Note: The symbol (=>) means what a certain function returns or what the outcome is from calling that function

Main function:

- **Purpose:**
    - Implement an HTTP server using sockets
        - Supports GET and PUT requests to read and write “files”.
        - Stores files persistently in a directory on the server
            - Server can be restarted, or be able to run on a directory that already has files
- **Inputs:**
    - **Server**
        - To boot up the server, the inputs are in the command-line arguments.
        - The user can declare the inputs of the IP and port that the server will be hosted on
        - If no port is specified, port 80 will be used
    - **Client**
        - The client can only send HTTP requests to the server and may or may not include the Content Length, (For example: curl requests)
- **Outputs:**
    - The server will receive the HTTP requests and send back a response to the client. Or it can fail the request and throw an error.

Detailed Functionality:

- **Creating the socket and listening for a connection**
    - The usage of our program is likeso:
        - “sudo ./httpserver <ip> [port]”
    - First assign the hostname to the <ip> provided in the command-line arguments
        - Set the default port as 80, or use the optional [port] that is provided in the command-line argument
        - ip, port = argv[1], argv[2]
        - Note: argv is our command-line arguments
    - Using that port and ip assign it to our serverAddress structure
    - Create a server_socket file descriptor using socket(AF_INET, SOCK_STREAM)
    - Note: AF_INET (ipv4 protocol), SOCK_STREAM (a reliable two-way connection)

        - socket(AF_INET, SOCK_STREAM) => New server socket file descriptor
        - If there is an error creating the socket. Throw an error and exit our program
    - Using setsockopt(), we allow our server_socket file descriptor to reuse a port/ip that is in use in order to avoid “Bind: Already In Use” error
        - setsockopt(server_socket_fd) => Options are now set to reuse
        - If there is an error setting the socket options. Throw an error and exit our program
    - Bind our server_socket to the port number and IP address specified in our server address structure
        - bind(server_socket_fd, serverAddress) => Socket is now binded to the server address with the specified port and ip
        - If there is an error on binding. Throw an error and exit our program
    - Listen for a connection to our server_socket
        - listen(server_sock_fd) => Wait for a connection to our server
        - If there is an error on listening. Throw an error and exit our program

- **Waiting for a connection**
    - Now we just while(1) loop until the user closes the server by exiting the program with Ctrl+C
    - Accept any connection sent to our server
        - client_sock_fd = accept(server_sock_fd, serverAddress)
            - This returns a new file descriptor and establishes a connection between client and server
            - If there is an error on accepting. Throw an error and exit our program
    - Attempt to receive the HTTP header from our client socket and put it into our buffer
        - recv(client_sock_fd, buffer) => Number of bytes received that is now in our buffer
        - If there is an error on receiving. Throw an error and exit our program
            - Since there is a possibility we do not receive the header in one receive call, we must check if “\r\n\r\n” was received in the buffer. (this indicates that we received the whole header)
            - If it wasn’t received in the buffer, we keep receiving until a “\r\r\r\n” is found. Keep resetting buffer and receiving until it is found

- **Parse the HTTP header and validity check of incoming requests**
    - Now that we have the http header in our buffer we must parse it
        - Use sscanf(buffer, request, filename, protocol)
            - Doing this will put the request, filename, and protocol into their respective character arrays taken from the buffer
            - Because HTTP puts a “/” in front of the filename, all we do is increment the filename by 1 to get rid of it
    - Now that we have all of our necessary header information, we must check if they are valid
        - Using strcmp(protocol, “HTTP/1.1”) we check if two strings are equal.
            - If the protocol version in the request is not 1.1 we send a 505 response, HTTP version Not Supported, and go back to the start of our while loop to wait for another connection request
        - Using strlen(filename) we check if the filename is length 10
            - If the filename in the request is not length 10 we send a 400 response, Bad Request, and go back to the start of our while loop to wait for another connection request
        - Using isalnum(*i), we loop through our filename and check if they are valid characters
            - If a character in our requested filename is not a valid character, we send a 400 response, Bad Request, and go back to the start of our while loop to wait for another connection request
        - Using strcmp(request) we check if our HTTP request is something other than GET/PUT
            - If the request is not a GET or PUT, we send a 400 response, Bad Request, and go back to the start of our while loop to wait for another connection request.

