Міністерство освіти і науки України

Національний університет «Львівська політехніка»



**Лабораторна робота №13**

на тему:

«Робота з сервером»

з курсу:

«Алгоритмізація та програмування»

Виконав:

ст. гр. КН-110

Король Орест

Прийняв:

Кривенчук Ю.П.

Львів – 2017 р.

*Лабораторна робота №13*

***Тема роботи:*** *навчитись користуватись сервером і набути вмінь у його реалізації*

***Мета роботи:*** покращити свої навички у роботі з мовою програмування С. Навчитись працювати з локальним сервером та здобути вміння у роботі з веб технологіями

**Завдання**

Запустити сервер і виконати усі інструкції​

**Текст програми**

#

. #define \_GNU\_SOURCE

#define \_XOPEN\_SOURCE 700

#define \_XOPEN\_SOURCE\_EXTENDED

// limits on an HTTP request's size, based on Apache's

// http://httpd.apache.org/docs/2.2/mod/core.html

#define LimitRequestFields 50

#define LimitRequestFieldSize 4094

#define LimitRequestLine 8190

// number of octets for buffered reads

#define OCTETS 512

// header files

#include <arpa/inet.h>

#include <errno.h>

#include <limits.h>

#include <math.h>

#include <signal.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <strings.h>

#include <unistd.h>

// types

typedef char octet;

// prototypes

bool connected(void);

bool error(unsigned short code);

void handler(int signal);

ssize\_t load(void);

const char\* lookup(const char\* extension);

ssize\_t parse(void);

void reset(void);

void start(short port, const char\* path);

void stop(void);

// server's root

char\* root = NULL;

// file descriptor for sockets

int cfd = -1, sfd = -1;

// buffer for request

octet\* request = NULL;

// FILE pointer for files

FILE\* file = NULL;

// buffer for response-body

octet\* body = NULL;

int main(int argc, char\* argv[])

{

// a global variable defined in errno.h that's "set by system

// calls and some library functions [to a nonzero value]

// in the event of an error to indicate what went wrong"

errno = 0;

// default to a random port

int port = 0;

// usage

const char\* usage = "Usage: server [-p port] /path/to/root";

// parse command-line arguments

int opt;

while ((opt = getopt(argc, argv, "hp:")) != -1)

{

switch (opt)

{

// -h

case 'h':

printf("%s\n", usage);

return 0;

// -p port

case 'p':

port = atoi(optarg);

break;

}

}

// ensure port is a non-negative short and path to server's root is specified

if (port < 0 || port > SHRT\_MAX || argv[optind] == NULL || strlen(argv[optind]) == 0)

{

// announce usage

printf("%s\n", usage);

// return 2 just like bash's builtins

return 2;

}

// start server

start(port, argv[optind]);

// listen for SIGINT (aka control-c)

signal(SIGINT, handler);

// accept connections one at a time

while (true)

{

// reset server's state

reset();

// wait until client is connected

if (connected())

{

// parse client's HTTP request

ssize\_t octets = parse();

if (octets == -1)

{

continue;

}

// extract request's request-line

// http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html

const char\* haystack = request;

char\* needle = strstr(haystack, "\r\n");

if (needle == NULL)

{

error(400);

continue;

}

else if (needle - haystack + 2 > LimitRequestLine)

{

error(414);

continue;

}

char line[needle - haystack + 2 + 1];

strncpy(line, haystack, needle - haystack + 2);

line[needle - haystack + 2] = '\0';

// log request-line

printf("%s", line);

// TODO: validate request-line

// TODO: extract query from request-target

char query[] = "TODO";

// TODO: concatenate root and absolute-path

char path[] = "TODO";

// TODO: ensure path exists

// TODO: ensure path is readable

// TODO: extract path's extension

char extension[] = "TODO";

// dynamic content

if (strcasecmp("php", extension) == 0)

{

// open pipe to PHP interpreter

char\* format = "QUERY\_STRING=\"%s\" REDIRECT\_STATUS=200 SCRIPT\_FILENAME=\"%s\" php-cgi";

char command[strlen(format) + (strlen(path) - 2) + (strlen(query) - 2) + 1];

sprintf(command, format, query, path);

file = popen(command, "r");

if (file == NULL)

{

error(500);

continue;

}

// load file

ssize\_t size = load();

if (size == -1)

{

error(500);

continue;

}

// subtract php-cgi's headers from body's size to get content's length

haystack = body;

needle = memmem(haystack, size, "\r\n\r\n", 4);

if (needle == NULL)

