CSE 29
Lecture 17 Summary

March 3, 2026

🧠 Review Questions & **Answers** (Logistics 🎉)

Q1: What room + time is your week 10 exam?

Lab you are enrolled in

Q2: What days are available for make-up exams?

Mon (12-4pm) Wed (10am-4pm)

Q3: How many make-up sessions can you schedule? **1**

How many exams can you attempt? **Up to 3**

Logistics Q&A

- How many exams in the 1 session?
  - You can take all three exams in the 1 session
  - It is 2 hours, so be sure to budget your time properly
  - It will have 3 sections, exam 1, 2, and 3
  - You do not have to do all 3 sections

HTTP servers!

Parts of a URL (Uniform Resource Locator)

- **https** - the protocol
  - A set of rules when communicating with a computer over networks
- **Domain** - who owns it
  - GitHub owns/controls the computer that sends over information
- **Path** - which specific resource the user wants from the computer

https://ucsd-cse29.github.io/wi26

The `curl` command

- Stands for **cat URL**
- Will make a request and get information from a URL
- Just like going to a URL in your browser, but this is in the terminal

```
joe@rooibos ~> curl https://ucsd-cse29.github.io/wi26/
```

vim site.txt ~

```
1  <!DOCTYPE HTML>
2  <html lang="en" class="sidebar-visible no-js light">
3      <head>
4          <!-- Book generated using mdBook -->
5          <meta charset="UTF-8">
6          <title>Systems Programming &amp;amp; Software Tools - UCSD C
7          <!-- Custom HTML head -->
8          <meta content="text/html; charset=utf-8" http-equiv="Con
9          <meta name="description" content="">
10         <meta name="viewport" content="width=device-width, initi
11         <meta name="theme-color" content="#ffffff" />
12
13         <link rel="icon" href="favicon.svg">
14         <link rel="shortcut icon" href="favicon.png">
15         <link rel="stylesheet" href="css/variables.css">
16         <link rel="stylesheet" href="css/general.css">
17         <link rel="stylesheet" href="css/chrome.css">
18         <link rel="stylesheet" href="css/print.css" media="print
19         <!-- Fonts -->
```

`curl`ing our course website gives you the HTML code that makes up the website!

Q: How does GitHub know to give me this HTML?

```
(base) ~ curl https://ucsd-cse29.github.io/wi26/ -o website.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 51944  100 51944    0     0   252k      0 --:--:-- --:--:-- --:--:--  253k
```

```
(base) ~ cat website.txt | head -n 20
<!DOCTYPE HTML>
<html lang="en" class="sidebar-visible no-js light">
    <head>
        <!-- Book generated using mdBook -->
        <meta charset="UTF-8">
        <title>Systems Programming &amp;amp; Software Tools - UCSD CSE29 Winter 2026</title>
        <!-- Custom HTML head -->
        <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
        <meta name="description" content="">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <meta name="theme-color" content="#ffffff" />

        <link rel="icon" href="favicon.svg">
        <link rel="shortcut icon" href="favicon.png">
        <link rel="stylesheet" href="css/variables.css">
        <link rel="stylesheet" href="css/general.css">
        <link rel="stylesheet" href="css/chrome.css">
        <link rel="stylesheet" href="css/print.css" media="print">
        <!-- Fonts -->
        <link rel="stylesheet" href="FontAwesome/css/font-awesome.css">
```

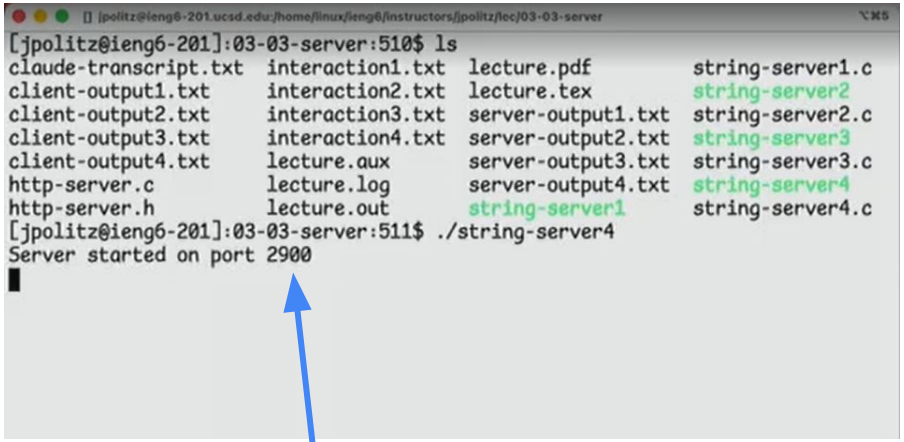
A: GitHub has some code on their server that is running. When Joe sends this request for the website, GitHub's computers returns this HTML.

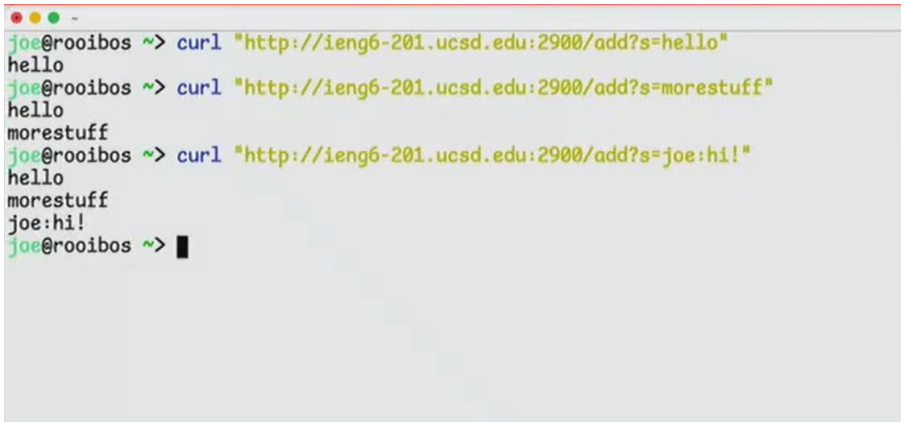
Let's try writing this server-side code!

Server Example

Let's see a server in action!

Lets run `string-server4` (from our handout). This is our final product!



