Lab Solution 7: ThreadPools and Networking

Students are encouraged to share their ideas in the #lab7 Slack channel. SCPD students are welcome to reach out to me directly if they have questions that can't be properly addressed without being physically present for a discussion section.

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::worker(size_t workerID) {
 while (true) {
   // code here waits for a thunk to be supplied and then executes it
   lock_guard<mutex> lg(outstandingThunkCountLock);
   outstandingThunkCount--;
   if (outstandingThunkCount == 0) allDone.notify_all();
 }
}void ThreadPool::wait() {
  lock_guard<mutex> lg(outstandingThunkCountLock);
  allDone.wait(outstandingThunkCountLock,
               [this]{ return outstandingThunkCount == 0; });
}

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(outstandingThunkCountLock);
    outstandingThunkCount--;
    if (outstandingThunkCount == 0) allDone.notify_all();
  }
}
```

- Briefly describe a simple `ThreadPool` test program that would have deadlocked had `ThreadPool::worker` called `allDone.notify_one` instead of `allDone.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 `allDone.wait()`, only one notified, other sleeps and never `join`s.*
- Assume `ThreadPool::worker` gets through the call to `allDone.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 `allDone.wait` call.
  - *`ThreadPool::schedule` is called **before** thread in `allDone.wait()` gets processor, acquires `mutex`, and reevaluates condition. `ThreadPool::schedule` increments `outstandingThunkCount`, so pathway for condition to fail exists.*
- Had `allDone.notify_all` been called unconditionally (i.e. not just because `outstandingThunkCount` was zero), 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 `allDone.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 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); });
  }
}// 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?
    - *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.*
- Describe the network architecture needed for:
    - Email servers and clients
        - *Most email clients and servers speak IMAP over a secure connection. IMAP is similar to HTTP, except that the request and response protocol is optimized for the selection of a mailbox, a digest of all emails in that mailbox, the ability to create and delete mailboxes, the ability to mark an email as read, and the ability, of course, to send an email. (Curious how you can securely telnet to, say, `imap.gmail.com`? Read this.)*
    - Peer-to-Peer Text Messaging via cell phone numbers
        - *By default, the cell service provider intercepts all messages via a centralized farm of servers and forwards messages (with images, emoji, etc) on to the intended recipient. In some cases (e.g. two iPhones in conversation over wifi), Apple mediates instead of, say, Verizon. For an accessible introduction to the actual protocol used by early SMS implementations, read this.*
    - Skype
        - *Same principle as SMS/text messaging, except that persistent connections between clients need to be maintained. This Wikipedia segment does a nice job explaining what Skype does, without going in to the weeds. If you like going into weeds, then this is a really, really well written technical piece explaining how it all works. If you take CS144, this last article is a reading assignment.*
- As it turns out, each of the three network applications above all make use of custom protocols. Which ones could have relied on HTTP and/or HTTPS instead of custom protocols?
    - *Even though it might have been cumbersome, all of them could have. HTTP/HTTPS is a fairly generic grammar that allows side effects, and everything needed for email, SMS, and video chat could, in principle, be codified via HTTP. However, custom protocols are generally constructed to optimize for common operations (as with email messages that need to be deleted) and/or the need for persistent connections (as with video conferencing).*

Solution 3: Hello Server Short Answers

Consider the following server called `hello-server`:

```
static void handleRequest(int client) {
  sockbuf sb(client);
  iosockstream ss(&sb);
  ss << ”Hello from server (pid: ” << getpid() << ”)” << endl;
}

static void runServer(int server) {
  cout << ”Firing up server inside process (pid: ” << getpid() << ”).” << endl;
  while (true) {
    int client = accept(server, NULL, NULL);
    cout << ”Request handled by server (pid ” << getpid() << ”).” << endl;
    handleRequest(client);
  }
}

int main(int argc, char *argv[]) {
  int server = createServerSocket(33334);
  for (size_t i = 0; i < kNumWorkers; i++) {
    if ((workerPIDs[i] = fork()) == 0) runServer(server);
  }
  signal(SIGINT, shutdownAllServers);
  runServer(server); // let master process be consumed by same server

  return 0;
}static const size_t kNumWorkers = 3;
static pid_t workerPIDs[kNumWorkers];
static void shutdownAllServers(int unused) {
  for (size_t i = 0; i < kNumWorkers; i++) {
    cout << "Shutting down worker (pid " << workerPIDs[i] << ")." << endl;
    kill(workerPIDs[i], SIGINT);
    waitpid(workerPIDs[i], NULL, 0);
  }
```

