CS 105
Web Server

1 About Hints

There is a large "Hints" section (Section 7) at the end of this handout. Be sure to read the entire handout and the hints before starting work, and refer back to the hints frequently while you are writing and debugging your program.

2 ABOUT SECURITY

The Web server you will write is willing to send arbitrary files to its clients; it is very low-security. For that reason, DO NOT leave your server running longer than necessary to test it. Also, be **certain** that you only run it on Wilkes. We will periodically scan Wilkes for leftover servers and kill them.

3 Introduction

A Web server is a program that accepts requests in the HTTP format and sends responses using the same protocol. A modern server can handle many kinds of requests and many ways of producing a response. We'll attack a simpler problem.

In this lab, you will write a simple Web server that can send the contents of text and HTML files to a client. Your server will be powerful enough to send simple Web pages to a browser—but not good enough to implement amazon.com! To keep things simple, we will only handle the original HTTP protocol, HTTP/1.0, which all browsers can speak. The lab will help you understand network programming basics, the HTTP protocol, and string processing in the C language.

For extra credit, you can upgrade your server so that it uses threads to handle multiple clients concurrently.

3.1 Logistics

As always, you must work with your partner. Handin will be electronic, using `cs105submit`.

3.2 Handout

The handout is distributed in a `tar` file named `network-handout.tar`, which you will find linked from the lab Web page. Start by copying `network-handout.tar` to a (protected) directory in which you plan to do your work. Then give the following command:

```
tar xvf networklab-handout.tar
```

This will cause a number of files to be unpacked in the directory:

`Makefile` A Makefile that will build your Web server. You should always compile using `make` so that you compile with the correct options.

`index.html` A trivial HTML file that you can use for testing.

`webserver.c` A skeleton for the Web server (see below). This is the only file you will hand in.

`networklab.pdf` A copy of this writeup.

At the top of `webserver.c` is a comment where you can put your names. Just after all the `#include` statements is a variable named "`team`", which you should modify to contain your CS login IDs. Do both now, before you forget!

3.2.1 Manual Pages

Remember that you can read a description of any Unix command or function by using the `man` command ("`man man`" is always a fun thing to do, although the modern version has way too many options). By default the manuals are presented to you one page (screen) at a time; you can move to the next screen with the space bar and get out of the manual page by typing "q". (The arrow keys also work, and "h" will give you help on advanced features.) We recommend that you look at manual pages in a tall window.

Although manual pages can be found online, we will warn you that the online versions are sometimes inaccurate or confusing. (To be fair, they are occasionally better-written, so if you have trouble understanding a certain function then a Web search might be useful.)

The Unix manual is divided into sections; for our purposes the most important ones are Section 1 (user commands to be issued at the command line), Section 2 (system calls), and Section 3 (library functions). If you want to read about `open`, which is a system call, you can type "`man open`." However, sometimes that will give you an answer from the wrong section; in that case you can type "`man 2 open`" or "`man -s 2 open`" to explicitly say you want the page from Section 2.

By tradition, manual pages are referenced with the section number in parentheses, so when we speak of `strcmp(3)` you should type "`man 3 strcmp`" to learn about that function.

We *strongly* recommend that before you begin this lab, you at least briefly familiarize yourself with the following manual pages: `accept`(2), `open`(2), `fopen`(3), `read`(2), `write`(2), `malloc`(3), `strchr`(3), `strcmp`(3), `strerror`(3), `strlen`(3), `strncmp`(3), `strncpy`(3), `strpbrk`(3), and `strstr`(3).

At this point don't try to learn every detail about every function; instead, just give yourself an idea of what each one does.

One way to look at them is to cut and paste the following two shell commands into your terminal (type "q" after each man page to see the next):

```
man -s 2 accept open read write

man -s 3 fopen malloc strchr strcmp strlen strncmp strncpy \
  strpbrk strstr
```

4 Part I (60 points): Implementing a Sequential Web Server

In this part you will implement a sequential (one-client-at-a-time) Web server. Like any good server, it will write a log of its activity so that a system administrator (you) can later see what has happened. Your server will open a `socket` and `listen` for connection requests. When it receives a connection, it will `accept` it, read the HTTP request, and parse it to determine what file is being requested. It will then open that file and send it to the client, carefully following the HTTP protocol. Finally, it will close file and the connnection: under HTTP 1.0 the client can only fetch one Web page per connection.