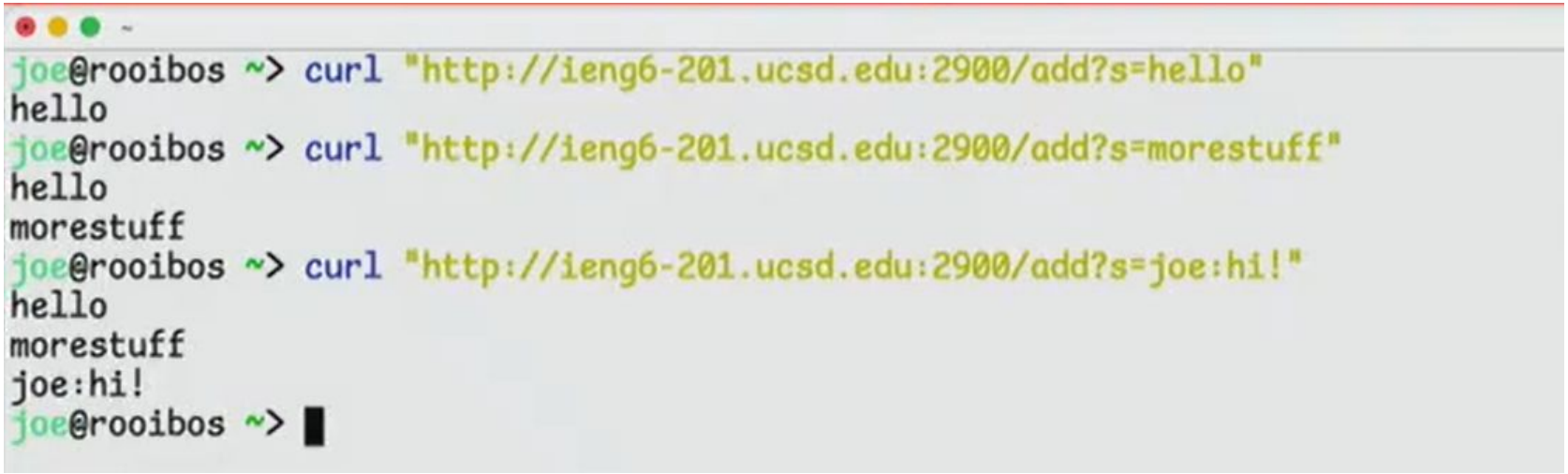


The terminal running our server

A new terminal to access the URL

We get port 2900, and now we can send requests to it (adding words to a list)

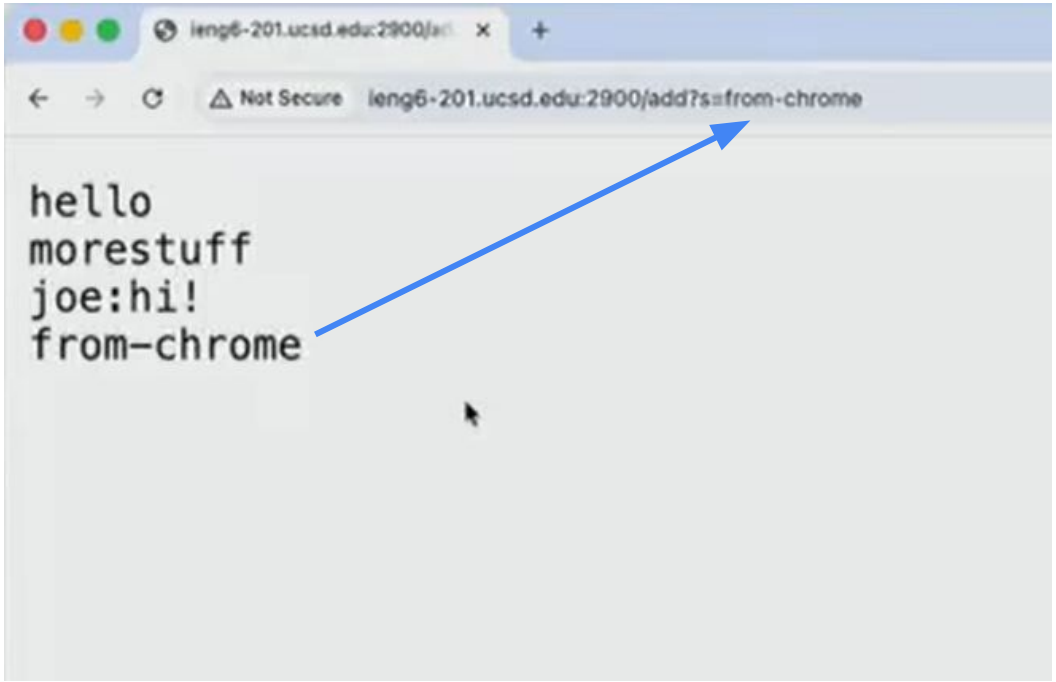
Accessing the Server



Anyone on **UCSD-PROTECTED** can access this, *as long as the server is running*, and receive any string added by anyone calling the server

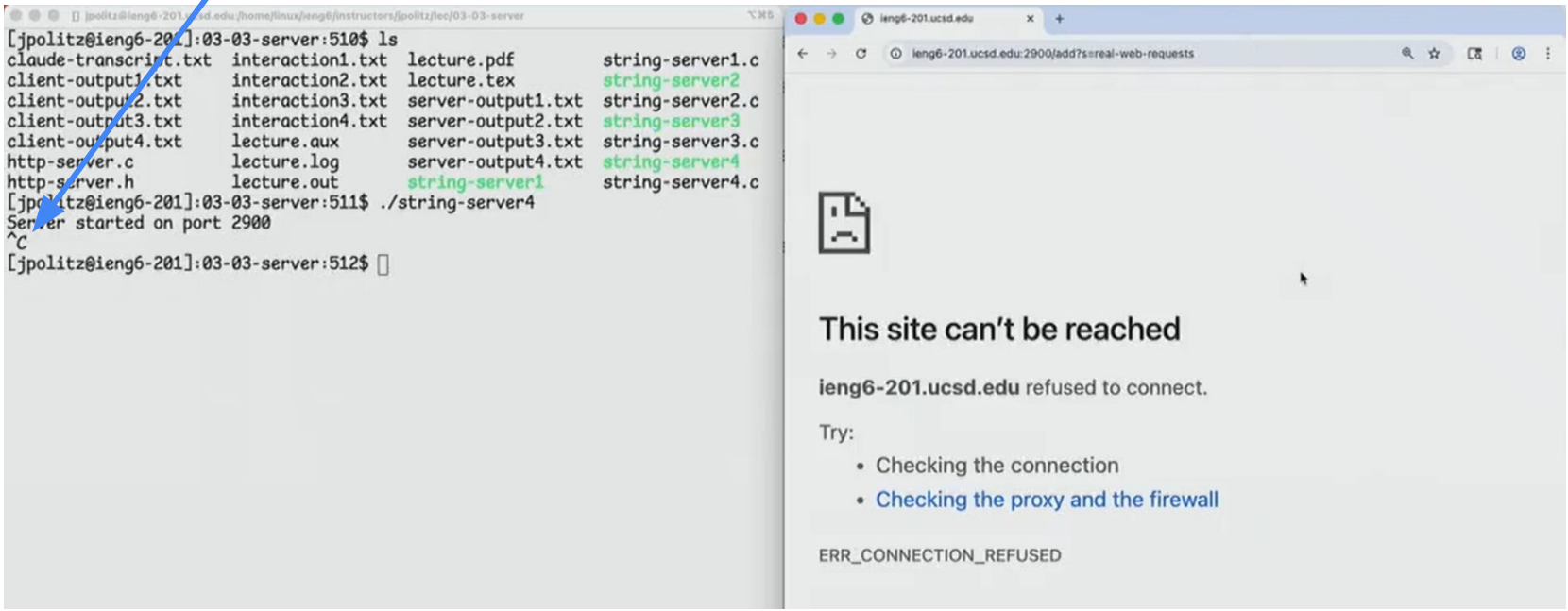
We can see it on the browser as well!

