Lab 2: A Concurrent Web Server

Checkpoint due: Monday Sep 28th @ 11:59pm

Lab due: October 5th @ 11:59pm

Let’s look at what we are implementing…

Program Workflow

1. Read the arguments, find your document root, bind to the specified port, and begin listening for incoming connections.
2. Accept a connection, and:
   a. **week 1**: hand the socket off to a function that handles the remaining steps.
   b. **week 2**: pass the socket to a new thread for concurrent processing.
3. Receive and parse a request from the client.

Program Workflow (continued)

4. Look for the path that was requested, starting from your document root (the second argument to your program). One of four things should happen: **You might want to make each of these cases a separate function!**
   a. If the path exists and it's a regular file, formulate a response (with the Content-Type header set) and send it back to the client.
   b. If the path exists and it's a directory that contains an index.html file, respond with that file.
   c. week 2: **BONUS** If the path exists and it's a directory that does *NOT* contain an index.html file, respond with a directory listing.
   d. If the path does not exist, respond with a 404 code with a basic HTML error page. The 404 HTML page can be static and very simple — it just needs to be enough for a user to see a 404 message in a real browser.

Program Workflow (continued)

4. Close the connection and continue serving other clients.

Week 1 Goals

Implement everything except handling multiple client concurrently and BONUS of returning directory listing.

Review of some socket things (from server's side)…

socket()

Server

socket()

bind()

listen()

accept()

recv()

send()

close()

socket()

Client

Initiate a descriptor and OS data structures including receive/send buffers. This is server_sock

connect()

send()

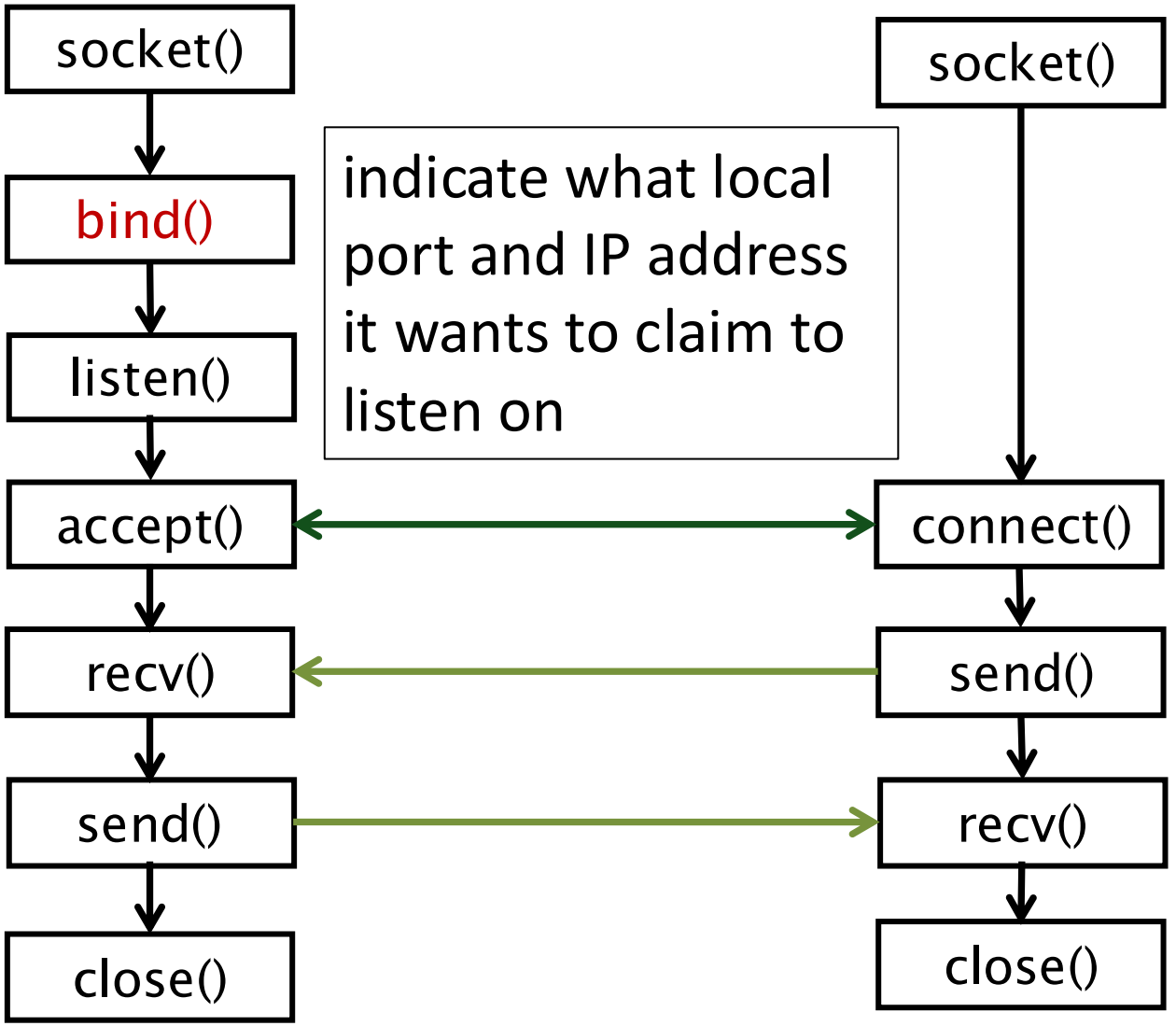
recv()

