Homework 4

Due: Thu, 12 Mar 2015 23:00:00 -0700

For homework #4, you will build on your homework #3 solution to implement a multithreaded Web server front-end to your query processor. In Part A, you will read through some of our code to learn about the infrastructure we have built for you. In Part B, you will complete some of our classes and routines to finish the implementation of a simple Web server. In Part C, you will fix some security problems in our Web server.

As before, pease read through this entire document before beginning the assignment, and please start early!

In HW4, as with HWs 2 and 3, you don't need to worry about propagating errors back to callers in all situations. You will use `Verify333()`'s to spot some kinds of errors and cause your program to crash out. However, no matter what a client does, your web server must handle that; only internal issues (such as out of memory) should cause your web server to crash out.

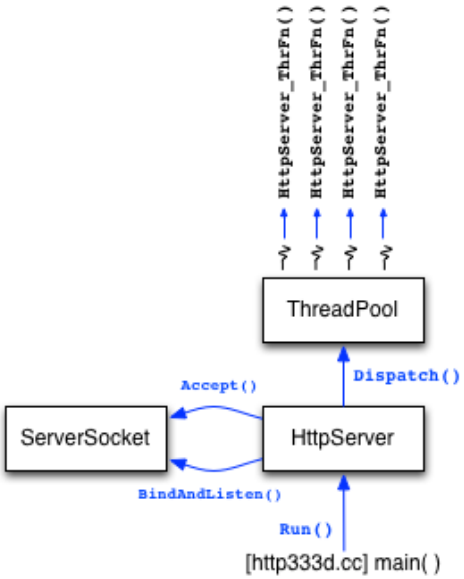
To help you schedule your time, here's a suggested order for the parts of this assignment. We're not going to enforce a schedule; it's up to you to manage your time.

- Read over the project specifications and understand which code is responsible for what.
- Finish `ServerSocket.cc`. Make sure to cover all functionality, not just what is in the unit tests.
- Implement `FileReader.cc`, which should be very easy, and `GetNextRequest` in `HttpConnection.cc`.
- Complete `ParseRequest` in `HttpConnection.cc`. This can be tricky as it involves both Boost and string parsing.
- Finish the code for `http333d.cc`. Implement `HttpServer_ThrFn` in `HttpServer.cc`.
- Complete `ProcessFileRequest` and `ProcessQueryRequest` in `HttpServer.cc`. At this point, you should be able to search the `333gle` site and view the webpages available under `/static/` (e.g., http://localhost:5555/static/bikeapalooza_2011/index.html (http://localhost:5555/static/bikeapalooza_2011/index.html)).
- Fix the security issues with the website, if you have any.

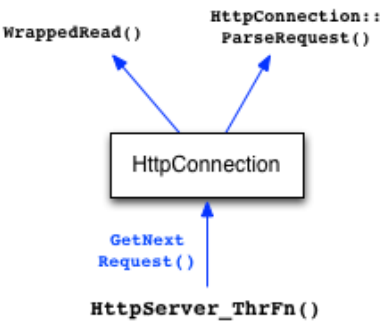
- Make sure everything works as it is supposed to.

Part A: read through our code

Our web server is a fairly straightforward multithreaded application. Every time a client connects to the server, the server dispatches a thread to handle all interactions with that client. Threads do not interact with each other at all, which greatly simplifies the design of the server.



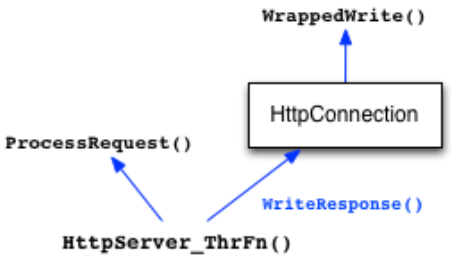
The figure to the right shows the high-level architecture of the server. There is a main class called `HttpServer` that uses a `ServerSocket` class to create a listening socket, and then sits in a loop waiting to accept new connections from clients. For each new connection that the `HttpServer` receives, it dispatches a thread from a `ThreadPool` class to handle the connection. The dispatched thread springs to life in a function called `HttpServer_ThrFn` within the `HttpServer.cc` file.



The `HttpServer_ThrFn` function handles reading requests from one client. For each request that the client sends, the `HttpServer_ThrFn` invokes `GetNextRequest` on an `HttpConnection` object to read in the next request and parse it.

To read a request, the `GetNextRequest` method invokes `WrappedRead()` some number of times until it spots the end of the request. To parse a request, the method invokes the `ParseRequest` method (also within `HttpConnection`). At this point, the `HttpServer_ThrFun` has a fully parsed `HttpRequest` object (defined in `HttpRequest.h`).

