Programming Assignment 6

Due Monday, November 23 before midnight.

NAME

`proxy` - An http proxy server.

USAGE

`proxy <port>`

DESCRIPTION

`proxy` is a simple multithreaded web proxy server. A web proxy server is a program that allows clients (web browsers) to make indirect network connections to other network servers (e.g. websites). When a user configures their web browser to use a proxy, all of their http requests are sent to the proxy server. The proxy server receives these requests, connects to the appropriate web servers to fill the requests and then passes the results back to the client.

Proxy servers can perform useful functions, such as content reformatting, caching, censorship, circumventing censorship, or anonymization. The `proxy` web server you are to write is not required to do any of these things, except to the extent that any proxy server provides a degree of anonymization.

When executed, `proxy` should open the port passed as a command line argument to accept incoming TCP connections. When a new connection is accepted, `proxy` should create a new detached thread to handle the connection (from here on known as the client connection). The new thread should:

- Read a single HTTP request from the client connection.
- Determine the target host (web server) by examining the Host: header of the request.
- Create a new TCP connection to port 80 of the target host and pass the request to the target host.
- Copy the response from the target host back to the client connection.
- Close both target host and client connections.

The `proxy` server should never exit.

You will need to create a makefile because at the very minimum you'll need to use the `-lpthread` option to compile. Make sure your program compiles with the command `make proxy`.

Your proxy server should never exit or crash. It must survive any input, no matter how badly formatted.

PROTOCOL DETAILS

To write a fully standards-compliant proxy server is beyond the scope of this assignment. However, certain details need to be handled for a functional proxy:

- If the HTTP request is missing the `Host:` header, `proxy` should respond to the client with a properly formatted `400 Bad Request` HTTP response. You may detect and respond to other request formatting problems if you like.
- If `proxy` cannot connect to the target host (because it doesn't exist or isn't accepting connections), it should respond to the client with a properly formatted `404 Not Found` HTTP response.
- The `Connection: keep-alive` header specifies that a TCP connection should be kept alive for multiple HTTP requests. Because `proxy` only processes one HTTP request per connection, this can lead to stalls in fetching webpages where either the client or server waits in vain for data. Fix this by changing all `Connection:` headers to `Connection: close`. Or, better still, handle the `keep-alive` request properly by processing more than one HTTP request per TCP connection.

At the end of this document, you have been assigned a range of ports to use on `turing`. To avoid conflicts, please use only your assigned ports for testing.

EXTENSIONS FOR EXTRA CREDIT

This basic server could be improved and extended in many ways. Here are some suggestions that you might implement for extra credit.

Protocol improvements

- A `POST` request is followed by a message body, which `proxy` is allowed to ignore. This means `POST` methods won't work, and so this proxy server isn't so useful for, say, filling out forms. Detect `POST` methods and properly pass along the message body to the target host.
- It is possible to put a web server at a port other than port 80. In the hostname, this looks like `www.somewhere.edu:8080` to specify port 8080 instead. Detect this situation and connect to the appropriate port.

Functionality extensions

These improvements should be off by default, activated by a command line switch.

- Let's go censorship! Only allow connections to servers matching some criterion, like ending with `.edu`.
- Mess with the web. Translate every page into pig latin. Replace all occurrences of "Rolla" with "Fools". Implement `http://xkcd.com/1288/`

- Make some money. Anytime a host is not found, redirect to an advertiser's website.
- Create a cache. Keep local copies of responses in files, so that you can quickly serve them without having to connect to the target host server.

HINTS

See the `Public/os/bin` folder for a working version. This prints a lot of debugging information to stdout that you might find helpful in understanding how it works. It also allows you to test your browser's proxy settings.

See the course web page for information on how to configure web browsers to use a proxy server.

Read "HTTP Made Really Easy", available on the links part of the OS webpage. Refer to the HTTP 1.0 and HTTP 1.1 specifications if you need more details.

Don't hesitate to copy code from the `iecho/ispeak/ilisten` demo programs from class.

There is a natural way to begin writing this program as two separate programs:

1. The client-side piece should accept connections from clients (web browsers), parse the hostname, and then simply print the request to stdout. This piece can be easily built from `ilisten2`, and is a good test to make sure your web browser is properly configured for the proxy server. As a next step, you could have the client-side piece return a "404 Not Found" response to all client connections.

2. The server-side piece should read data on stdin (a request), open a connection to some web server and pass the input to it. The web server's response should be copied to stdout. This piece can be based on `ispeak`, and is a good test to make sure you can successfully connect to external web servers.

Merging these two programs into one is much easier than trying to do it all at the same time.

The lines of text in the HTTP protocol are terminated with CR/LF, a pair of ASCII characters. Typical Unix text files only end with LF. You can get CR/LF as a string with `"\r\n"`. In particular, a blank line is the string `"\r\n"`.

You'll want to pass the file descriptor for a newly accepted connection to the thread you create to handle that connection. One way to do this is to simply typecast the integer file descriptor to a `(void *)` pointer then cast it back to an `int` in the thread routine. Since a pointer is a number too, this works, but it produces a compiler warning because it's not guaranteed to be portable. A better solution is to call `malloc` (or use C++'s `new`) to get space for one integer, store the file descriptor there, and then pass the pointer to the thread routine. Within the thread routine, cast back to an `(int *)` pointer, copy the file descriptor into a local variable, then free (or delete) the allocated memory. Note that you cannot simply pass a pointer to a local variable in the main accept

loop - that leads to race conditions when two connections are accepted quickly one after the other.

Be aware that when your proxy server is running, anyone on the internet can connect to it and use it. Don't leave it running for long periods of time.

USEFUL MAN PAGES

`strtok_r(3)`	`connect(2)`
`strcmp,strncmp,strcasecmp(3)`	`bind(2)`
`fdopen(3)`	`listen(2)`
`fclose(3)`	`accept(2)`
`fgets(3)`	`close(2)`
`fputs(3)`	`pthread_create(3)`
`getaddrinfo(3)`	`pthread_detach(3)`
`socket(2)`	

PORT ASSIGNMENTS

aalhamad	9100 - 9109	andersonn	9110 - 9119	asiliun2	9120 - 9129
bgovreau	9130 - 9139	caldwell	9140 - 9149	criddell	9150 - 9159
djacob10	9160 - 9169	djoosten	9170 - 9179	eramos	9180 - 9189
gaoj	9190 - 9199	georges	9200 - 9209	gocampo2	9210 - 9219
jcisneros	9220 - 9229	jgovreau	9230 - 9239	kgray20	9240 - 9249
kyu9	9250 - 9259	mmeyer71	9260 - 9269	pokhrelb	9270 - 9279
scurran7	9280 - 9289	sravicha	9290 - 9299	ssantana	9300 - 9309
trelzfm	9310 - 9319	tthai5	9320 - 9329	ywang85	9330 - 9339
zlucas	9340 - 9349				