If something goes wrong (for example, the file can't be found, the client sends a bad request, or the client tries to access forbidden files) your server must log the problem and return an appropriate error to the client, using the proper HTTP protocol. Search the Web for "HTTP response codes" to get a list of the kinds of errors your server can potentially return; you are only required to support a few of them (see Section 4.6).

To make the problem more tractable, we have provided scaffolding in `webserver.c`. That includes a complete copy of `open_listenfd` from `echoserver.c` so that you don't have to type it in yourself, and an `http_error` function that you can use to send error responses to the client. It also includes skeletons for most of the functions you will need to write, and a complete (for this part) logging function.

There are a number of places in the server that you will need to expand. Each of those is marked with a NEEDSWORK comment; you can search for that string to find the places you must modify.[1]

4.1 Logging

Your server should log the first line—the GET line—of each request it receives. In addition, it should log any unusual situations it encounters, such as bad requests, HTTP errors, clients closing the connection early, etc. The exact set of events to log is up to you.

We have provided a function, `write_log`, that will format a log entry and write it to a log file. (It is your responsibility to set up the log file itself; do that in the beginning of `main`.) The arguments to `write_log` are:

`args` A pointer to the argument structure that was created by `main` when the connection was accepted.

[1] You will also need to add some variable declarations; we didn't include NEEDSWORK comments for those.

`message` A string containing a message to be written to the log file. An example would be `"Sending file to client:"`

`data` A second string that, if not NULL, will be appended to the first (with a space separating them). For example, `data` might contain the name of the file being sent to the client. For convenience, `data` is allowed to end with or without a newline; `write_log` will append a newline if there isn't one there already.

4.2 Port Numbers

Your server should listen for its connection requests on the port number passed in on the command line:

```
unix> ./webserver 15213
```

You may use any port number p, where $1024 \leq p \leq 65535$ and p is not currently being used by any other system or user service (including other students' Web servers, or forgotten copies of your own server). See `/etc/services` on Wilkes for a list of the port numbers reserved by other system services. **We strongly suggest that you use one of your team's login ID numbers (see the `id` command) to avoid collisions with other students.**

4.3 The HTTP protocol

HTTP/1.0 is a request/response protocol: a client sends a request, and the server sends a response. In both cases, the message contains four parts. The first three are encoded in pure ASCII, and consist of zero or more lines. Each line is terminated by a **carriage return** *and* a newline (in C terms, `"\r\n"`). The four parts are:

1. A single line that identifies the nature of the request or response.
2. Zero or more "header" lines, each of which contains a nonblank string, a colon, a blank, and parameter information.
3. A single blank line consisting of just `"\r\n"`. (Note that since the previous line also ended with `"\r\n"`, the net result you will see or generate is `"\r\n\r\n"`.)
4. For responses only, an arbitrary amount of data, which may be in any format (e.g., text, image, sound, video, PDF, etc.). The format of the data is defined in a header line. Note that the data can contain arbitrary bytes, including null bytes (`'\0'`) and newlines. Your code may not make *any* assumptions about what is and is not allowed in the data; in particular, using string-style functions like `strlen` and `fgets` will cause your server to misbehave.

Item 4 is not present in an HTTP request but should always be present in a response.
An example of a near-minimal request is:

```
GET /index.html HTTP/2.0
User-Agent: CS 105 webget platt+wwart
Host: www.cnn.com:80
```
blank line

This request says that a Web client is contacting `www.cnn.com` on port 80 and asking to fetch a file named "`index.html`". The client also politely identifies itself ("User-Agent") with the string "CS 105 webget platt+wwart".

An example of a small response (not from CNN!) is:

```
HTTP/1.0 200 OK
Server: CS 105 Web server platt+wwart
Connection: close
Content-Type: text/html

<html>
This is a minimal Web page.
</html>
```

Here, the first line says that even though the client might have asked for a more advanced version of HTTP, the server is going to stick to HTTP/1.0. The "200 OK" part is a numeric response code (200) and its English translation ("OK", i.e., everything worked). The header lines—everything up to the blank line—identify the Web server software (including the CS 105 team name); the "Connection: close" line says that the client should close the connection after receiving the data; and the "Content-Type" line says that the response is HTML data, i.e., a formatted Web page. These are the only headers that your Web server needs to generate. After the blank line, the Web page itself appears. In this case, the file `index.html` is copied verbatim to the client.