```
  cout << "Shutting down orchestrator." << endl;
  exit(0);
}

static void handleRequest(int client) {
  sockbuf sb(client);
  iosockstream ss(&sb);
  ss << "Hello from server (pid: " << getpid() << ")" << endl;
}

static void runServer(int server) {
  cout << "Firing up server inside process (pid: " << getpid() << ")." << endl;
  while (true) {
    int client = accept(server, NULL, NULL);
    cout << "Request handled by server (pid " << getpid() << ")." << endl;
    handleRequest(client);
  }
}

int main(int argc, char *argv[]) {
  int server = createServerSocket(33334);
  for (size_t i = 0; i < kNumWorkers; i++) {
    if ((workerPIDs[i] = fork()) == 0) runServer(server);
  }
  signal(SIGINT, shutdownAllServers);
  runServer(server); // let master process be consumed by same server


  return 0;
}
```

If I launch the above executable on `myth4`, hit it 12 times by repeatedly typing `telnet myth4 33334` at the prompt from a `myth22` shell, and then press ctrl-c on `myth4`, I get the following:

```
myth4> ./hello-server
Firing up hello server inside process with pid 19941.
```

```
Firing up hello server inside process with pid 19940.
Firing up hello server inside process with pid 19942.
Firing up hello server inside process with pid 19943.
Request handled by server (pid 19940).
Request handled by server (pid 19942).
Request handled by server (pid 19941).
Request handled by server (pid 19943).
Request handled by server (pid 19940).
Request handled by server (pid 19942).
Request handled by server (pid 19941).
Request handled by server (pid 19943).
Request handled by server (pid 19940).
Request handled by server (pid 19942).
Request handled by server (pid 19941).
Request handled by server (pid 19943).
^CShutting down server with pid 19941.
Shutting down server with pid 19942.
Shutting down server with pid 19943.
Shutting down master.
```

Based on the code and test run you see above, answer each of the following five short answer questions.

- Note the very first line of the server's `main` function creates a server socket that listens to 33334 for incoming network activity. Why, after the `for` loop within `main`, are all of the child processes also listening to port 33334 through the same server socket descriptor?
    - *Because server sockets are descriptors, they are replicated across `fork` boundaries as traditional descriptors are, and they're all bound to the same session/resource in the file entry table.*
- During lecture, we referred to port numbers as virtual process ids. What did we mean by that, and why are virtual process ids needed with networked applications?
    - We need exposed port number to remain constant and map to a bona fide pid internally, much as virtual addresses map to physical ones. We can control the port number, but we can't control the chosen pid.
- What happens if the call to `exit(0)` is removed from the implementation of `shutdownServers` and we send a `SIGINT` to the master process in the suite of 4 `hello-server` servers?

    - *The three child servers are killed, as they rely on the default `SIGINT` handler, which interrupts (i.e. terminate) the process. Parent process continues on without children.*
- If the call to the `signal` function is moved to reside above the `for` loop in `main` instead of below it, does that impact our ability to close down the full suite of 4 `hello-server` servers?
    - *We can still close all four by sending `SIGINT` to the parent. Now all servers respond to the handler, but we don't check the return values of `kill` or `waitpid`, so if those calls fail or are redundant, the handlers still run without crashing (even if there are more `cout` statements now).*
- The above program doesn't close the server sockets in `shutdownServers`. Describe a simple way to ensure a server socket is properly closed before the surrounding server executable exits.
    - *Store the value of `server` as a global so that handlers can access it. Add `close(server)` just before the `exit` call, and install `SIGINT` handlers in child servers that call `close(server)`, `exit(0)`, and nothing else.*

Solution 4: Web Servers and Python Scripts

It's high time you implement your own multithreaded HTTP web server, making use of a `ThreadPool` to manage the concurrency, and all of the multiprocessing functions (`fork` and all related functions) to launch another executable–in all cases, a Python script–to ingest the majority of the HTTP request from the client and publish the entire HTTP response back to it.

Here are the pertinent details:

- The web server listens to port 80 like all normal web servers do.
- Every single HTTP request is really a request to invoke a python script. For instance, if the first line of a request is `GET /add.py?one=11&two=50 HTTP/1.0`, the web server executes the `{"python", "./scripts/add.py", "GET", "/add.py?one=11&two=50", "HTTP/1.0"}` argument vector.
- All python scripts are housed within the `scripts` directory. `GET /one.py` invokes `./scripts/one.py`, `GET /deep/down/below.py` invokes `./scripts/deep/down/below.py`, and so forth.
- All python scripts expect to read the entire HTTP request (save for the first line) in through standard in, and the entire HTTP response is published via standard out.
- All python scripts accept three arguments, which are the tokens that make up the first line of the HTTP request. The web server itself must read in the first line from the client, tokenize it so it knows what to pass as arguments to the relevant python script, and then assume the python script pulls everything else.
- If the URL identifies a resource whose name doesn't end up `".py"`, then your server can simply ignore the request and close the connection.

- Similarly, if the URL identifies a resource whose name ends in `".py"`, but the python script doesn't exist, your web server can ignore the request and close the connection.
- You may not use the `system` function as you're permitted in your map-reduce assignment (which goes out tonight). You must use `fork`, `execvp`, and other related functions.
- You may assume that all python scripts successfully invoked always succeed, so you needn't check any status codes via `WIFEXITED`, `WEXITSTATUS`, etc.
- Your implementation must close all unused descriptors and cull all zombie processes.
- Your implementation should not make use of any signal handlers.

Your implementation can make use of the following routine, which reads the first line of an HTTP request coming over the provided client socket (up to and including the `"\r\n"`), and surfaces the method (e.g. `"GET"`), the full URL (`"/add.py?one=11&two=50"`), the path (`"/add.py"`), and the protocol (`"HTTP/1.1"`) through the four `string`s shared by reference.