These are normal real web requests happening. This is us on Google Chrome



Shutting Down the Server

We can hit **Ctrl+C** to stop the `string-server4` server from running



The Networking Process

1. Joe types `curl “https://ieng6-201.ucsd.edu:2900/add?helloimjoe”`
2. Joe’s computer sends this request over to the nearest WiFi router
3. The request goes to a central campus router
4. Central campus router checks if the computer is on campus (UCSD-PROTECTED) since Joe is trying to access ieng6
5. Central campus router sends response all the way back to Joe’s computer

- In CSE 29, we will be wiring the code that listens for requests and sends responses back
    - Take CSE 123 and 124 for more networking fun!

Creating a Server

Let’s look at the handout

From `http-server.c/h`

```
void start_server(void(*handler)(char*, int), int port);
```

Starts an HTTP server on the given `port`. Runs forever (until Ctrl-C), calling `handler` once per incoming request.

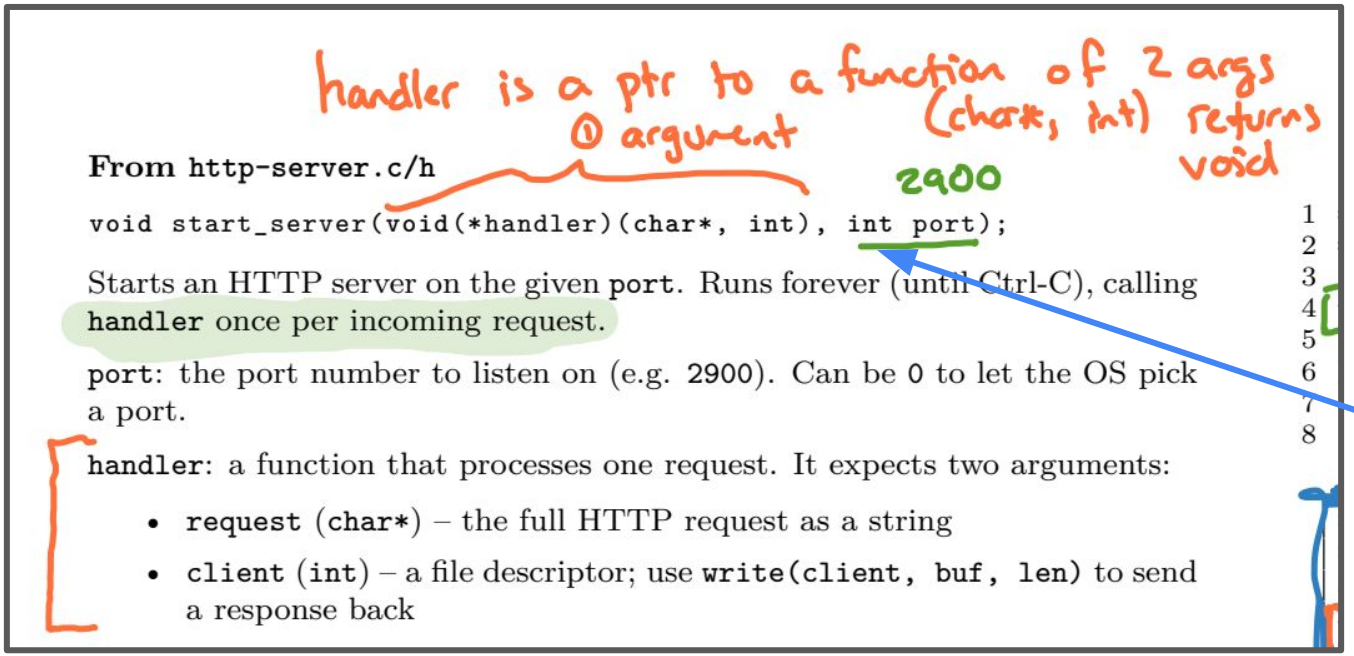
`port`: the port number to listen on (e.g. `2900`). Can be `0` to let the OS pick a port.

`handler`: a function that processes one request. It expects two arguments:

- `request` (`char*`) – the full HTTP request as a string
- `client` (`int`) – a file descriptor; use `write(client, buf, len)` to send a response back

- We have a library called `http-server` with a function `start_server`
- `start_server` starts an HTTP server on the given port (2900 in our case)
  - It listens for requests and sends responses back
- The server runs forever, always listening for new requests, calls a `handler` to send a response back

The `handler` parameter

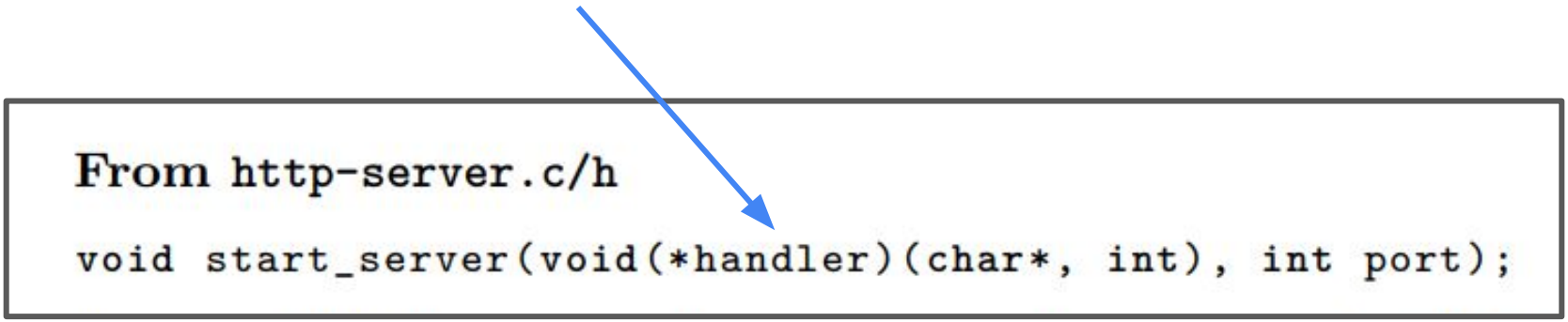


`port` is just a number

That entire thing is the type for the `handler` parameter!

