HTTP Server Application

1 Introduction

You are to design and develop a concurrent TCP server that implements the HTTP protocol in the form of what is commonly called a "web server." This server will accept and process `HEAD` and `GET` transations from a web browser using the `HTTP/1.0` protocol.

Each time your server receives a connection (aka "hit") it shall process one single request and count its type. Those counters are to be used for a server status report.

2 HTTP Protocol Basics

HTTP is a protocol that allows a client application, such as a web browser, to request a data item (refered to as an *entity-body*) from a listening server appication.

This section describes the bare minumum requirements for an HTTP server. The complete details of the HTTP/1.0 protocol are described in RFC 1945 (you can view this and other RFCs at `http://www.ietf.org/`).

2.1 Client Connection and Transaction Request

Your web server will `accept(2)` connections that are initiated from client applications running on machines connected to the internet.

Once a connection is accepted, the client application will transmit one request. Your server support request types that are called `HEAD` and `GET`.

Each request will be made using a seperate TCP connection. This mode of operation is called `Connection: close` mode because the server will close the socket after sending its response data. While there are other modes that we could support, we will use this one because it is the simplest.

Each TCP connection will consist of two phases. Each phase is characterized by the fact that data flows in only one direction.

The first phase is called the *request* phase. This is where the client delivers a `HEAD` or a `GET` request by sending *one or more* lines of text (called *headers*) followed by a blank line. After the client sends the blank line, the request phase is complete. After the request phase, the client will not send any more data.

The second phase is the response phase. This is where the web server delivers a set of headers describing the requested entity–body, a blank line and the requested entity–body and then closes the connection.

NOTE that *all* the text lines that comprise the request phase and all the headers of the response phase are terminated with `\r\n` and not just `\n`. This is because the HTTP protocol specification requires it.

Your server *must* generate these proper line terminators *and* it will *tolerate* clients terminating their request headers with either `\r\n` or just `\n`. This will make it easier to test your server with the `nc(1)` command.

2.1.1 The GET Transaction

In terms of the the world wide web, a `GET` request is the transaction that is performed when a web browser connects to a web server and downloads a web page of some kind.

A typical `GET` request might be the transaction that is used to display my faculty page on the NIU CS department web server as described by the URL `http://faculty.cs.niu.edu/`

Request Phase

When a URL is given to a web browser, it will open a TCP connection to the named machine and port and send a `GET` request. Given the URL:

`http://faculty.cs.niu.edu`

... a web browser will open a TCP socket to port 80 on `faculty.cs.niu.edu` (note that port 80 is a default port for HTTP if none is explicitly specified in a URL) and send the following request:

```
GET / HTTP/1.0
```

To connect to a web server on a port other than 80, you would include the port number in the URL after the hostname. For example, if the above web site were running on port 7212, rather than 80, one would use `http://faculty.cs.niu.edu:7212` In this case, the `GET` request message would be identical, but the connection would be made to port 7212 rather than to port 80.

Note that a client/web browser *may* provide additional information during the request phase. That additional information can be ignored for this assignment. This additional information (when present) will be provided as additional lines of text that you will read and discard. Eventually, the client will send a blank line of text to terminate the request phase.

The following is an example of a Firefox web browser accessing a listening `nc` command with the URL: `http://localhost:12345/`

```
[winans@w520 http]$ nc -l 12345
GET / HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:31.0) Gecko/20100101 Firefox/31.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
```

The `GET` request will always appear on the first line. The requested protocol may or may not be `HTTP/1.0`, however, the HTTP protocol allows the server to downgrade the protocol to 1.0 and you will will take advantage of that fact to simplify your server application.

Note that this request includes a request for a `keep-alive` connection. As you will see below, you can take advantage of downgrading this option to `close` in order to, again, simplify your server application.

