is on the campus network, and should allow you to use a web browser to connect.
   (b) SSH to systems.cs.uic.edu or bertvm.cs.uic.edu or any other on campus network, and use curl or wget to programmatically retrieve files from your webserver.