Our Task

- Our **code needs to be called** in response **to an event happening**
  - The event in our case would be a request coming in
  - Our application in the example was adding strings to a list
- This is a common pattern when we are writing code in a system
- We have to pass in a function that does whatever work we want it to do
  - This is why we pass in `handler` function



Lets try it! (This is also on the handout)

string-server1.c

```
#include “http-server.h”

#include <string.h>

void handle(char *request, int client) {

    printf(“Received request: \n%s\n”, request);

}


int main() { start_server(&handle, 2900); }
```

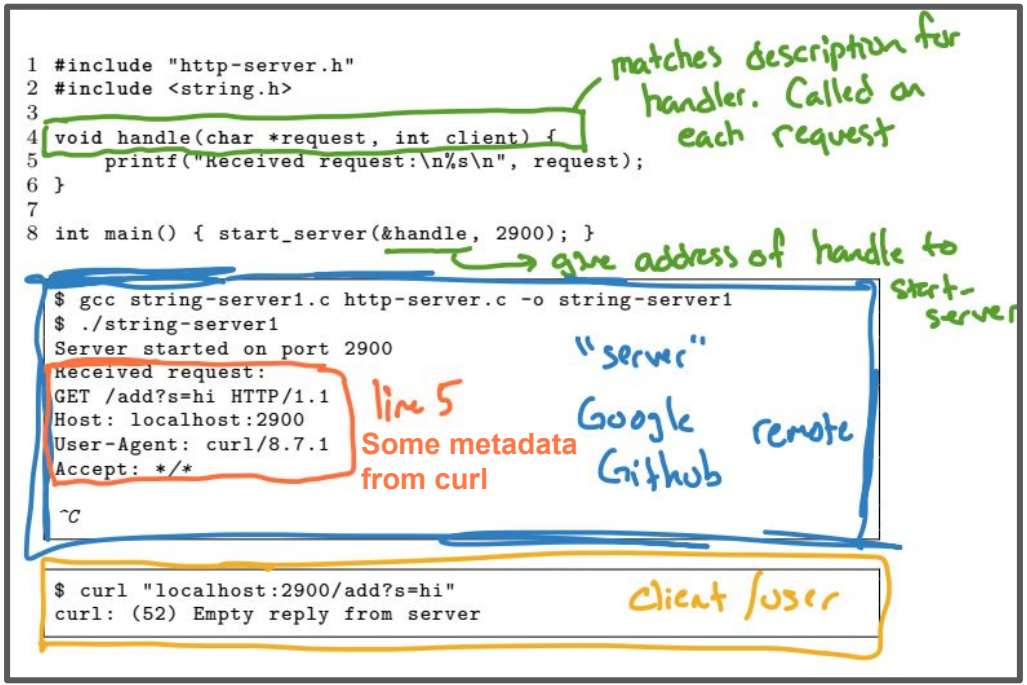
This matches the description for handlers (returns `void`, take a `char*` and an `int`)

Give the address of `handle` to `start_server`

Running our Server

Terminal 1 (The server)

Terminal 2 (The client)



`start_server` has a while loop internally that listens to requests and calls handle once a request comes in

Questions

- Where in our address space is address for the `handle` function?
  - `handle` is in the **Code** region of our address space so it won't go away
- Is there a range the `port` number can be?
  - When we are testing, a port that is [2000, 10000) is good
  - Low number ports (ex. 100) is for system level stuff
    - Port 80 is the default port for HTTP stuff
    - Port 443 is the default port for HTTPS stuff
- What if two people use the same port?
  - The second person would not be able to start their server
  - It would say "Port already in use"
- Is this like virtual memory?
  - It's more like files for the network
  - The OS keeps track of the port number that corresponds to a specific resource

Why does `curl` say “Empty reply from server”?

- The `handler` **only printed to our terminal**!
- The `handler` **sent no reply** to the requester.
- We took no action to send any response (`printf` was local to the server)

```
$ curl "localhost:2900/add?s=hi"
curl: (52) Empty reply from server
```

Printed to terminal, nothing was returned

```
1 #include "http-server.h"
2 #include <string.h>
3
4 void handle(char *request, int client) {
5   printf("Received request:\n%s\n", request);
6 }
7
8 int main() { start_server(&handle, 2900); }
```

How do we send a response to the requester?

Questions

- Where is `handle` running, on the server or the client side?
  - `handle` is only running on the server, not on the client's side
- When is `handle` called?
  - `handle` is called once per incoming request to the server
- How does the client know it established a connection with the server?
  - There's a bit more going on in `start_server` where it receives bytes from `curl`
  - In this previous example, `curl` didn't get a response from the server, so it concluded that the port was closed

HTTP Response

HTTP (Hypertext Transfer Protocol)

- A protocol is a set of rules of specific piece of metadata being sent and received
- HTTPS is a protocol for the World Wide Web, designed for client-server data communication
- Browsers and curl all use HTTP
- When a server sends a response with HTTP, it is in this format ⬇️

HTTP Response Format

When the server receives a request, it must `write` back a *response*, which has a specific format.

```
HTTP/1.1 200 OK
Content-Type: text/plain

...response body...
```

Breaking Down The Response

HTTP Response Format

When the server receives a request, it must `write` back a *response*, which has a specific format.

```
HTTP/1.1 200 OK
Content-Type: text/plain

...response body...
```

status

header

not shown to user, only used by curl/browser

the thing to show for curl/in browser

As a C string literal:

```
"HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n"
```

(For status, you've probably seen a 404 error. This is a status for "couldn't find it"!)

Breaking Down The Response

1. **Status** - tells the client if the server was successful or not
    - 200 - Yay! Success
    - 404 - Couldn't find what you were looking for
    - A bunch more
2. **Header** - tells the client what kind of data is coming back from the server
    - For plaintext, it is `Content-Type: text/plain`
    - For a PNG image, it is `Content-Type: image/png`
    - We need a header because what's sent over is just bytes and the client needs to be told how to interpret the bytes
3. **Response body** - what is shown to clients
    - The actual bytes that was sent over

(Status and header is only used by `curl`/browser and not shown to the user)

Historical Note: `\r\n` instead of `\n`

- In HTTP, where we would expect to see a newline character (`\n`), we would actually see `\r\n`
- This is a result of HTTP being old and we can't change it now or a lot of things would break

As a C string literal:

```
"HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n"
```

Important: HTTP uses **\r\n** (not just **\n**) for line endings.