Response Phase

The response phase for a `GET` request starts with meta–data/information about the entity–body/data in the form of headers. Continuing with our example of the faculty web server, the above `GET` request returns the following header lines blank line and entity body:

```
HTTP/1.1 200 OK
Date: Sun, 19 Apr 2015 22:37:26 GMT
Server: Apache/2.2.22 (Debian)
Last-Modified: Thu, 30 Jul 2009 22:32:31 GMT
```

```
ETag: "48009-41-46ff3e1179dc0"
Accept-Ranges: bytes
Content-Length: 65
Vary: Accept-Encoding
Connection: close
Content-Type: text/html

<html><body><h1>Welcome to babbage.cs.niu.edu</h1></body></html>
```

In this typical example, we see several more lines than are necessary. The only header lines that are required for proper operation are those described below. We show this complete example here to illustrate additional options and possibilities.

In order to complete this assignment, you need only provide the following header lines in your responses:

```
HTTP/1.0 200 OK
Content-Type: text/plain
Connection: close
```

The first response line is the status line. The possible values for the numeric status field are discussed in the HTTP protocol specification. The status code for a successful completion is 200 as shown here.

The `Content-Type:` header is what tells the browser the type of data that it will receive in the entity–body. In this case, it is an HTML document.

The `Connection` header is used by the server to tell the browser NOT to try to make multiple requests in the same connection. If you leave this line out, your server may periodically appear to get stuck.

Following the header, the *entity-body* is sent and the socket connection is closed.

2.1.2 The HEAD Transaction

The `HEAD` request performs the same operation as the `GET` except the entity–body is not delivered in the response phase. It can be used by web browsers that cache large entity–bodies to detemine if their copy has become out of date.

3 Writing the Server Application

As when writing any program, start with a simple function and get it working perfectly before moving onto more complex operations.

3.1 Phase I

Implement a status reporting "page" that displays some details about your server and test it using `nc(1)` so you can see exactly what is happening. Even a small mistake could prevent a web browser from displaying anything at all from your server thus making it very tough to use for debugging.

Your server statistics report should only be returned when a request is made for `/status`. Your report must provide the following items:

- The current date and time (see `ctime(3)` and `time(2)`).
- The date and time that the server was started.

- An indication of how long your web server has been up and running. This is commonly referred to as the *uptime* of the application.
- The total number of hits.
- The total number of GET requests.
- The total number of HEAD requests.
- The total number of invalid/bad requests.
- The total number of transactions that were aborted due to timeouts.
- The total number of transactions that were aborted due to interrupted writes to the client (aka SIGPIPE).

To test the status report page, use the `nc(1)` command and type in a `GET` request by hand:

```
[winans@w520 http]$ nc localhost 56316
GET /status HTTP/1.0                              <----  request header
                                                  <----  request (blank line)
```

```
HTTP/1.0 200 OK                                   <----  response header
Content-Type: text/plain                          <----  response header
Connection: close                                 <----  response header
                                                  <----  response (blank line)
Current time: Sun Apr 19 16:00:34 2015            <----  response entity-body
Last restart: Sun Apr 19 15:57:48 2015            <----  response entity-body
      Uptime: 0 hours, 2 minuites, 46 seconds     <----  response entity-body
                                                  <----  response entity-body
Socket Peer details:                              <----  response entity-body
  Address: 127.0.0.1                              <----  response entity-body
     Port: 24448                                  <----  response entity-body
                                                  <----  response entity-body
Program Status                                    <----  response entity-body
  Total Hits.....:9                               <----  response entity-body
  GET requests...:8                               <----  response entity-body
  HEAD requests..:0                               <----  response entity-body
  Bad requests...:1                               <----  response entity-body
  Timeouts ......:0                               <----  response entity-body
  Broken pipes...:0                               <----  response entity-body
```

A detailed look at the above response message shows that each of the lines are terminated with `\r\n` can be created by using `nc(1)` and `hexdump(1)` like this:

```
nc localhost 56316 | hexdump -C
```