{

error(500);

continue;

}

size\_t length = size - (needle - haystack + 4);

// respond to client

if (dprintf(cfd, "HTTP/1.1 200 OK\r\n") < 0)

{

continue;

}

if (dprintf(cfd, "Connection: close\r\n") < 0)

{

continue;

}

if (dprintf(cfd, "Content-Length: %i\r\n", length) < 0)

{

continue;

}

if (write(cfd, body, size) == -1)

{

continue;

}

}

// static content

else

{

// look up file's MIME type

const char\* type = lookup(extension);

if (type == NULL)

{

error(501);

continue;

}

// open file

file = fopen(path, "r");

if (file == NULL)

{

error(500);

continue;

}

// load file

ssize\_t length = load();

if (length == -1)

{

error(500);

continue;

}

// TODO: respond to client

}

// announce OK

printf("\033[32m");

printf("HTTP/1.1 200 OK");

printf("\033[39m\n");

}

}

}

/\*\*

\* Accepts a connection from a client, blocking (i.e., waiting) until one is heard.

\* Upon success, returns true; upon failure, returns false.

\*/

bool connected(void)

{

struct sockaddr\_in cli\_addr;

memset(&cli\_addr, 0, sizeof(cli\_addr));

socklen\_t cli\_len = sizeof(cli\_addr);

cfd = accept(sfd, (struct sockaddr\*) &cli\_addr, &cli\_len);

if (cfd == -1)

{

return false;

}

return true;

}

/\*\*

\* Handles client errors (4xx) and server errors (5xx).

\*/

bool error(unsigned short code)

{

// ensure client's socket is open

if (cfd == -1)

{

return false;

}

// ensure code is within range

if (code < 400 || code > 599)

{

return false;

}

// determine Status-Line's phrase

// http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.1

const char\* phrase = NULL;

switch (code)

{

case 400: phrase = "Bad Request"; break;

case 403: phrase = "Forbidden"; break;

case 404: phrase = "Not Found"; break;

case 405: phrase = "Method Not Allowed"; break;

case 413: phrase = "Request Entity Too Large"; break;

case 414: phrase = "Request-URI Too Long"; break;

case 418: phrase = "I'm a teapot"; break;

case 500: phrase = "Internal Server Error"; break;

case 501: phrase = "Not Implemented"; break;

case 505: phrase = "HTTP Version Not Supported"; break;

}

if (phrase == NULL)

{

return false;

}

// template

char\* template = "<html><head><title>%i %s</title></head><body><h1>%i %s</h1></body></html>";

char content[strlen(template) + 2 \* ((int) log10(code) + 1 - 2) + 2 \* (strlen(phrase) - 2) + 1];

int length = sprintf(content, template, code, phrase, code, phrase);

// respond with Status-Line

if (dprintf(cfd, "HTTP/1.1 %i %s\r\n", code, phrase) < 0)

{

return false;

}

// respond with Connection header

if (dprintf(cfd, "Connection: close\r\n") < 0)

{

return false;

}

// respond with Content-Length header

if (dprintf(cfd, "Content-Length: %i\r\n", length) < 0)

{

return false;

}

// respond with Content-Type header

if (dprintf(cfd, "Content-Type: text/html\r\n") < 0)

{

return false;

}

// respond with CRLF

if (dprintf(cfd, "\r\n") < 0)

{

return false;

}

// respond with message-body

if (write(cfd, content, length) == -1)

{

return false;

}

// announce Response-Line

printf("\033[31m");

printf("HTTP/1.1 %i %s", code, phrase);

printf("\033[39m\n");

return true;

}

/\*\*

\* Loads file into message-body.

\*/

ssize\_t load(void)

{

// ensure file is open

if (file == NULL)

{

return -1;

}

// ensure body isn't already loaded

if (body != NULL)

{

return -1;

}

// buffer for octets

octet buffer[OCTETS];

// read file

ssize\_t size = 0;

while (true)

{

// try to read a buffer's worth of octets

ssize\_t octets = fread(buffer, sizeof(octet), OCTETS, file);

// check for error

if (ferror(file) != 0)

{

if (body != NULL)

{

free(body);

body = NULL;

}

return -1;

}

// if octets were read, append to body

if (octets > 0)

{

body = realloc(body, size + octets);

if (body == NULL)

{

return -1;

}

memcpy(body + size, buffer, octets);

size += octets;

}

// check for EOF

if (feof(file) != 0)

{

break;

}

}

return size;

}

/\*\*

\* Handles signals.