- Although it's fine to just have `\n` in the response body

So how do we actually send a response?

- We use the `write` system call
  - This is conceptually similar to `fprintf` that we used in PA3 for the pish_history file

Sending a response

`write(client, buf, len)` sends `len` bytes from `buf` to whoever made the request. Works the same as writing to a file – `client` is a file descriptor.

You can call `write` multiple times to build up a response:

```
write(client, HTTP_200, strlen(HTTP_200));
write(client, "Hello!", 6);
```

Note: `write` needs a length – it's not like `printf`. We'll often use `strlen` to compute it. Unlike C strings, HTTP responses don't use a null terminator – the length tells the receiver how many bytes we've written.

2 differences from `fprintf`

- The `client` file descriptor is an `int` instead of a `FILE*`
- `fprintf` takes a null terminated string, but `write` just takes a `char*` and the number of bytes to read

Let's put it in our `handler`!

Program 2: Send a response

```
 4 char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";
 5
 6 void handle(char *request, int client) {
 7     printf("Received request:\n%s\n", request);
 8     write(client, HTTP_200, strlen(HTTP_200));
 9     write(client, "Hello!", 6);
10 }
11
12 int main() { start_server(&handle, 2900); }
```

metadata
response body

Terminal output for `string-server2`

Server side

```
$ ./string-server2
Server started on port 2900
Received request:
GET /add?s=hi HTTP/1.1
Host: localhost:2900
User-Agent: curl/8.7.1
Accept: */*
```

Client side

```
$ curl "localhost:2900/add?s=hi"
Hello!
```

`printf` vs `write`

What we learned is

- `printf` should be used to display to the server terminal
- `write` should be used to send bytes over the network to the client

Questions

- Where does the `int client` come from?
  - When `start_server` gets a connection, it receives the `char`* for the bytes that came in and an int from the operating system representing the connection
  - This `int` is also associated with the port
- How does the requester know what kind of data they are requesting?
  - Requesters do not know
  - In the header from `curl`, it says “**Accept: */***”, meaning it will accept anything that is given
  - The server is sending back the type of data
- How does the client know they received all the bytes?
  - Two answers:
    - 1. The server closed the connection (like getting an EOF from a file, but for networks)
    - 2. Putting the content length in the header saying how many bytes was sent

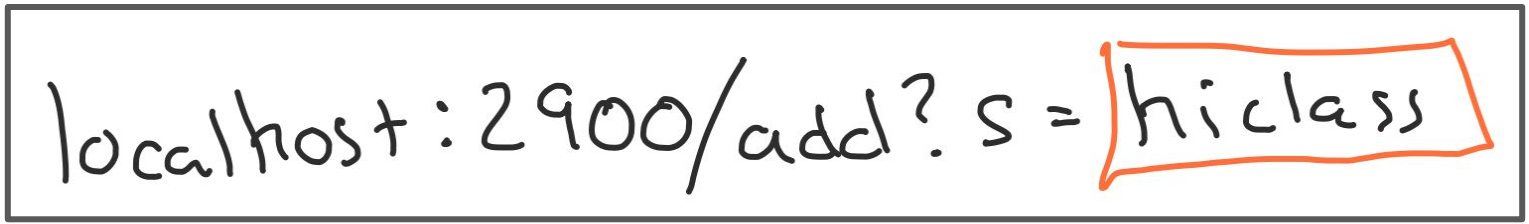
HTTP Request & Parsing

Our new challenge is to parse the string in the URL

- We know how to parse using things like `strtok`, `strchr`, etc.
- This is a string problem, but with HTTP it's a bit more interesting

Side Note: localhost

- localhost lets you connect to the computer you are on
- It means “send a request to the computer I’m on”
- It’s meant for testing



We’re not writing out the longer ieng6 part because we are on the ieng6 computer (we can use localhost)

What does the HTTP Request look like?

HTTP Request Format

The `request` string passed to our handler looks like:

```
GET /add?s=hi HTTP/1.1\r\n
Host: localhost:2900\r\n
User-Agent: curl/8.7.1\r\n
```

For our server, we want to extract the path and query string from the first line: `/add?s=hi`

All we care about is this little part that we want to take out for this application

- Remember this is called the "**path**" (what comes after the slash in a URL)

Parsing with `sscanf` and `strstr`

Parsing with sscanf and strstr

`sscanf` reads formatted data from a string – the inverse of `sprintf`:

```
char path[256];
sscanf(request, "GET %s", path);
// path is now "/add?s=hi" (%s stops matching at the space)
```

the matching string stored in path

`strstr(haystack, needle)` returns a pointer to the first occurrence of `needle` in `haystack`, or `NULL`. We can use it to find the `"?s="` for the query.

strstr(path, "?s=")

Let’s apply what we’ve learned

- We want our program to do what’s pictured below
  - This is not the final product yet where we have a list of strings, but we are getting there!

```
$ curl "localhost:2900/add?s=hi"
Added: hi
$ curl "localhost:2900/add"
Missing ?s= parameter
$ curl "localhost:2900/unknown"
Unknown path
```

Try to fill in `handle` in Program 3

Program 3: Parse the request

```
 4 char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";
 5 char *HTTP_404 = "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\n";
 6
 7 void send_404(int client, char *message) {
 8     write(client, HTTP_404, strlen(HTTP_404));
 9     write(client, message, strlen(message));
10 }
11
12 void handle(char *request, int client) {
13     char path[256];
14     sscanf(request, "GET %s", path);
15
16     if (strncmp(path, "/add", 4) == 0) {
17         char *query_start = strstr(path, "?s=");
18         if (query_start) {
23         } else {
24             send_404(client, "Missing ?s= parameter");
25         }
26     } else {
27         send_404(client, "Unknown path");
28     }
29 }
```

Program 3 Solution

```
11
12 void handle(char *request, int client) {
13     char path[256];
14     sscanf(request, "GET %s", path);
15
16     if (strncmp(path, "/add", 4) == 0) {
17         char *query_start = strstr(path, "?s=");
18         if (query_start) {
             char* val_start = query_start + 3;
             write(client, HTTP_200, strlen(HTTP_200));
             write(client, "Added: ", 7);
             write(client, val_start, strlen(val_start));
23         } else {
24             send_404(client, "Missing ?s= parameter");
25         }
26     } else {
27         send_404(client, "Unknown path");
28     }
29 }
```

check valid

"/add?s=hi"

