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 - 4p)   Wed (10a - 4p)

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

How many exams can you attempt? up to 3

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

From http-server.c/h

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

① ② a pointer to a function w/2 args, char*, int and returns void

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

```
char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";

void handle(char *request, int client) {
    printf("Received request:\n%s\n", request);
    write(client, HTTP_200, strlen(HTTP_200));
    write(client, "Hello!", 6);
}

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

HTTP protocol

response data

server code (Google Github)

SERVER

```
$ ./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: */*
```

in a datacenter

```
$ 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 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:

store this string in path

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

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

```
char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";
char *HTTP_404 = "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\n";

void send_404(int client, char *message) {
    write(client, HTTP_404, strlen(HTTP_404));
    write(client, message, strlen(message));
}

void handle(char *request, int client) {
    char path[256];
    sscanf(request, "GET %s", path);   "/add?s=hi"

    if (strncmp(path, "/add", 4) == 0) {
        char *query_start = strstr(path, "?s=");
                                              query-start
        if (query_start) {
          write(client,
               HTTP_200, strlen(HTTP_200));

        write(client, query-start + 3,
                        strlen(query-start+3))
        } else {
            send_404(client, "Missing ?s= parameter");
        }
    } else {
        send_404(client, "Unknown path");
    }
}
```

Program 4: Keeping state across requests

Unlike a program that runs and exits, a server runs forever. Data that should persist between requests can't live on the stack – it needs to be in global variables or on the heap.

The copy problem

The request buffer in http-server.c is reused for every request. If we store a pointer into it, that pointer's data gets overwritten on the next request.

```
#define MAX_STRINGS 100

char *strings[MAX_STRINGS];
int num_strings = 0;

void add_string(char *s) {
    if (num_strings >= MAX_STRINGS) { return; }
    strings[num_strings] = malloc(strlen(s) + 1);
    strcpy(strings[num_strings], s);
    num_strings++;
}

void respond_with_list(int client) {
```

handle has the same structure as Program 3, but calls add_string and respond_with_list:

```
void handle(char *request, int client) {
    char path[256];
    sscanf(request, "GET %s", path);

    if (strncmp(path, "/add", 4) == 0) {
        char *query_start = strstr(path, "?s=");
        if (query_start) {
            char *string_start = query_start + 3;
            add_string(string_start);
            respond_with_list(client);
        } else {
            send_404(client, "Missing ?s= parameter");
        }
    } else {
        send_404(client, "Unknown path");
    }
}
```

```
$ curl "localhost:2900/add?s=hello"
hello
$ curl "localhost:2900/add?s=world"
hello
world
$ curl "localhost:2900/add?s=goodbye"
hello
world
goodbye
```

handler is a ptr to a function of 2 args (char*, int) returns void

① argument

From http-server.c/h

2900

```
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 took no action to send any response (printf was local to the server)

```
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); }
```

matches description for handler. Called on each request

give address of handle to start-server

```
$ 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
```

line 5

"server"

Google

Github

remote

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

Client/user

HTTP Response Format

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

status

not shown to user, only used by curl/browser

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

...response body...
```

header

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"
```

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); }
```

metadata

response body

```
$ ./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!
```

What's the difference between printf and write here?

printf is for display on server terminal

write() sends bytes over network to client

localhost:2900/add?s=hiclass

Parse out this string, send back to client 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 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=")

```
$ 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=zoo+pandas&utm=...
&x...
&x...

Use a library for parsing

Program 3: Parse the request

```
char *HTTP_200 = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\n\r\n";
char *HTTP_404 = "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\n";

void send_404(int client, char *message) {
    write(client, HTTP_404, strlen(HTTP_404));
    write(client, message, strlen(message));
}

void handle(char *request, int client) {
    char path[256];
    sscanf(request, "GET %s", path);

    if (strncmp(path, "/add", 4) == 0) {
        char *query_start = strstr(path, "?s=");
        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));
        } else {
            send_404(client, "Missing ?s= parameter");
        }
    } else {
        send_404(client, "Unknown path");
    }
}
```

check valid

"/add?s=hi"

query-start

Program 4: Keeping state across requests

Unlike a program that runs and exits, a server runs forever. Data that should persist between requests can't live on the stack – it needs to be in global variables or on the heap.

The copy problem

The **request** buffer in **http-server.c** is reused for every request. If we store a pointer into it, that pointer's data gets overwritten on the next request.

```
#define MAX_STRINGS 100

char *strings[MAX_STRINGS];
int num_strings = 0;

void add_string(char *s) {
    if (num_strings >= MAX_STRINGS) { return; }
    strings[num_strings] = malloc(strlen(s) + 1);
    strcpy(strings[num_strings], s);
    num_strings++;
}

void respond_with_list(int client) {
```

handle has the same structure as Program 3, but calls **add_string** and **respond_with_list**:

```
void handle(char *request, int client) {
    char path[256];
    sscanf(request, "GET %s", path);

    if (strncmp(path, "/add", 4) == 0) {
        char *query_start = strstr(path, "?s=");
        if (query_start) {
            char *string_start = query_start + 3;
            add_string(string_start);
            respond_with_list(client);
        } else {
            send_404(client, "Missing ?s= parameter");
        }
    } else {
        send_404(client, "Unknown path");
    }
}
```

```
$ curl "localhost:2900/add?s=hello"
hello
$ curl "localhost:2900/add?s=world"
hello
world
$ curl "localhost:2900/add?s=goodbye"
hello
world
goodbye
```