Lab Handout 6: `ThreadPool` and Networking

Before starting, go ahead and clone the **`lab6`** folder, which contains the code for **`file-server`**, discussed in Problem 3.

```
myth52$ git clone /usr/class/cs110/repos/lab6/shared lab6
myth52$ cd lab6
myth52$ make
```

Problem 1: `ThreadPool` Thought Questions

- Presented below are partial implementations of my own **`ThreadPool::wait`** and my **`ThreadPool::worker`** method.

  ```
  void ThreadPool::wait() {
    lock_guard<mutex> lg(m);
    done.wait(m, [this]{ return <condition>; });
  }

  void ThreadPool::worker(size_t workerID) {
    while (true) {
      // code here waits for a thunk to be supplied and then executes it
      lock_guard<mutex> lg(m);
      <update to variables>
      if (<update made condition true>) done.notify_all();
    }
  }
  ```

  - Briefly describe a simple **`ThreadPool`** test program that would have deadlocked had **`ThreadPool::worker`** called **`done.notify_one`** instead of **`done.notify_all`**.
  - Assume **`ThreadPool::worker`** gets through the call to **`done.notify_all`** and gets swapped off the processor immediately after the **`lock_guard`** is destroyed. Briefly describe a situation where a thread that called **`ThreadPool::wait`** still won't advance past the **`done.wait`** call.
  - Had **`done.notify_all`** been called unconditionally, the **`ThreadPool`** would have still worked correctly. Why is this true, and why is the **`if`** test still the right thing to include?

- Consider the two server implementations below, where the sequential **`handleRequest`** function always takes exactly 1.500 seconds to execute. The two servers would respond very differently if 1000 clients were to connect –one per 1.000 seconds–over a 1000 second window. What would the 500th client experience when it tried to connect to the first server? What would the 500th client experience when it tried to connect to the second?

  ```
  // Server Implementation 1
  int main(int argc, char *argv[]) {
     int server = createServerSocket(12345); // sets the backlog to 128
     while (true) {
        int client = accept(server, NULL, NULL);
        handleRequest(client);
     }
  }

  // Server Implementation 2
  int main(int argc, char *argv[]) {
     ThreadPool pool(1);
     int server = createServerSocket(12346); // sets the backlog to 128
     while (true) {
  ```

```
        int client = accept(server, NULL, NULL);
        pool.schedule([client] { handleRequest(client); });
    }
}
```

Problem 2: Networking, Client/Server, Request/Response

- Explain the differences between a pipe and a socket.
- Explain how system calls are a form of client/server and request/response.
- Describe how networking is just another form of function call and return. What "function" is being called? What are the parameters? And what's the return value? Assume HTTP is the operative protocol.

Problem 3: File Server [courtesy of Chris Gregg]

In class, we built a server that used our `subprocess` function to run an external program. Using most of the same code (minus the subprocess, timing, caching, and JSON functionality), we can write a *file server* that sends files from our file system to a client, and also produces directory listings if the client requests a directory. Many World Wide Web servers have this ability built-in, and it is a quick way to access files on your web server. If you cloned the repository, you have all of the code for the file server.

In order to request a file, the client requests the server name and port number, followed by the file path, which has its root wherever the server is running. e.g. `http://myth57:13133/filename`

The code for `getFilename` is shown below:

```
/**
 * Function: getFilename
 * ---------------------
 * Retrieves the file name from the GET request.
 */
static string getFilename(iosockstream& ss) {
   string method, path, protocol;
   ss >> method >> path >> protocol;
   string rest;
   getline(ss, rest);
   cout << "\tPath requested: " << path << endl;
   if (path == "/") return "."; // serve current directory
   size_t pos = path.find("/");
   return pos == string::npos ? path : path.substr(pos + 1);
}
```

- The function populates the `method`, `path`, and `protocol` variables, but only uses the `path` variable. What is the purpose of reading in data we won't use?
- What is the purpose of the `getline(ss, rest)` statement?