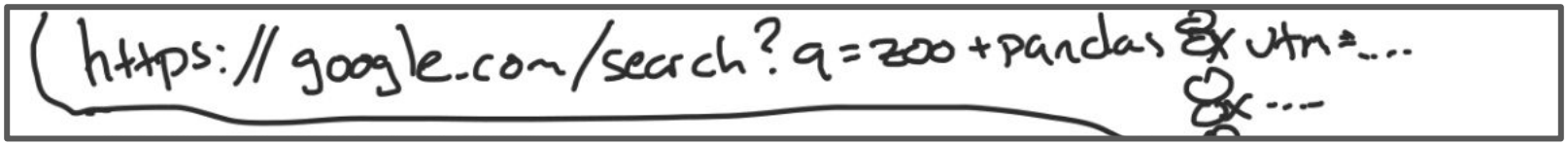
query-start

Questions

- What would happen if we didn’t send over the HTTP_200
  - The browser or curl would just say “invalid response” or “bad response”
- What does the `send_404` function do?
  - It sends back the 404 error code and also an extra message
  - `curl` will print the extra information
- Do we need to deal with the newline `\r\s` thing?
  - That is just in the string at the top (metadata in lines 4 and 5)
  - If we have newlines in our response body, we can just use `\n` like normal

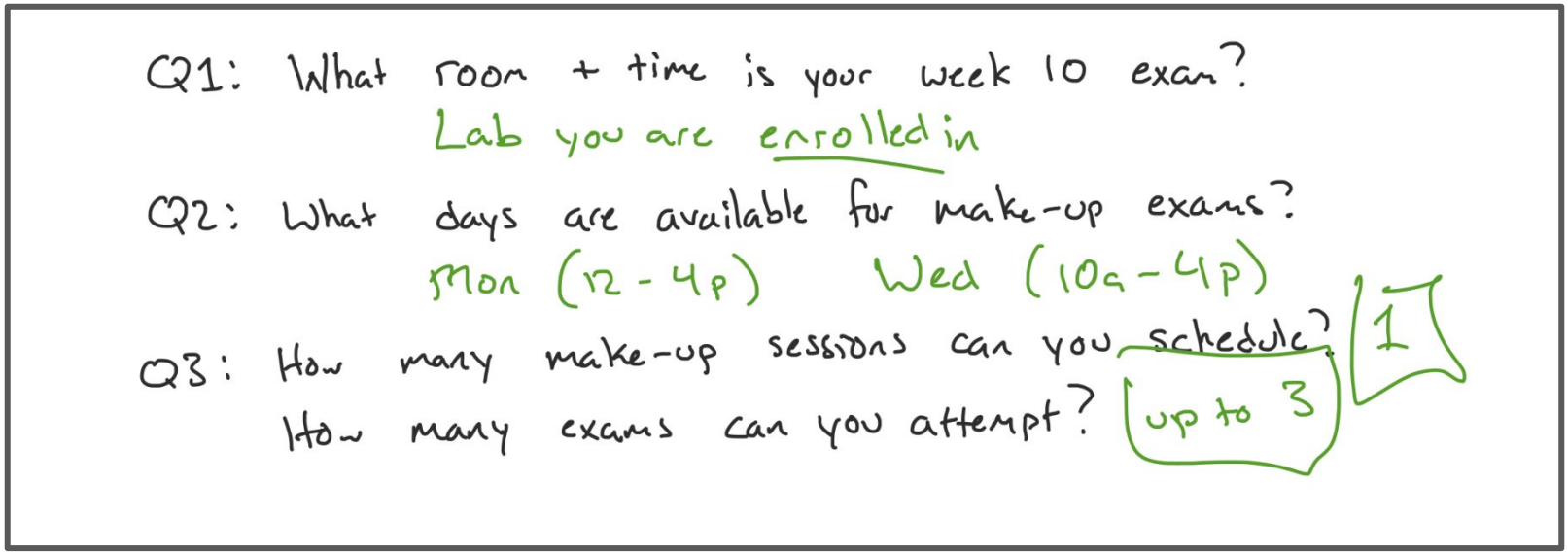
Side Note: URLs can be much more complicated

- When you make a Google Search request, you'd have a URL like this
  - Has a bunch of parameters separated by ampersands

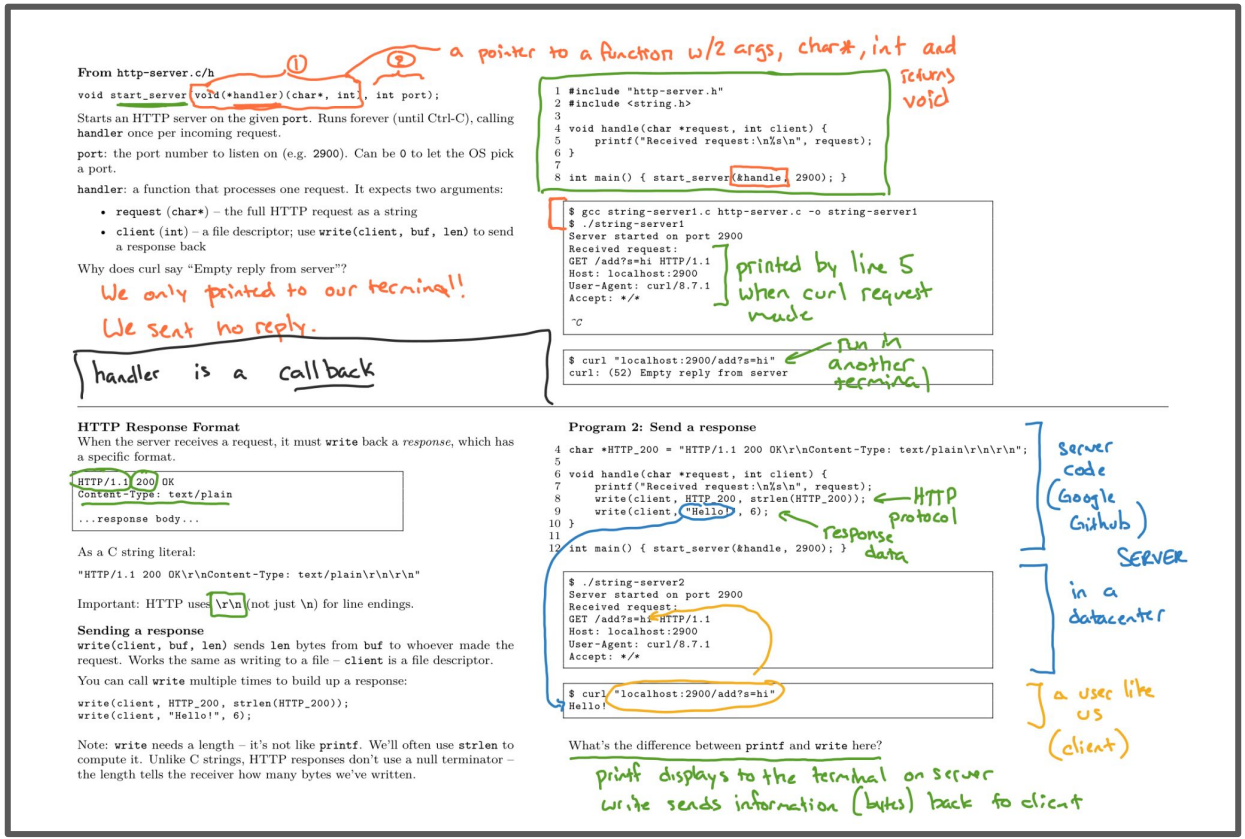


- Parsing these URLs are non-trivial and very hard with just `sscanf` and `strstr`
- Use a parsing library instead when you're in the real world

Joe's Notes (11am)



Joe's Notes (11am)



From `http-server.c/h`

a pointer to a function w/2 args, char*, int and returns void

```
void start_server(void(*handler)(char*, int), int port);
```

Starts an HTTP server on the given `port`. Runs forever (until Ctrl-C), calling `handler` once per incoming request.