As a practical matter, your Web server can ignore all of the header lines in the request that it receives. It **must** read those lines, up to the blank line that indicates the end of the request, but it can be lazy and discard all of the options. The first line, which begins with GET, is the only one that matters. (Quality Web servers normally log the User-Agent and respond to the other fields, but that's too much work for us!)

Many headers are allowed in the response, but again we can get away with just a few. The server should identify itself out of politeness, and "Connection: close" notifies the client that the server is going to close the connection after it sends the data (but in truth the "HTTP/1.0" protocol says the same thing). The really important header is the Content-Type, and your server will need to offer at least two options there: `text/plain` and `text/html` (see Section 4.8).

4.4 A Threading Note

The supplied code contains a skeleton function for handling server requests. Because you'll be adding threading later, the skeleton is written on the assumption that it will run as a thread. Thus, you might find it easiest to call it as a thread (although it's not absolutely necessary to do so—change the "`#if 1`" in `process_request` to "`#if 0`" if instead you choose to call the `process_request` function directly from `main`).

Because it is set up for threading, `main` passes arguments to `process_request` in a structure of type `arglist_t`. That structure is allocated and initialized (`calloc`) in `main`, and passed by pointer to `process_request`. It is `process_request`'s responsibility to free that structure.

4.5 HTTP Request Format

Take note that the request format given above involves multiple lines, and is ended by a blank line. (Actually, the blank line is signaled by the moderately complicated sequence "`\r\n\r\n`". As mentioned, you can ignore the header lines and concentrate on just the first one (in `parse_uri`). You only need to handle GET requests. However, you need to be cautious about the request format. In particular, don't assume that "HTTP/1.0" (or "/1.1" or "2.0"—you need to handle all of those) appears a certain distance from the end of the request, or that fields are separated by exactly one blank, or that the request only contains a single line. As a general rule, if you're counting characters from the beginning or end of the request, your code will be fragile and will be likely to break with real Web browsers.

We have provided most of the request-parsing code in `parse_uri`, but you must complete it.

4.6 HTTP Response codes

HTTP has approximately a zillion defined response codes that are designed to handle different situations. You can find descriptions of them on the Internet. They are divided into categories that are identified by the leading digit (i.e., the 200 series is for success and the 400's are for errors made by the client). You are welcome to generate as many different response codes as you wish, but you **must** generate the following minimum set of codes, as appropriate:

200 OK Sent when the client "did right" and you are feeding it a valid answer.

400 Bad request Sent when the client sends a syntactically invalid request.

403 Forbidden Sent when the client tries to violate a security restriction. The supplied code generates a 403 when the pathname contains the string "`../`", indicating that it is trying to access a file outside of the directory tree the server was run in. **DO NOT REMOVE THAT CODE.**

404 Not found Sent when the client asks for a file that doesn't exist.

The supplied version of `http_error` can handle all of the above error codes, plus 500 ("Internal server error"). If you want to add more response codes, you will need to modify `http_error`.

4.7 Serving Up Files

Once you have received, parsed, and validated a request, you need to send the requested file back to the client. That's a fairly simple operation:

1. Open the file with `open`(2)[2] (and, as a side effect, verify that it exists).
2. Send an HTTP response header (see Sections 4.3 and 4.8). If you're feeling friendly toward the client, this header could include a `Content-Length:` parameter; you can determine the length of a file with `stat`(2) or `fstat`(2).

[2]For this lab, don't use `fopen`(3) or `fdopen`(3) for this purpose—doing so is likely to lead you astray and may even keep your server from working correctly.

3. Copy the file by *repeatedly* reading one block (4096 bytes) at a time and writing it to the client.[3] Your loop should run until the `read` call returns 0. *Do not* attempt to read the entire file before writing it; doing so slows your server and wastes memory if you do it right, or causes errors if you don't.

4.8 File Types

As you know, real Web servers can send files of many types, and real clients (browsers) can handle most or all of those types. The scheme for identifying file types is too complicated for this lab. Instead, we will use an extremely simple rule: any filename that ends in "`.html`" will be considered to be in HTML format ("`text/html`"),[4] and all other files will be ordinary text ("`text/plain`"). Your server should set the `Content-Type` appropriately.

There are a few extra-credit points available for supporting other file types; see Section 6.