close()

bind()

Server

Client



Run `man 2 bind` in the terminal…

```
int bind(int sockfd, const struct sockaddr *addr,
                socklen_t addrlen);
```

1. What does bind use about its inputs?
2. What does bind return on success?
3. What does bind return on error?

listen()

Server

socket()

bind()

listen()

start listening for new clients

accept()

recv()

send()

close()

socket()

Client

connect()

send()

recv()

close()

Run `man 2 listen` in the terminal…

```
int listen(int sockfd, int backlog);
```

1. How does listen use its inputs?
2. What does listen return on success?
3. What does listen return on error?

accept()

Server

socket()

bind()

listen()

accept()

recv()

send()

close()

Client

socket()

connect()

send()

recv()

close()

receive TCP connection request from client

On the server side:
It must call and return from accept()

On the client side it will call connect() to initiate a TCP connection

accept()

Server

socket()

bind()

listen()

accept()

recv()

send()

close()

accept() returns a new socket per_client_sock that it can now send and recv on

socket()

Client

onnect()

send()

recv()

close()

Run `man 2 accept` in the terminal…

```
int accept(int sockfd,
           struct sockaddr *_Nullable restrict addr,
           socklen_t *_Nullable restrict addrlen);
```

1. What does accept do with its inputs?
2. What does accept return on success?
3. What does does return on error?

(note: *_Nullable means an argument is allowed to be NULL)

send(), recv()

Server

socket()

bind()

listen()

accept()

recv()

send()

close()

Client

socket()

nnect()

send()

recv()

close()

What is the web server receiving and what is it sending?

send(), recv()

Server

Client

socket()

bind()

listen()

accept()

recv()

send()

close()

socket() returns
server_sock

accept() return
per_client_sock

ocket()

nnect()

send()

recv()

close()

send(), recv()

Server

socket()

bind()

listen()

accept()

recv()

send()

close()

socket() returns server_sock

accept() return per_client_sock

Client

cket()

nnect

send()

recv()

close()

After connecting with a client the server has two sockets: server_sock and per_client_sock. Why do I have two different sockets? Which socket do I call recv() and send() on?

close()

Server

socket()

bind()

listen()

accept()

recv()

send()

close()

socket()

Client

connect()

recv()

close()

Does the server or the client close the connection first?

Let’s look at the starter code….

Some more useful system calls…

- Use `chdir()` to move current working directory to test_documents
- Use `stat()` to check if an input is a file, a path that exists and/or if it is a directory. You can statically allocate a buffer for stat's second argument. You can also use this to get the size of files which may also be helpful
  - See test_stat.c in starter code

You can assume…

- Paths will end in a slash if it is a directory
- Assume file suffices correctly correspond to their type
- Only worry about GET requests
- When user send HTTP/1.1 you should respond with HTTP/1.0
- Size of request from client is not larger than 4 KB

Unlike Lab 1: You CANNOT assume, the size of the file that a client requests.

- You can open a file, read a chunk of it (maybe 4096 bytes) and then send that chunk. Do this in a loop until reaching end of file.
- DO NOT attempt to read entire file into memory and then send. If file was massive this is a very bad use of memory.

Important things to consider…

- Make the code that handles a new client a helper function to make implementing concurrent server in Week 2 easier!
- You need to return status code and Content-Type so browser knows how to render the HTTP response. You can assume the filename after the dot is the type.
  - No other header lines are required
- If user asks for path that ends in / then return index.html file if there is one in directory
  - BONUS: Returning a directory listing if a directory does not contain an index.html file. Note: Last semester no one got to the bonus…
- Return a 404 Not found status code AND a basic HTML file that would show 404 Not Found if the path/file doesn’t exist

Testing…

- Use telnet and testing with a browser to see non-text replies are rendering correct
- Use md5sum to check file sent to client is exact copy of file in test_documents
- To see an example of what your server should do run a python server

```
cd test_documents
python3 -mhttp.server 8080
```

- Always run with valgrind