`port`: the port number to listen on (e.g. `2900`). Can be `0` to let the OS pick a port.

`handler`: a function that processes one request. It expects two arguments:

- `request` (`char*`) – the full HTTP request as a string
- `client` (`int`) – a file descriptor; use `write(client, buf, len)` to send a response back

Why does curl say "Empty reply from server"?

We only printed to our terminal!

We sent no reply.

handler is a callback

```
1 #include "http-server.h"
2 #include <string.h>
3
4 void handle(char *request, int client) {
5     printf("Received request:\n%s\n", request);
6 }
7
8 int main() { start_server(&handle, 2900); }
```

```
$ gcc string-server1.c http-server.c -o string-server1
$ ./string-server1
Server started on port 2900
Received request:
GET /add?s=hi HTTP/1.1
Host: localhost:2900
User-Agent: curl/8.7.1
Accept: */*
^C
```

printed by line 5 when curl request made

```
$ curl "localhost:2900/add?s=hi"
curl: (52) Empty reply from server
```

run in another terminal

HTTP Response Format

When the server receives a request, it must `write` back a *response*, which has a specific format.

```
HTTP/1.1 200 OK
Content-Type: text/plain

...response body...
```

As a C string literal:

```
"HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n"
```

Important: HTTP uses `\r\n` (not just `\n`) for line endings.

Sending a response

`write(client, buf, len)` sends `len` bytes from `buf` to whoever made the request. Works the same as writing to a file – `client` is a file descriptor.

You can call `write` multiple times to build up a response:

```
write(client, HTTP_200, strlen(HTTP_200));
write(client, "Hello!", 6);
```

Note: `write` needs a length – it's not like `printf`. We'll often use `strlen` to compute it. Unlike C strings, HTTP responses don't use a null terminator – the length tells the receiver how many bytes we've written.

Program 2: Send a response

```
4  char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";
5
6  void handle(char *request, int client) {
7      printf("Received request:\n%s\n", request);
8      write(client, HTTP_200, strlen(HTTP_200));
9      write(client, "Hello!", 6);
10 }
11
12 int main() { start_server(&handle, 2900); }
```

HTTP protocol

response data

server code (Google Github)

SERVER in a datacenter

```
$ ./string-server2
Server started on port 2900
Received request:
GET /add?s=hi HTTP/1.1
Host: localhost:2900
User-Agent: curl/8.7.1
Accept: */*
```

```
$ curl "localhost:2900/add?s=hi"
Hello!
```

a user like us (client)

What's the difference between `printf` and `write` here?

printf displays to the terminal on server

write sends information (bytes) back to client

Joe's Notes (11am)

path

HTTP Request Format

The **request** string passed to our handler looks like:

```
GET /add?s=hi HTTP/1.1\r\n
Host: localhost:2900\r\n
User-Agent: curl/8.7.1\r\n
```

query

For our server, we want to extract the path and query string from the first line: **/add?s=hi**

Parsing with sscanf and strstr

sscanf reads formatted data from a string – the inverse of **sprintf**:

```
char path[256];
sscanf(request, "GET %s", path);
// path is now "/add?s=hi" (%s stops matching at the space)
```

store this string in path

strstr(haystack, needle) returns a pointer to the first occurrence of **needle** in **haystack**, or **NULL**. We can use it to find the **"?s="** for the query.

```
$ curl "localhost:2900/add?s=hi"
Added: hi
$ curl "localhost:2900/add"
Missing ?s= parameter
$ curl "localhost:2900/unknown"
Unknown path
```

Program 3: Parse the request

```
4  char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";
5  char *HTTP_404 = "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\n";
6
7  void send_404(int client, char *message) {
8      write(client, HTTP_404, strlen(HTTP_404));
9      write(client, message, strlen(message));
10 }
11
12 void handle(char *request, int client) {
13     char path[256];
14     sscanf(request, "GET %s", path);
15
16     if (strncmp(path, "/add", 4) == 0) {
17         char *query_start = strstr(path, "?s=");
18         if (query_start) {
23         } else {
24             send_404(client, "Missing ?s= parameter");
25         }
26     } else {
27         send_404(client, "Unknown path");
28     }
29 }
```

"/add?s=hi"

query_start

write(client, HTTP_200, strlen(HTTP_200));

write(client, query_start + 3, strlen(query_start + 3))