- **Processing a PUT request**
    - Using strcmp(request,"PUT") we check if it is a put request
    - We then check if the file already exists in our directory
        - stat(filename, buffer) => returns 0 or -1 indicating if the file exists or not
        - We do this before we open the file
        - Depending on if a file exists or not we respond with 200 or 201 indicating if the file was created or OK (file is overwritten, etc.)
    - We then open/create a file descriptor for read/write + truncation permissions which returns our newly opened file descriptor we will call put_fd
        - open(filename, O_CREAT|O_RD_WR|O_TRUNC) => creates or opens a file in our server's directory with the given filename from the HTTP header
            - If there is an error opening the existing file, it must mean we have a permissions error, we send a 403 response, Forbidden, and go back to the start of our while loop to wait for another connection request
    - If everything is fine, we need to parse the Content-Length in our HTTP header.
        - We use token = strtok(buffer, "\r\n") and strtok(NULL, "\r\n") to split up the buffer into tokens we can loop through that are split by "\r\n"
        - We loop through all of the tokens and search for the string content-length: "N".
        - We use a variable found = strstr(token, "Content-Length"). Once found is not equal to NULL, that means we have located the Content-Length string.
            - So we use sscanf(token, content_length) to load the token into our content_length variable
    - If we received a request with Content-Length 0.
        - This means that the file is empty so we just create or overwrite a file and make it empty. So we send the correct 200 or 201 response depending on if the file already exists and go back to the start and wait for the next connection request
    - Now we start receiving the bytes sent from our client
        - recv(client_sock_fd, buffer,1) => Number of bytes received
            - If there is an error receiving, we respond with a 500 Internal Server Error, and go back to the start of our while loop to wait for another connection request

        - We keep receiving from the client and writing to our opened file descriptor, put_fd. We keep doing this until we reach EOF or the content length limit given in the HTTP header. While we are doing this we are keeping track of the amount of bytes we have written, and check if the amount of bytes written is equal to our provided content length.
            - If the number of bytes_writen is equal to content length or recv() returns 0 bytes (EOF), we send back a response to the server 201/200 depending on if the file already exists. Then we go back to the start of the while loop and wait for the next connection request.

- **Processing a GET request**
    - Using strcmp(request, “GET”) we check if it is a get request
    - We then try to open the requested file which returns a new file descriptor
        - We open with READ access only and give permissions to the file owner
        - f_desc = open(filename, O_RDYONLY, S_IRWXU) -> this either returns a non-negative integer(opening success) or -1 (failed to open)
            - If open() returns -1, we must check if the file exists or not to determine the correct error response to send back to the client
            - We use fstat(open_file_descriptor, strict stat &finfo). This returns us the status of whether the file exists or not.
            - We also check the size of the file in bytes using finfo.st_size so that we can respond with the correct content length along with our status code
                - If the file doesn’t exist it just means file is not found
                    - Respond 404 and go back to the start of the while loop to wait for another connection request
                - If the file does exist it just means our current user does not have permission to open the file
                    - Respond 403 and go back to the start of the while loop to wait for another connection request
        - Otherwise, if there is no error when using open() we continue program execution

- Now we create a while loop and start reading from the open file descriptor and writing to the client_socket_fd
    - x = read(f_desc) => returns the number of bytes read
    - write(client_sock_fd, x) => writes the number of bytes read to the client_socket
    - Otherwise, we check at every loop whether read() returns an error or not, read() == -1. If there is an error
        - Respond 500 and go back to the start of the while loop to wait for another connection request
    - Otherwise, if nothing bad occurs, we were able to read and write all the bytes to the client_socket
        - Respond 200 and go back to the start of the while loop to wait for another connection request

High-level Rundown of Program Functionality

Create a socket that listens for connections on an ip/port

- Get user-defined arguments, ip and port
- Declare our serverAddress structure, assign the ip and port
- Create client_socket_fd
- Set socket options
- Bind the socket to the ip and port
- Allow the socket to listen for a connection

Wait for a connection/ parse HTTP header/ check for valid requests

- Accept a connection that is sent to our specified ip and port
- Receive the data that is sent through our connection, and put it in our buffer
- Make sure we have received the whole HTTP header into our buffer
- Parse the buffer and assign the request, filename, and protocol version
- HTTP provides a "/" to the filename so just increment it
- If HTTP version is not 1.1
    - 505. Wait for next request
- If filename is not length 10
    - 400. Wait for next request
- Filename is not alphanumeric
    - 400. Wait for next request
- Not a PUT/GET request
    - 400. Wait for next request
- Otherwise,
    - Is it a PUT/GET request?
        - Continue through the program