4.9 Testing Notes

Web browsers can be remarkably opaque about what went wrong when a Web server misbehaves. For that reason, we suggest that you do your initial testing with `telnet`, as discussed in Section 7. After you have your server working, you can try it with a browser.

It can be very useful to run your server under `gdb` so that you can step through the code and see what is happening; this approach is especially helpful with `process_request`. However, `gdb` isn't entirely friendly with threads, so if you plan to use `gdb` it's best to start with a non-threaded server. It will be easy to switch to a threaded version later.

4.10 Details About Functions

This section describes every supplied function and what (if anything) you'll need to do to make it work.

We recommend that you begin by studying `write_log` and `http_error` to see how they work and how you will use them. Then it's probably best to attack `parse_uri`, followed by `main`. When you've figured out what you want those functions to do, you can finish `process_request`.

Here are all the functions in `webserver.c`. Functions marked "**" don't need to be changed; functions marked "*" need few or no changes depending on your exact approach.

`main` The main program does a bit of initialization (which you must provide) and then goes into a loop, waiting for connections. When one arrives, it fills in the argument structure (`args`) and invokes `process_request` to handle it. There are two options for calling `process_request`: you can call it directly, or you can invoke it as a thread. The latter makes your server perform better but will require a bit of synchronization code in some other parts of the server (see Section 5).

[3] You need to use `read`(2) and `write`(2) for this purpose, not `fread`(3) and `fwrite`(3).

[4] You are welcome to also recognize "`.htm`" files as being HTML, in true Windows fashion, but it's not required.

****open_listenfd** This is almost identical to the version in the echo server from class. You don't need to modify it.

process_request This function processes a single request from a single client. As supplied, it's set up for threading. If you're doing a non-threaded version, turn the "`#if 1`" into "`#if 0`". Look for "READ THIS" to see how to handle errors and debugging; you will need to add more error handling and debugging output to `process_request` and possibly other functions.

Before `process_request` answers the client, it must log the first line of the request (the GET) to the log file. You need to add that code. You also need to add the code that actually sends the response to the client.

Note that `process_request` works with a Unix file descriptor (`args->connfd`) rather than a stdio-style `FILE *`. That means you can't use functions like `printf`(3) or `fgets`(3). Instead, you are limited to `read`(2) and `write`(2).

****read_request** This function reads a request from the client and returns it as multiple lines, all in a `malloc`ed buffer. The function that calls `read_request` (i.e., `process_request`) *must* free that buffer. You shouldn't need to make any changes to `read_request`.

parse_uri This function parses a GET request, extracts the pathname the client wants to retrieve, and returns that pathname in a `malloc`ed buffer. The caller *must* free that buffer. `parse_uri` does fairly extensive error checking on the format of the request. However, it has one flaw: it assumes that the components of the GET line are each separated by a single space. You need to modify it to handle multiple spaces. Also, the code at the end that allocates and returns the pathname is missing; you must provide that.

copy_to_client Our solution includes a function named `copy_to_client`. You are welcome to write a similar function of your own. If you do, it's up to you what to name it, what its arguments are, and what it does.

***write_log** This function makes it easy to write the log file (which you opened in `main`). It accepts a copy of the argument list from `main`, plus two strings. It writes those strings to the log file, accompanied by useful information like the current time, the client's identity, and the internal ID of the thread that's doing the work.

As provided, `write_log` is a complete implementation for a non-threaded server; you can ignore the NEEDSWORK comment unless you add threading. If you choose a threaded implementation, you will need to think about what changes you need to make to this function.

***http_error** If an HTTP error happens, `http_error` will generate an appropriate response and send it to the client. This function is complete as-is, but you will want to study it because it serves as an example of how to send a client response.

5 Part II (Extra Credit: up to 10 points): Dealing with Concurrent Requests

Real Web servers don't process requests sequentially, because if one client is slow about swallowing data (perhaps because it asked for a really large file) it's not nice to make other clients wait. Instead, they handle multiple requests simultaneously. Once you have a working sequential logging server, you can choose to alter it to be concurrent. The simplest approach is to create a new thread to deal with each new connection that arrives, detach that thread, and have it exit when the connection is terminated.

With this approach, it is possible for multiple peer threads to write to the log file at the same time. Thus, you will need to use a pthreads mutex to synchronize access to the file such that only one peer thread can modify it at a time. If you do not synchronize the threads, the log file might be corrupted, for example by having one line in the file begin in the middle of another.

