```
// ------------------------------------------------------------------------
----------
//  File: webserver.c
// Author: Carter Shean  Login: cshea892  Class: CpS 320
//  Desc:   This program  expands on the Echo server program
//             handling client requests for files and returning either
//             the contents or an error message
// ------------------------------------------------------------------------
---------

/* Echo Server: an example usage of EzNet
 * (c) 2016, Bob Jones University
 */

#include "eznet.h"      // Custom networking library
#include "utils.h"

//GLOBAL: variable for current number of threads running
int currentNumThreads = 0;

//GLOBAL mutex to protect the NumThreads
pthread_mutex_t num_lock;

// GLOBAL: settings structure instance
struct settings {
    const char *bindhost;   // Hostname/IP address to bind/listen on
    const char *bindport;   // Portnumber (as a string) to bind/listen on
    int numthreads;
} g_settings = {
    .bindhost = "localhost",    // Default: listen only on localhost interface
    .bindport = "5000",         // Default: listen on TCP port 5000
    .numthreads = 5,
};

// Parse commandline options and sets g_settings accordingly.
// Returns 0 on success, -1 on false...
int parse_options(int argc, char * const argv[]) {
    int ret = -1;
    char op;
    while ((op = getopt(argc, argv, "r:h:p:w:")) > -1) {
        switch (op) {
            case 'h':
                g_settings.bindhost = optarg;
                break;
            case 'p':
                g_settings.bindport = optarg;
                break;
            case 'r':
                //set the path equal to the one specified
                chdir(optarg);
                break;
            case 'w':
                //specify the number of threads
                g_settings.numthreads = atoi(optarg);
                break;
            default:
                // Unexpected argument--abort parsing
                goto cleanup;
        }
    }

    ret = 0;
    cleanup:
    return ret;
}

// GLOBAL: flag indicating when to shut down server
volatile bool server_running = false;
```



```
// SIGINT handler that detects Ctrl-C and sets the "stop serving" flag
void sigint_handler(int signum) {
    blog("Ctrl-C (SIGINT) detected; shutting down...");
    server_running = false;
}
//this function checks the error number of each system call for success or failu
re
//and takes appropriate action on failure
/*void checkForError(FILE * stream, int returnValue){
        if (returnValue
}*/

//takes the path string and stream to print from and compares the path with vari
ous
//substrings to see which type of file was opened
void outputBodyType(char * path, FILE * stream){
        //if statements to check the file type (not the most efficient, but effe
ctive nonetheless)
        if (strstr(path, ".txt") != NULL) {
                fprintf(stream, "Content-type: text/plain\n\n");
        } else if (strstr(path, ".html") != NULL || strstr(path, ".htm") != NULL)
{
                fprintf(stream, "Content-type: html/htm\n\n");
        } else if (strstr(path, ".png") != NULL) {
                fprintf(stream, "Content-type: image/png\n\n");
        } else if (strstr(path, ".jpg") != NULL || strstr(path, ".jpeg") != NULL) {
                fprintf(stream, "Content-type: image/jpeg\n\n");
        } else if (strstr(path, ".gif") != NULL) {
                fprintf(stream, "Content-type: image/gif\n\n");
        } else {
                fprintf(stream, "Content-type: application/octet-stream\n\n");
        }

}

//This method takes an errorNum and stream, and based
//on the supplied errorNum, outputs the desired HTTP response with
//to the supplied stream. The function returns nothing
void handleError(int errorNum, FILE *stream){
        char * errorString = NULL;
        //switch statement to handle the supplied error number
        switch (errorNum){
                case (404) :
                        fprintf(stream, "HTTP/1.0 404 ERROR\n");
                        errorString = ("\nFile not found\n");
                        break;
                case (403):
                        fprintf(stream, "HTTP/1.0 403 ERROR\n");
                        errorString = ("\nForbidden\n");
                        break;
                case (501) :
                        fprintf(stream, "HTTP/1.0 501 ERROR\n");
                        errorString = ("\nNot Implemented\n");
                        break;
                case (400) :
                        fprintf(stream, "HTTP/1.0 400 ERROR\n");
                        errorString = ("\nBad Request\n");
                break;
                default:
                        fprintf(stream, "HTTP/1.0 500 ERROR\n");
                        errorString = "\nInternal Server Error\n";
                        break;
        }
        //print out the headers and body
        fprintf(stream, "Content-type: text/plain\n");
        fprintf(stream, "%s", errorString);
}

//check to see if the path is valid by comparing the requested path to the curre
```

```
nt directory
//if the path contains the current directory and is a file, proceed as usual
//otherwise, check to see if the file is openable at all, and if it is, return 4
03 error
//if the file is not found, return 404.