\*/

void handler(int signal)

{

// control-c

if (signal == SIGINT)

{

// ensure this isn't considered an error

// (as might otherwise happen after a recent 404)

errno = 0;

// announce stop

printf("\033[33m");

printf("Stopping server\n");

printf("\033[39m");

// stop server

stop();

}

}

/\*\*

\* Returns MIME type for supported extensions, else NULL.

\*/

const char\* lookup(const char\* extension)

{

// TODO

return NULL;

}

/\*\*

\* Parses an HTTP request.

\*/

ssize\_t parse(void)

{

// ensure client's socket is open

if (cfd == -1)

{

return -1;

}

// ensure request isn't already parsed

if (request != NULL)

{

return -1;

}

// buffer for octets

octet buffer[OCTETS];

// parse request

ssize\_t length = 0;

while (true)

{

// read from socket

ssize\_t octets = read(cfd, buffer, sizeof(octet) \* OCTETS);

if (octets == -1)

{

error(500);

return -1;

}

// if octets have been read, remember new length

if (octets > 0)

{

request = realloc(request, length + octets);

if (request == NULL)

{

return -1;

}

memcpy(request + length, buffer, octets);

length += octets;

}

// else if nothing's been read, socket's been closed

else

{

return -1;

}

// search for CRLF CRLF

int offset = (length - octets < 3) ? length - octets : 3;

char\* haystack = request + length - octets - offset;

char\* needle = memmem(haystack, request + length - haystack, "\r\n\r\n", 4);

if (needle != NULL)

{

// trim to one CRLF and null-terminate

length = needle - request + 2 + 1;

request = realloc(request, length);

if (request == NULL)

{

return -1;

}

request[length - 1] = '\0';

break;

}

// if buffer's full and we still haven't found CRLF CRLF,

// then request is too large

if (length - 1 >= LimitRequestLine + LimitRequestFields \* LimitRequestFieldSize)

{

error(413);

return -1;

}

}

return length;

}

/\*\*

\* Resets server's state, deallocating any resources.

\*/

void reset(void)

{

// free response's body

if (body != NULL)

{

free(body);

body = NULL;

}

// close file

if (file != NULL)

{

fclose(file);

file = NULL;

}

// free request

if (request != NULL)

{

free(request);

request = NULL;

}

// close client's socket

if (cfd != -1)

{

close(cfd);

cfd = -1;

}

}

/\*\*

\* Starts server.

\*/

void start(short port, const char\* path)

{

// path to server's root

root = realpath(path, NULL);

if (root == NULL)

{

stop();

}

// ensure root exists

if (access(root, F\_OK) == -1)

{

stop();

}

// ensure root is executable

if (access(root, X\_OK) == -1)

{

stop();

}

// announce root

printf("\033[33m");

printf("Using %s for server's root", root);

printf("\033[39m\n");

// create a socket

sfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sfd == -1)

{

stop();

}

// allow reuse of address (to avoid "Address already in use")

int optval = 1;

setsockopt(sfd, SOL\_SOCKET, SO\_REUSEADDR, &optval, sizeof(optval));

// assign name to socket

struct sockaddr\_in serv\_addr;

memset(&serv\_addr, 0, sizeof(serv\_addr));

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(port);

serv\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

if (bind(sfd, (struct sockaddr\*) &serv\_addr, sizeof(serv\_addr)) == -1)

{

stop();

}

// listen for connections

if (listen(sfd, SOMAXCONN) == -1)

{

stop();

}

// announce port in use

struct sockaddr\_in addr;

socklen\_t addrlen = sizeof(addr);

if (getsockname(sfd, (struct sockaddr\*) &addr, &addrlen) == -1)

{

stop();

}

printf("\033[33m");

printf("Listening on port %i", ntohs(addr.sin\_port));

printf("\033[39m\n");

}

/\*\*

\* Stop server, deallocating any resources.

\*/

void stop(void)

{

// preserve errno across this function's library calls

int errsv = errno;

// reset server's state

reset();

// free root, which was allocated by realpath

if (root != NULL)

{

free(root);

}

// close server socket

if (sfd != -1)

{

close(sfd);

}

// terminate process

if (errsv == 0)

{

// success

exit(0);

}

else

{

// announce error

printf("\033[33m");

printf("%s", strerror(errsv));

printf("\033[39m\n");

// failure

exit(1);

}

}

**Приклад виконання програми**

****

***Висновок:*** завдяки набутим знанням з програмування, ми навчились реалізовувати складні програми.