```
static void parseRequest(int client, string& method, string& url, string& path,
string& protocol);
```

You're to complete the implementation of the web server by fleshing out the details of the `handleRequest` function.

```
static void handleRequest(int client) {
  string method, url, path, protocol;
  // everything below represents a reasonable solution that meets all requirements
  parseRequest(client, method, url, path, protocol);
  if (!endsWith(path, ".py")) { close(client); return; }
  pid_t pid = fork();
  if (pid == 0) {
    dup2(client, STDIN_FILENO);
    dup2(client, STDOUT_FILENO);
    close(client);
    string script = "./scripts" + path;
    const char *argv[] = {
      "python", script.c_str(), method.c_str(), url.c_str(), protocol.c_str(), NULL
    };
    execvp(argv[0], const_cast<char **>(argv));
    exit(0);
```

```
  }
  close(client);
  waitpid(pid, NULL, 0);
} int main() {
   int server = createServerSocket(/* port = */ 80);
   runServer(server);
   return 0; // never gets here, but compiler doesn't know that
}


static const kNumThreads = 16;
static void runServer(int server) {
  ThreadPool pool(kNumThreads);
  while (true) {
    int client = accept(server, NULL, NULL);
    pool.schedule([client] { handleRequest(client); });
  }
}

static void handleRequest(int client) {
   string method, url, path, protocol;
   // everything below represents a reasonable solution that meets all
requirements
   parseRequest(client, method, url, path, protocol);
   if (!endsWith(path, ".py")) { close(client); return; }
   pid_t pid = fork();
   if (pid == 0) {
      dup2(client, STDIN_FILENO);
      dup2(client, STDOUT_FILENO);
      close(client);
      string script = "./scripts" + path;
      const char *argv[] = {
         "python", script.c_str(), method.c_str(), url.c_str(),
protocol.c_str(), NULL
```

```
      };
      execvp(argv[0], const_cast<char **>(argv));
      exit(0);
    }
    close(client);
    waitpid(pid, NULL, 0);
}
```