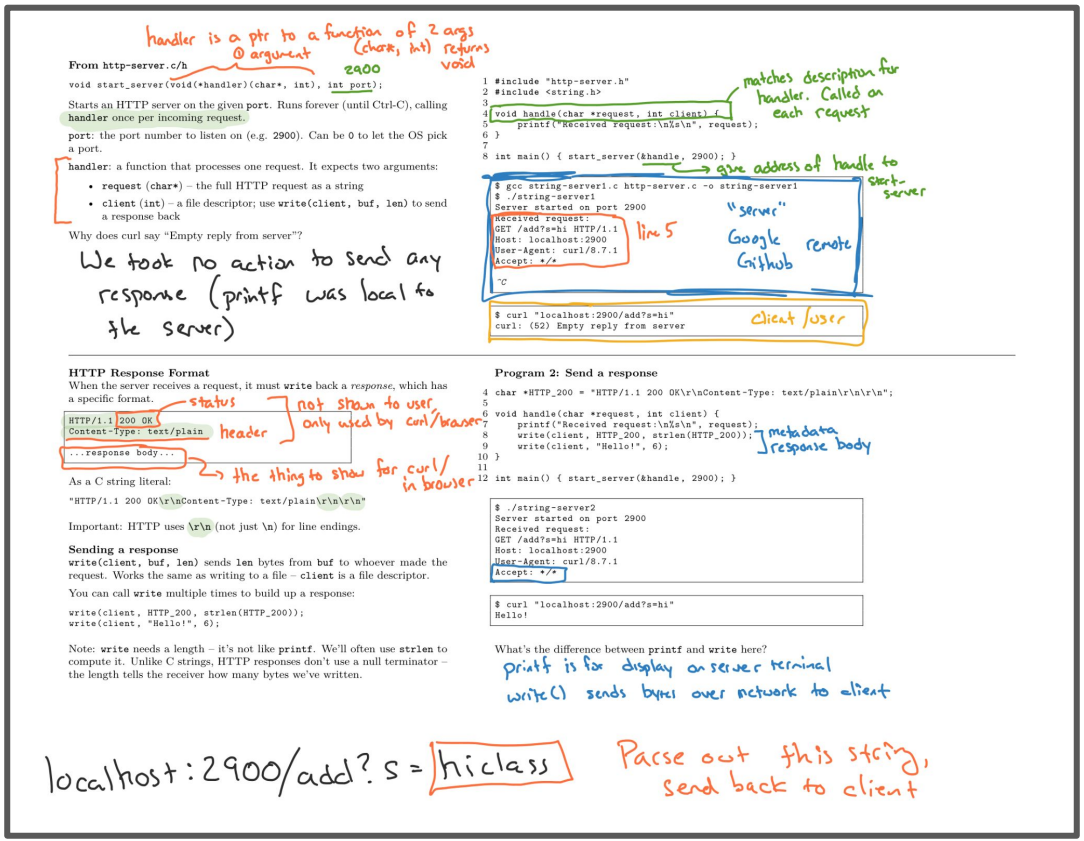
Joe's Notes (12:30pm)

Q1: What room + time is your week 10 exam?
The lab you are enrolled in

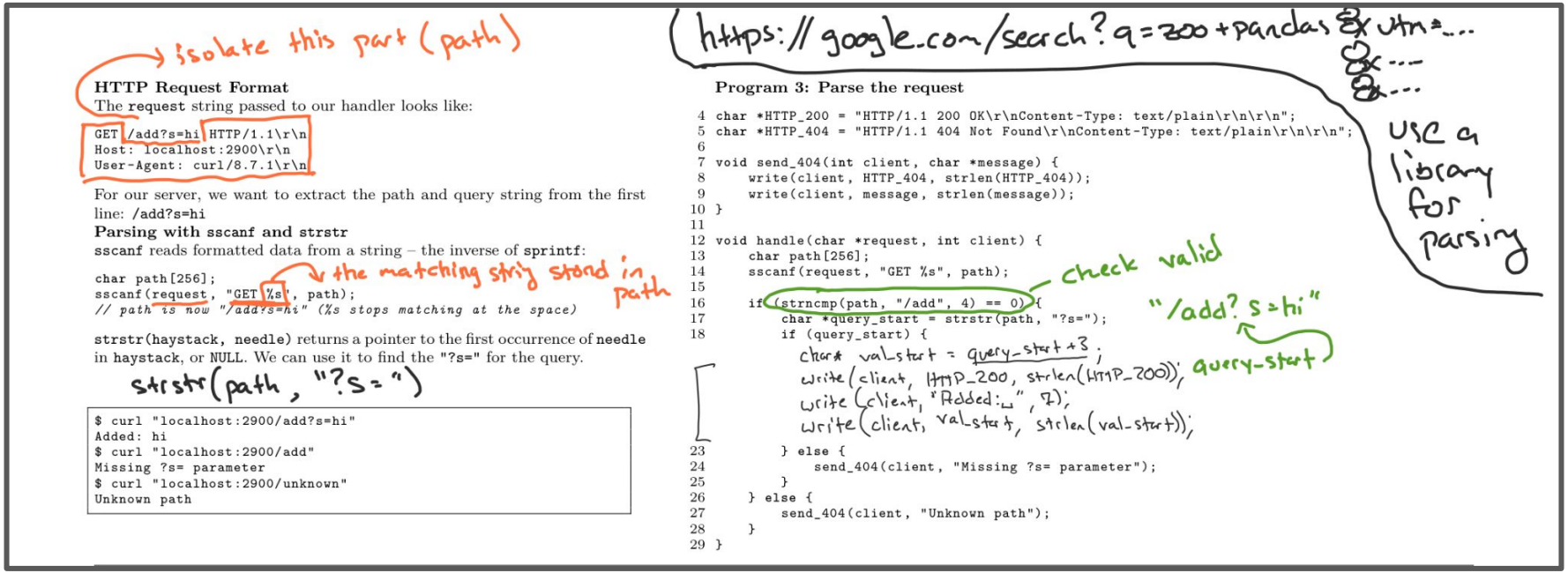
Q2: What days are available for make-up exams?
Mon 12p-2 2p-4p Wed 10-12, 12-2, 2-4 (1h40)

Q3: How many make-up sessions can you schedule? 1
How many exams can you attempt? up to 3

Joe's Notes (12:30pm)



Joe's Notes (12:30pm)



isolate this part (path)

HTTP Request Format

The request string passed to our handler looks like:

```
GET /add?s=hi HTTP/1.1\r\n
Host: localhost:2900\r\n
User-Agent: curl/8.7.1\r\n
```

For our server, we want to extract the path and query string from the first line: /add?s=hi

Parsing with sscanf and strstr

sscanf reads formatted data from a string – the inverse of sprintf:

```
char path[256];
sscanf(request, "GET %s", path);
// path is now "/add?s=hi" (%s stops matching at the space)
```

the matching strig stored in path

strstr(haystack, needle) returns a pointer to the first occurrence of needle in haystack, or NULL. We can use it to find the "?s=" for the query.

strstr(path, "?s=")

```
$ curl "localhost:2900/add?s=hi"
Added: hi
$ curl "localhost:2900/add"
Missing ?s= parameter
$ curl "localhost:2900/unknown"
Unknown path
```

https://google.com/search?q=300+pandas&utm=...

Program 3: Parse the request

```
4  char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";
5  char *HTTP_404 = "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\n";
6
7  void send_404(int client, char *message) {
8      write(client, HTTP_404, strlen(HTTP_404));
9      write(client, message, strlen(message));
10 }
11
12 void handle(char *request, int client) {
13     char path[256];
14     sscanf(request, "GET %s", path);
15
16     if (strncmp(path, "/add", 4) == 0) {
17         char *query_start = strstr(path, "?s=");
18         if (query_start) {
               char* val_start = query_start + 3;
               write(client, HTTP_200, strlen(HTTP_200));
               write(client, "Added: ", 7);
               write(client, val_start, strlen(val_start));
23         } else {
24             send_404(client, "Missing ?s= parameter");
25         }
26     } else {
27         send_404(client, "Unknown path");
28     }
29 }
```

check valid

"/add?s=hi"

query_start

use a library for parsing