Processing a PUT request

- Open our file for create, truncation, read/write permissions
    - Error opening file - we don't have permissions
        - 403. Wait for next request
- Extract the content length from the HTTP header
- Check if content-length = 0 (empty file)
    - Write to the requested file and overwrite it to be empty
        - If file exist before this request
            - 200. Wait for next request
        - If file did not exist before this request
            - 201. Wait for next request

- Receive from the client socket until there is an error, reached EOF, or reached content length
    - Receive from client socket
    - Write the amount of bytes received to the opened file
    - Keep track of how many bytes we have written and compare to the content length. If amount of bytes written is equal to the content length we break out the while loop
        - If there is an error receiving at any point
            - 500. Wait for next request
        - Otherwise,
            - If file exist before this request
                - 200. Wait for next request
            - If file did not exist before this request
                - 201. Wait for next request

Processing a GET request

- Open our file for read access permissions
    - Error opening file
        - Check if the file exists in our directory
        - Check for the file size in bytes. In our responses include Content-Length: (file size in bytes)
            - If a file exists. Therefore, permission error
                - 403. Wait for next request
            - If the file does not exist. File not found
                - 404. Wait for next request
- Start reading the file and writing to the client until we reach EOF or an error occurs
    - If an error occurs reading the file
        - 500. Wait for next request
    - Otherwise, if we reach EOF and nothing bad occurs, we are all good and the client received everything that is in the file
        - 200. Wait for next request

Testing

For testing I did unit-testing and whole-system testing

Testing the socket ip/port

I first tested if my server correctly listened for and was able to accept connections on user defined ports and ips
On the server terminal I used commands like

- $ sudo ./httpserver localhost 80

On the client terminal I used commands like

- $ telnet localhost 80

I did various combinations of ips and ports to make sure it worked correctly

Testing PUT requests

I then used various curl commands to check if PUT requests worked properly
I used various files with different data - binary, etc.
I also used various HTTP file names

- $ curl -T file.txt localhost:8080/abcdefghij -v

Then, I used commands to check the difference between the file on the server and the file on the client

- $ diff -s abcdefghij file.txt

Testing GET requests

I then used various curl commands to check if GET requests worked properly
I used various files with different data - binary, etc.
I also used various HTTP file names

- $ curl localhost:8080/abcdefghij > output

Then I used commands to check the difference between the file on the server and the file on the client

- $ diff -s file.txt output

Testing Edge Cases

I tested all the examples Daniel provided in his slides, and a bunch of other curl commands.

Basic examples:

~~~~~~~~~~~~~~~~~~~~

curl localhost:8080/abcdefghij

- Requests file abcdefghij from a server running in localhost on port 8080

curl -v localhost:8080/abcdefghij

- Same, but verbose version (note -v)
- Will output request sent and response head received

curl ‘localhost:8080/123456789[0-3]’

- Will request files 1234567890, 1234567891, 1234567892, and 1234567893
- Useful for testing multiple requests
- Notice the quotes surrounding the address, it’s important to stop your shell from trying to expand the expression

curl -T file.txt localhost:8080/abcdefghij

- Sends content of file.txt to be stored by the server in file abcdefghij

Trickier examples:

~~~~~~~~~~~~~~~~~~~~

curl localhost:8080/abcdefghij -o output.file

- File retrieved is store in output.file
- Useful for testing transfer of binary data
- If using verbose (-v), that won’t be in the output file

curl localhost:8080/abcdefghij --header ‘Content-Length: 10’

- The --header flag allows you to add extra headers, or overwrite existing headers in the request

curl localhost:8080/1234567890 --request DELETE

- This will change the request type to DELETE
- Use this for testing identification of non-implemented request types

curl localhost:8080/1234567890 --request-target 1234567890

- This will change the target from /1234567890 to 1234567890

Assignment Question

- What happens in your implementation if, during a PUT with a `Content-Length`, the connection was closed, ending the communication early? This extra concern was not present in your implementation of `dog`. Why not? Hint: this is an example of complexity being added by an extension of requirements (in this case, data transfer over a network).

I assume you mean the PUT request would send the HTTP header and body. Then the connection would close there at that moment.

Therefore, the server would complete the PUT request, but would not be able to send a response back to the client. Then it would just wait for the next connection.