... and by then entering the sage `GET` request and blank line as shown above.

```
00000000  48 54 54 50 2f 31 2e 30  20 32 30 30 20 4f 4b 0d  |HTTP/1.0 200 OK.|
00000010  0a 43 6f 6e 74 65 6e 74  2d 54 79 70 65 3a 20 74  |.Content-Type: t|
00000020  65 78 74 2f 70 6c 61 69  6e 0d 0a 43 6f 6e 6e 65  |ext/plain..Conne|
00000030  63 74 69 6f 6e 3a 20 63  6c 6f 73 65 0d 0a 0d 0a  |ction: close....|
00000040  43 75 72 72 65 6e 74 20  74 69 6d 65 3a 20 53 75  |Current time: Su|
00000050  6e 20 41 70 72 20 31 39  20 31 36 3a 30 30 3a 33  |n Apr 19 16:00:3|
00000060  34 20 32 30 31 35 0d 0a  4c 61 73 74 20 72 65 73  |4 2015..Last res|
00000070  74 61 72 74 3a 20 53 75  6e 20 41 70 72 20 31 39  |tart: Sun Apr 19|
00000080  20 31 35 3a 35 37 3a 34  38 20 32 30 31 35 0d 0a  | 15:57:48 2015..|
00000090  20 20 20 20 20 20 55 70  74 69 6d 65 3a 20 30 20  |      Uptime: 0 |
000000a0  68 6f 75 72 73 2c 20 32  20 6d 69 6e 75 69 74 65  |hours, 2 minuite|
000000b0  73 2c 20 34 36 20 73 65  63 6f 6e 64 73 0d 0a 0d  |s, 46 seconds...|
000000c0  0a 53 6f 63 6b 65 74 20  50 65 65 72 20 64 65 74  |.Socket Peer det|
000000d0  61 69 6c 73 3a 0d 0a 20  20 41 64 64 72 65 73 73  |ails:..  Address|
000000e0  3a 20 31 32 37 2e 30 2e  30 2e 31 0d 0a 20 20 20  |: 127.0.0.1..   |
000000f0  20 20 50 6f 72 74 3a 20  32 34 34 34 38 0d 0a 0d  |  Port: 24448...|
00000100  0a 50 72 6f 67 72 61 6d  20 53 74 61 74 75 73 0d  |.Program Status.|
00000110  0a 20 20 54 6f 74 61 6c  20 48 69 74 73 2e 2e 2e  |.  Total Hits...|
00000120  2e 2e 3a 39 0d 0a 20 20  47 45 54 20 72 65 71 75  |..:9..  GET requ|
00000130  65 73 74 73 2e 2e 2e 3a  38 0d 0a 20 20 48 45 41  |ests...:8..  HEA|
00000140  44 20 72 65 71 75 65 73  74 73 2e 2e 3a 30 0d 0a  |D requests..:0..|
00000150  20 20 42 61 64 20 72 65  71 75 65 73 74 73 2e 2e  |  Bad requests..|
00000160  2e 3a 31 0d 0a 20 20 54  69 6d 65 6f 75 74 73 20  |.:1..  Timeouts |
00000170  2e 2e 2e 2e 2e 2e 3a 30  0d 0a 20 20 42 72 6f 6b  |......:0..  Brok|
00000180  65 6e 20 70 69 70 65 73  2e 2e 2e 3a 30 0d 0a     |en pipes...:0..|
0000018f
```

Your parent process must count the transactions and their types that are being handled by the child process in your concurrent server. Recall that the child process does not have the ability to directly alter variables in the parent process. To identify the operations performed in your child processes, use different exit status codes that you can inspect from the SIGCHLD handler in your parent process. Note that you have to indicate the request type (`GET` or `HEAD`) as well as an indication of how it was completed (invalid, timeout, `SIGPIPE`).

Get this working perfectly before proceeding to the next phase.

3.2 Phase II

Once the status report is working, enhance your web server to serve web pages from files.

3.2.1 Illegal URLs

Before adding the code to deliver the contents of files, you must verify that your application has not been fooled into sending the wrong files by mistake or in response to a malicous user is trying to hack into your application.

Any URL processed by your web server should be considered illegal if it is **not** composed of a slash '`/`'followed by zero or more of the following characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 0 / ? . _ % & - #
```

Further, requested filename that contains a directory name of `..` shall be considered legal. To check for this case make sure that the request does not include "`/../`" anywhere and that the pathname does not end with "`/..`" You can easily check for these cases by using `strstr(3)`.

Requests that are made to illegal URLs should be handled by sending an error message page as the response. (See the HTTP specification for the meanings of the status codes on the request status line in the response headers. For this project using just 200 and 404 will suffice.)