The next job of `HttpServer_ThrFn` is to process the request. To do this, it invokes the `ProcessRequest()` function, which looks at the request URI to determine if this is a request for a static file, or if it is a request associated with the search functionality. Depending on what it discovers, it either invokes `ProcessFileRequest()` or `ProcessSearchRequest()`.



Once those functions return an `HttpResponse`, the `HttpServer_ThrFn` invokes the `WriteResponse` method on the `HttpConnection` object to write the response back to the client.

Our web server isn't too complicated, but there is a fair amount of plumbing to get set up. In this part of the assignment, we want you to read through a bunch of lower-level code that we've provided for you. You need to understand how this code works to finish our web server implementation, but we won't have you modify this plumbing.

What to do

- Change to the directory that has your hw1, hw2, hw3, and projdocs directories in it. Click (or right-click if needed) on this hw4.tar.gz (hw4.tar.gz) link to download the archive containing the starter code for hw4. Extract its contents (`tar xzf hw4.tar.gz`). You will need the hw1, hw2, and hw3 directories in the same folder as your new hw4 folder since hw4 links to files in those previous directories. Also, as with previous parts of the project, you can use the solution binary versions of the previous parts of the project if you wish.

- Run `make` to compile the HW4 binaries. One of them is the usual unit test binary called `test_suite`. Run it, and you'll see the unit tests fail, crash out, and you won't yet earn the automated grading points tallied by the test suite. The second binary is the web server itself (`http333d`). Try running it to see its command line arguments. When you're ready to run it for real, you can use a command like:

  ```
  ./http333d 5555 ../projdocs ../hw3/unit_test_indices/*
  ```

  You might need to pick a different port than 5555 if someone else is using that port on the same machine as you.

  Try using our `solution_binaries` server, and running it using a similar command line:

  ```
  ./solution_binaries/http333d 5555 ../projdocs ../hw3/unit_test_indices/*
  ```

Next, launch Firefox or Chrome on that machine, visit http://localhost:5555/ (http://localhost:5555/), and try issuing some searches. As well, visit http://localhost:5555/static/bikeapalooza_2011/Bikeapalooza.html (http://localhost:5555/static/bikeapalooza_2011/Bikeapalooza.html) and click around. This is what your finished web server will be capable of.

To shut down an `http333d` server when you are done with it, open another terminal window on the same machine and run the command

```
kill pid
```

where `pid` is the server process number. Use the `ps` command to find that number.

- Read through `ThreadPool.h` and `ThreadPool.cc`. You don't need to implement anything in either, but several pieces of the project rely on this code. The header file is well-documented, so it ought to be clear how it's used. (There's also a unit test file that you can peek at.)
- Read through `HttpUtils.h` and `HttpUtils.cc`. This class defines a number of utility functions that the rest of HW4 uses. Make sure that you understand what each of them does, and why.
- Finally, read through `HttpRequest.h` and `HttpResponse.h`. These files define the `HttpRequest` and `HttpResponse` classes, which represent a parsed HTTP request and response, respectively.

Part B: get the basic web server working

You are now going to finish a basic implementation of the `http333d` web server. We'll have you implement some of the event handling routines at different layers of abstraction in the web server, culminating with generating HTTP and HTML to send to the client.

What to do

- Take a look at `ServerSocket.h`. This file contains a helpful class for creating a server-side listening socket, and accepting a new connection from a client. We've provided you with the class declaration in ServerSocket.h but no implementation in `ServerSocket.cc`; your next job is to build it.
- You'll need to make the code handle either IPv4 or IPv6 addresses. Run the `test_suite` to see if you make it past the server socket unit tests.

- Read through `FileReader.h` and `FileReader.cc`. Note that the implementation of `FileReader.cc` is missing; go ahead and implement it. See if you make it past the filereader unit test code.
- Read through `HttpConnection.h` and `HttpConnection.cc`. The two major functions in `HttpConnection.cc` have their implementations missing, but have generous comments for you to follow. Implement the missing functions, and see if you make it past the httpconnection unit test code.
- Now comes the hardest part of the assignment. Read through `HttpServer.cc`, `HttpServer.h`, and `http333d.cc`. Note that some parts of `HttpServer.cc` and `http333d.cc` are missing. Go ahead and implement those missing functions. Once you have them working, test your `http333d` binary to see if it works. Make sure you exercise both the web search functionality as well as the static file serving functionality. You'll probably need to look at the source of pages that our solution binary serves and emulate that HTML to get the same "look and feel" to your server as ours.