The three functions shown below determine whether the file requested by the client is a file or a directory, and format the `fileContents` appropriately. If the name requested is a directory, the `DIR *` and `readdir` functions are used to populate an html listing, complete with hyperlinks. For actual files, the file is sent in plain text without any extra formatting.

```
/**
 * Function: listDir
 * -----------------
 * Populates fileContents with a directory listing, with a minimal amount of HTML
 */
static void listDir(string& fileContents, const string fileName, bool& html) {
   DIR *dir;
   struct dirent *ent;
   if ((dir = opendir (fileName.c_str())) != NULL) {
      /* append all files to fileContents, with some html */
      fileContents = "<h1>Files in " + fileName + "</h1><p>\r\n";
      while ((ent = readdir (dir)) != NULL) {
         string d_name = string(ent->d_name);
         // handle . and .. as special cases
         if (d_name == ".") {
            fileContents += d_name + "<br>\r\n"; // don't link
         } else { // make link
            fileContents += "<a href=\"/";
            if (d_name == "..") {
               // remove up to the final slash
               size_t pos = fileName.rfind("/");
               if (pos != string::npos) {
                  fileContents += fileName.substr(0,pos) + "\">..";
               } else {
                  fileContents += "\">..";
               }
            } else {
               fileContents += fileName + "/" + d_name + "\">" + d_name;
            }
            fileContents += "</a><br>\r\n";
          }
       }
       closedir (dir);
       html = true;
    } else {
       /* could not open directory */
       fileContents = "<h1>Could not open directory.</h1>\r\n";
       html = true;
    }
}

/**
 * Function: showFile
 * ------------------
 *  Populates fileContents with the file contents
 */
static void showFile(string& fileContents, const string& fileName, bool& html) {
   std::ifstream t(fileName);
   std::stringstream buffer;
   buffer << t.rdbuf();
   fileContents = buffer.str();
   html = false;
}

/**
```

```
 * Function: getFileContents
 * -------------------------
 * If fileName is a file, populate fileContents with
 * the file's data. If fileName is a directory,
 * populate fileContents with a directory listing
 */
static void getFileContents(string &fileContents, const string fileName, bool &html) {
   struct stat s;
   if (stat(fileName.c_str(),&s) == 0) {
      if (s.st_mode & S_IFDIR) {
          //it's a directory
          listDir(fileContents, fileName, html);
      }
      else if( s.st_mode & S_IFREG )
      {
          //it's a file
          showFile(fileContents, fileName, html);
      }
      else
      {
          fileContents = "<h1>Requested name was not a file or directory.</h1>\r\n";
          html = true;
          return;
      }
   } else {
      fileContents = "<h1>File \"" + fileName + "\" was not found.</h1>\r\n";
      html = true;
      return;
   }
}
```

- Explain how the hyperlink is generated for the "`..`" file. Why does it need to be handled as a special case?
- Assuming that the client is using a web browser, some files that the client might request could actually be html files, yet we are displaying them as raw text files. How might you modify `showFile` to set the `html` flag appropriately for html files?

The `sendResponse` function is shown below:

```
static void sendResponse(iosockstream& ss, const string& payload, bool html) {
   ss << "HTTP/1.1 200 OK\r\n";
   if (html) {
      ss << "Content-Type: text/html; charset=UTF-8\r\n";
   } else {
      ss << "Content-Type: text/plain; charset=UTF-8\r\n";
   }
   ss << "Content-Length: " << payload.size() << "\r\n";
   ss << "\r\n";
   ss << payload << flush;
}
```

Notice that we use the `html` bool we populated in the file-handling functions to tell the client's browser what type of data we are sending back. The "Content-Type" string, known as a "media type" or "MIME type" is an important identifier for the Internet. We have already used the `application/javascript` type for the scrabble solver server, but there are many other types. You can find an overwhelming list here.

- Why must we declare the character set along with the content type?
- What is the purpose of the **`flush`** on the last line?

You can try out the server by running it and then requesting files using your web browser (if you aren't on the Stanford campus, you should use the Stanford VPN so you can access the **`myth`** machines with your browser). There are two directories (**`subdir`** and **`anotherdir`**) with files in them to test going up and down directories.