```
[winans@w520 Desktop]$ nc localhost 33885
GET /../../../a/b/c HTTP/1.0

HTTP/1.0 404 Not Found
Content-Type: text/plain
Connection: close

Error: Query containing .. path element is illegal.
```

3.2.2 Content Types

In order to serve web pages from files, your new web server will have to be able to open the requested files and return their contents. When this operation takes place, your web server **must** report the *type* of the file to the client before delivering it. In your status report, this is simply hardcoded to `text/plain`. To deliver files of varying types, you can use the name of the requested file to determine its type. For example, if a file name ends with `.txt`, you will deliver it with a `Content-type: text/plain` response header. If the requested file name ends with `.gif`, you will deliver it witn `Content-type: image/gif` and so on. The set of types that you are required support with your server are given in the table below:

Ends with	Content-Type:
`.gif`	`image/gif`
`.jpg`	`image/jpeg`
`.jpeg`	`image/jpeg`
`.html`	`text/html`
`.txt`	`text/plain`
(default)	`text/plain`

If the requested file name ends with anything else, use `Content-type: text/plain`.

3.2.3 Delivering Files

In order for your web server to be useful, it has to be able to deliver the contents of files as they are requested (when their names are valid).

In order to do this, use `open(2)` and `read(2)` to read the requested file and `write(3)` to write the data to the client socket.

While `write(2)`ing to the client, it is possible that an error could occur. Specifically, the client could decide to close the TCP connection/socket before you are finished writing the content body! This situation will result in the O/S delivering a broken pipe signal to your process. Be sure to check for and handle these problems and return a proper `exit(3)` status for the parent process to use when counting transaction types and completion statuses.

3.2.4 Debugging Output

In order to grade your assignment, it will be necessary to the details about errors related to illegal requests and file processing on your server. Therefore you **must** provide meaningful messages in the entity–body of any 4xx–series error response (like the illegal "`..`" example above.)

During development include debug `printf(3)` lines that print the file descriptor numbers returned from the `accept(2)` call. This will provide a way to verify that programming errors and timeouts don't result in file descriptors that are left open by mistake. Note: each connection serviced by this web server *should* end up reusing the *same* file descriptor number. If you forget to `close(2)` them after you `fork(2)`, the next one that you `accept(2)` will be a different number each time.

3.2.5 Handling Exceptional Situations

You have to deal with errors in your child processes and communication problems.

Most of the time a communication problem results in your process becomming blocked and waiting indefinitely. The simplest way to address this is to register a `SIGALRM` handler and use `alarm(2)` to create a timeout in your child process after your parent calls `fork(2)`. A suitable timeout value for this project is 5 seconds. You might want to raise the timout value during debugging if you type slowly.

Similarly, you will also require a `SIGPIPE` handler.

You can easily test your alarm timeout with a `nc(1)` connection that never sends a complete request.

You can test the `SIGPIPE` by adding a `sleep(2)` to your child process after it reads the request–phase headers and before it delivers a response so that it takes a few seconds to finish. This will give you enough time to do a test using `nc(1)` where you enter a command and a blank line and then press `^C` to kill `nc(1)` before your server has a chance to start writing. (Implement your server so that an optional `-s` command–line argument can be used to turn on this syntnetic delay so that it can be used during the grading process.)

3.3 Files I Give You

Implement your web server using the skeleton files that are available on my faculty web page.

This file is what is known as a `tar(1)` archive. It contains a number of files that are packaged into a single archive file similar to a `.zip` file. In order to make use of it, you will need to open it up somewhere under your home directory on tiger. Opening this tar archive will make a directory named `http` and fill it with the files representing the skeleton web site and the Makefile that will be used to grade your project.

You can open the compressed tar file like this:

```
tar zxvf http.tar.gz
```

3.4 How to Hand In Your Solution

You will turn in your assignment in person by scheduling a time to meet with your professor during the last week of lectures. Be prepared to discuss the details of your solution and answer questions about it for 30 minutes.