Lab Solution 6: `ThreadPool` and Networking

Students are encouraged to share their ideas in the #labs Slack channel.

Solution 1: `ThreadPool` Thought Questions

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

```
void ThreadPool::wait() {
  lock_guard<mutex> lg(m);
  done.wait(lock, [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`**.
    - *Allocate **`ThreadPool`** of size 1 on main thread, schedule **`[]{ sleep(10); }`** on main thread, create two standalone threads that call **`ThreadPool::wait()`** and call **`join`** on both. Standalone threads descend into **`done.wait()`**, only one notified, other sleeps and never **`join`**s.*
  - 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.
    - ***`ThreadPool::schedule`** is called **before** thread in **`done.wait()`** gets processor, acquires **`mutex`**, and reevaluates condition. **`ThreadPool::schedule`** alters variables that make condition fail, so pathway for condition to fail exists.*
  - 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?
    - *Just because a thread wakes prematurely doesn't mean it'll rise from **`done.wait()`**. It needs to meet the supplied condition, and very often it won't. The **`if`** test identifies situations where the chances the condition will be met are very good, thereby making better use of the CPU.*
- 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?
  - *Recall that the implementation of createServerSocket calls listen, which instructs the kernel to cap the number of outstanding connection requests yet to be accepted at 128. By the time the 500th client attempts to connect to the first server, the queue will be either full or nearly full, so there's a very good chance that the 500th client will be dropped. The second server, however, immediately accepts all incoming connection requests and passes the buck on to the thread pool, where the client connection will wait its turn in the dispatcher queue.*

```
// 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); });
    }
}
```

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

- Explain the differences between a pipe and a socket.
    - *Fundamentally, a pipe is a unidirectional communication channel and a socket is a bidirectional one. Pipes also are only used to communicate within a given system while sockets are used to communicate over IP, almost always between different hosts.*
- Explain how system calls are a form of client/server and request/response.
    - *The system call is providing a clear service to the user program, which is the client of that service. The request protocol is one we've discussed in several prior lectures and lab handouts (populate `%rax` with an opcode, additional registers with arguments required of that system call, and so forth), and the response protocol has also been established (success or failure [with side effects] expressed via single return value in `%rax`).*
- 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.
    - *The client requires some computation to be performed in another context, and in this case that context is provided on another machine as opposed to some other function on the same machine. The function being called is the URL (where the function lives, and which particular service is relevant, e.g. http://cs110.stanford.edu/cgi-bin/gradebook), the parameters are expressed via text passed from client to server, and the return value is expressed via text passed from server to client.*

Solution 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?
    - *We must read in the **`method`** and **`protocol`** information, because we are getting a well-formatted stream of data. We can't simply pick and choose the data we read in, and we must read all of it. So, we read in the **`method`** and **`protocol`** and then discard them.*
- What is the purpose of the **`getline(ss, rest)`** statement?
    - *There is always a blank line after the protocol that we need to read in.*

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?
    - *Because we want to go back to the previous directory, we need to remove the last part of the path. We do a reverse-find for "/" and this tells us where the last slash is. We then use the `string::substr(0,pos)` function to get the string up to the last slash.*
- 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?
    - *We could check if the extension for the file is `.html`, and then set the `html` flag to `true`.*

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?
    - *Because the world doesn't run on ASCII any more, and we want the ability to send characters from any language (including emojis 🤦‍♀️).*
- What is the purpose of the `flush` on the last line?
    - *Streams are often **buffered**, which means that they don't always send the data immediately. If we didn't flush, then there would be a deadlock condition where the data wouldn't get sent and the client would wait until timeout.*

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.