At this point, your web server should run correctly, and everything should compile with no warnings. Try running your web server and connecting to it from a browser. Also try running the `test_suite` under valgrind to make sure there are no memory issues. Finally, launch the web server under valgrind to make sure there are no issues or leaks; after the web server has launched, exercise it by issuing a few queries, then kill the web server. (The supplied code does have some leaks, but your code should not make things significantly worse.)

Part C: fix security vulnerabilities

Now that the basic web server works, you will discover that your web server (probably) has two security vulnerabilities. We are going to point these out to you, and you will repair them.

What to do

We'll bet that your implementation has two security flaws.

- The first is called a "cross-site scripting" flaw. See this for background if you're curious:

  http://en.wikipedia.org/wiki/Cross-site_scripting (http://en.wikipedia.org/wiki/Cross-site_scripting)

  Try typing the following query into our example web server, and into your web server, and compare the two. (Note: do this with Firefox or Safari; it turns out that Chrome will attempt

to help out web servers by preventing this attack from the client-side!)

```
hello <script>alert("Boo!");</script>
```

To fix this flaw, you need "escape" untrusted input from the client before you relay it to output. We've provided you with an escape function in `HttpUtils`.

- Try telnet'ing to your web server, and manually typing in a request for the following URL. (Browsers are smart enough to help defend against this attack, so you can't just type it into the URL bar, but nothing prevents attackers from directly connecting to your server with a program of their own!)

  ```
  /static/../hw4/http333d.cc
  ```

  This is called a directory traversal attack. Instead of trusting the file pathname provided by a client, you need to normalize the path and verify that it names a file within your `test_tree/` subdirectory; if the file names something outside of that subdirectory, you should return an error message instead of the file contents. We've provided you with a function in `HttpUtils` to help you test to see if a path is safe or not.

Fix these two security flaws, assuming they do in fact exist in your server. As a point of reference, in `solution_binaries/`, we've provided a version of our web server that has both of these flaws in place (`http333d_withflaws`). Feel free to try it out, but DO NOT leave this server running, as it will potentially expose all of your files to anybody that connects to it.

Congrats, you're done with the HW4 project sequence!!

Bonus

There are two bonus tasks for this assignment. As before, you can do them, or not; if you don't, there will be no negative impact on your grade. You should not attempt either bonus task unless and until the basic assignment is working properly. We will not award any bonus credit if the basic assignment is not substantially correct.

- The first bonus task is to perform a performance analysis of your web server implementation, determining what throughput your server can handle (measured both in requests per second and bytes per second), what latency clients experience (measure in seconds per request), and what the performance bottleneck is. You might want to look at the `httperf` tool for Linux to generate synthetic load.

  You should conduct this performance analysis for a few different usage scenarios; e.g., you

could vary the size of the web page you request, and see its impact on the number of pages per second your server can deliver. If you choose to do the bonus, please include a PDF file in your submission containing relevant performance graphs and analysis.

- The second bonus task is to figure out some interesting feature to add to your web server, and implement it! As one idea, find the implementation of a "chat bot", such as Eliza, and add it to your web server. As another idea, implement logging functionality; every time your server serves content, write out some record with a timestamp to a log file; make the log file available through the web server itself. As a third idea, change the results page to show excerpts from matching documents, similar to how Google shows excerpts from matching pages; specifically, make it so that each result in the result list shows:

  ```
  x words + <bold>hit word</bold> + y words
  ```

  for one or more of the query words that hit. If you do this part of the assignment, please include a `readme_bonus` file in your submission describing what you've added.

This part of the assignment is deliberately open-ended, with much less structure than earlier parts. The (small) amount of extra credit granted will depend on how interesting your extension is and how well it is implemented.

What to turn in

When you're ready to turn in your assignment, do the following:

In the hw4 directory:

```
$ make clean
$ cd ..
$ tar czf hw4_<username>.tar.gz hw4
$ # make sure the tar file has no compiler output files in it, but
$ # does have all your source and other files you intend to submit
$ tar tzf hw4_<username>.tar.gz
```

Turn in `hw4_<username>.tar.gz` using the course dropbox linked on the main course webpage.

Grading

We will be basing your grade on several elements:

- The degree to which your code passes the unit tests. If your code fails a test, we won't attempt to understand why: we're planning on just including the number of points that the test drivers print out.
- We have some additional unit tests that test a few additional cases that aren't in the supplied test drivers. We'll be checking to see if your code passes these.
- The quality of your code. We'll be judging this on several qualitative aspects, including whether you've sufficiently factored your code and whether there is any redundancy in your code that could be eliminated.
- The readability of your code. For this assignment, we don't have formal coding style guidelines that you must follow; instead, attempt to mimic the style of code that we've provided you. Aspects you should mimic are conventions you see for capitalization and naming of variables, functions, and arguments, the use of comments to document aspects of the code, and how code is indented.