Note that the skeleton code we provided *does not* contain any thread synchronization. It is your responsibility to identify any shared variables (including shared access to the log file) and protect them appropriately. Remember to watch out for thread-unsafe functions!

5.1 Thread Safety and I/O

It will help you to understand how I/O functions interact with threads. You should assume that any `write`(2) call can cause data to be intermixed with data from another thread. Functions that call `write`(2) indirectly include `fprintf`(3), `fputs`(3), and `fflush`(3), among others.

However, as long as you ensure that only one thread writes at a time, you can safely share a single open file between threads. In particular, it's not necessary (and is quite inefficient) for each thread to open a file, write to it, and close it every time you want to generate a log entry.

6 Part III (Extra Credit: Up to 5 Points): More File Types

As mentioned in Section 4.8, your server only needs to handle HTML files and plain-text files. However, you are welcome to offer a few more file types. In particular, you might wish to support `image/jpeg` and `image/gif`. Again, it's sufficient to detect those based on their suffixes (note that JPEG files can have a suffix of either `.jpg` or `.jpeg`).

7 Hints

- For debugging, you can run your server on your own machine (which is always named `localhost`) or on Wilkes. However, if you run it on Wilkes then you must either test by running the test programs (`telnet`(1), `curl`(1), or `wget`(1)) on Wilkes, or use "ssh port forwarding" to make it available on your local machine. To do that, add the following switch to the `ssh`(1) command line:

  `-L` *port*`:localhost:`*port*

where *port* is the port you have chosen for your Web server. You can then use "`localhost`" wherever you would otherwise write `wilkes.cs.hmc.edu`. (Note that you must keep the `ssh` connection open as long as are still testing.)

- Initially, you should debug your server using `telnet` as the client (see the textbook, Section 11.5.3). Try "`telnet wilkes.cs.hmc.edu` *port*" where *port* is the port your sever is running on. You can get away without using any headers; simply type the GET request. Note that you will have to hit Enter twice before your server will respond. If your server is written correctly, `telnet` will show you the headers and the page, and then terminate.

  An advantage of testing with telnet is that you can see the response code and response headers; this will help you be sure your server is producing output in the right format. Be sure to test both normal and error paths.

- After your server is more robust, give it a try with `wget`(1) or `curl`(1), both of which are command-line programs that will fetch one Web page at a time. **Warning:** by default `wget` will write its output to a file that matches the name of the file you are fetching, which means that you should not run it in the same directory your Web server is running in! A good alternative is to run `wget` as follows:

  `wget -O - -q http://wilkes.cs.hmc.edu:`*port*`/index.html`

  to get it to write to standard output (`curl` writes to standard output by default, so it doesn't have the same problem).[5] Again, if you are running `wget` or `curl` on Wilkes, you can substitute `localhost` for `wilkes.cs.hmc.edu`.

- Later, test your server with a real browser. For example, you can browse to:

  `http://wilkes.cs.hmc.edu:`*port*`/index.html`

  to see the sample HTML file.

- Make sure your server can serve multiple files (i.e., more than one request) without crashing!

- Be careful about memory and file-descriptor leaks. When the processing for an HTTP request fails for any reason, the thread must close all open socket descriptors and free all memory resources before terminating.

- Again, stick to low-level Unix I/O functions (`read`(2) and `write`(2)) for dealing with the network sockets. Do not try to use the Unix "standard I/O" (`stdio`) functions such as `fgets`(3), `fputs`(3), and `fprintf`(3).

- Reads and writes can fail for a variety of reasons. The most common read failure is an `errno = ECONNRESET` error caused by reading from a connection that has already been closed by the peer on the other end, typically an overloaded server. The most common write failure is an `errno = EPIPE` error caused by writing to a connection that has been closed by its peer on the other end. This can occur, for example, when a user hits their browser's Stop button during a long transfer. You must check for failures (a return of -1) and handle them appropriately.

[5] See the manual pages `wget`(1) and `curl`(1) for full, exhaustive, and overwhelming documentation.

- The first time you write to a connection that has been closed by the peer, you will get an error with `errno` set to `EPIPE`. Writing to such a connection a second time elicits a `SIGPIPE` signal, whose default action is to terminate the process. For that reason, the supplied `main` program uses `signal` to ignore `SIGPIPE`.