int openFile(char * path, FILE * stream){
    FILE * fp = NULL;
    struct stat s;
    int success;
   char cwd[1024];
   char expandedPathName[400];
   realpath(path, expandedPathName);
   //check to see if stat worked correctly
   if (stat(path,&s) != 0) {
           perror("stat() error:");
           handleError(404, stream);
           success = 0;
          return success;
   }
//get the current working directories name, and if it's not NULL, proceed
   if (getcwd(cwd, sizeof(cwd)) != NULL) { //use stat to check to make sure the
file is not a folder
           //check to see if the requested path is contained in the current work
ing directory
          if( strstr(expandedPathName, cwd) != NULL && s.st_mode & S_IFREG) {
                fp = fopen(path, "rb");
                        //open the path for reading and if we get an error, hand
le it
                        if (fp == NULL) {
                                handleError(404, stream);
                                success = 0;
                                return success;
                        //otherwise, handle the file normally
                        } else {
                                fprintf(stream, "HTTP/1.0 200 OK\n\n");
                                outputBodyType(path, stream);
                                int c;
                                while ((c = getc(fp)) != EOF){
                                        fprintf(stream, "%c", c);
                                }
                                fclose(fp);
                                printf("\n");
                                success = 1;
                                return success;
                        }
          //if the path is not in the directory, the program checks to see if it
 does in fact exist
          } else {
                  //return 403 if the file exists
                  fp = fopen(path, "r");
                  if (fp != NULL) {
                          handleError(403, stream);
                          success = 0;
                          return success;
                 //if the file couldn't be found, return 404
                  } else {
                          handleError(404, stream);
                          success = 0;
                          return success;
                  }
          }
   //handle an error with getcwd
  } else {

       handleError(500, stream);
       success = 0;
       return success;
   }
```



```
}



// Connection handling logic: reads/echos lines of text until error/EOF,
// then tears down connection.
void * handle_client(void * client1) {
    FILE *stream = NULL;
    struct client_info * client = (struct client_info *)client1;
    // Wrap the socket file descriptor in a read/write FILE stream
    // so we can use tasty stdio functions like getline(3)
    // [dup(2) the file descriptor so that we don't double-close;
    // fclose(3) will close the underlying file descriptor,
    // and so will destroy_client()]
    if ((stream = fdopen(client->fd, "r+"))== NULL) {
        perror("unable to wrap socket");
        goto cleanup;
   }

    // set up variables to take input from the buffer
    char *line = NULL;
    size_t len = 0u;
    ssize_t recd;
    ssize_t failureCheck;
    char input [251];
    char  verb [251];
    char  path [251];
    char  protocol [251];

    //use fgets to get the input from the stream, checking if the call worked
       if (fgets (input , 400 , stream) != NULL){

             failureCheck = sscanf(input, "%s %s %s", verb, path, protocol);
           //if scanf didn't find three inputs or reached EOF
            if (failureCheck != 3 || failureCheck == EOF) {
                handleError(400, stream);
                goto cleanup;
            }
    //if the fgets failed, handle the error and goto cleanup (my code is not bei
ng reached here, but in my tests the server did not crash)
    } else {
            handleError(400, stream);
            goto cleanup;
    }
    //compare the verb to GET, and if the verb is not equal, return a 501 error
    if (strcmp(verb, "GET") != 0) {
            handleError(501, stream);
            goto cleanup;
    }
    //do file processing inside the openfile method
    int success = openFile(path, stream);
    //check for file success, and if none exists, return
    if (success == 0) {
            goto cleanup;
    }

     //read and discard the rest of what the user enters
    while ((recd = getline(&line, &len, stream)) > 0 ) {
        printf("\tReceived %zd byte line\n", recd);
            if (recd == 2) {
                    goto cleanup;
            }
    }


cleanup:
    // Shutdown this client
    if (stream) fclose(stream);
```

```
    destroy_client_info(client);
    //decrement the global number of clients
    pthread_mutex_lock(&num_lock);
    --currentNumThreads;
    pthread_mutex_unlock(&num_lock);
    free(line);
    printf("\tSession ended.\n");
    blog("%d client(s) connected", currentNumThreads);
    return client1;
}

int main(int argc, char **argv) {
    int ret = 1;

    // Network server/client context
    int server_sock = -1;

    //initialize mutex
    pthread_mutex_init(&num_lock, NULL);
    // Handle our options
    if (parse_options(argc, argv)) {
        printf("usage: %s [-r ROOTDIRECTORY] [-p PORT] [-h HOSTNAME/IP] [-w NUMTHREADS]\
n", argv[0]);
        goto cleanup;
    }

    // Install signal handler for SIGINT
    struct sigaction sa_int = {
        .sa_handler = sigint_handler
    };
    if (sigaction(SIGINT, &sa_int, NULL)) {
        LOG_ERROR("sigaction(SIGINT, ...) -> '%s'", strerror(errno));
        goto cleanup;
    }

    // Start listening on a given port number
    server_sock = create_tcp_server(g_settings.bindhost, g_settings.bindport);
    if (server_sock < 0) {
        perror("unable to create socket");
        goto cleanup;
    }
    blog("Bound and listening on %s:%s", g_settings.bindhost, g_settings.bindport);

    server_running = true;
    while (server_running) {
        struct client_info client;

        // Wait for a connection on that socket
        if (wait_for_client(server_sock, &client)) {
            // Check to make sure our "failure" wasn't due to
            // a signal interrupting our accept(2) call; if
            // it was  "real" error, report it, but keep serving.
            if (errno != EINTR) { perror("unable to accept connection"); }
        } else {
             //create a thread and check to see if the max number of threads is
being used
             if (currentNumThreads < g_settings.numthreads) {

                    blog("connection from %s:%d", client.ip, client.port);

                    pthread_t thread1;

                    pthread_create(&thread1, NULL, handle_client, &client);
                     //increment the total number of threads running
                    pthread_mutex_lock(&num_lock);
                    ++currentNumThreads;
                    pthread_mutex_unlock(&num_lock);
                    blog("%d client(s) connected", currentNumThreads);
                    sleep(3);
```



```
             //otherwise, print the max connections reached  and don't allow th
e client to connect
             } else {
                    perror("Max number of connections reached");
             }
        }
    }
    ret = 0;

cleanup:
    if (server_sock >= 0) close(server_sock);
    return ret;
}
```

```
// ------------------------------------------------------------------------
----------
//  File: utils.c
// Author: Carter Shean  Login: cshea892  Class: CpS 320
//  Desc:   This file contains the blog method taken from webserver.c
// ------------------------------------------------------------------------
--------

#include "utils.h"

// Generic log-to-stdout logging routine
// Message format: "timestamp:pid:user-defined-message"
void blog(const char *fmt, ...) {
    // Convert user format string and variadic args into a fixed string buffer
    char user_msg_buff[256];
    va_list vargs;
    va_start(vargs, fmt);
    vsnprintf(user_msg_buff, sizeof(user_msg_buff), fmt, vargs);
    va_end(vargs);

    // Get the current time as a string
    time_t t = time(NULL);
    struct tm *tp = localtime(&t);
    char timestamp[64];
    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", tp);

    // Print said string to STDOUT prefixed by our timestamp and pid indicators
    printf("%s:%d:%s\n", timestamp, getpid(), user_msg_buff);
}
```

```
// ------------------------------------------------------------------------
----------
//  File: utils.h
// Author: Carter Shean  Login: cshea892  Class: CpS 320
//  Desc:   This file contains the #include statements and prototpye
//             for the blog method to be used in the webserver.c file
// ------------------------------------------------------------------------
---------

#ifndef UTILS_H
#define UTILS_H

#include <stdbool.h>    // For access to C99 "bool" type
#include <stdio.h>      // Standard I/O functions
#include <stdlib.h>     // Other standard library functions
#include <string.h>     // Standard string functions
#include <errno.h>      // Global errno variable
#include <sys/stat.h>

#include <stdarg.h>     // Variadic argument lists (for blog function)
#include <time.h>       // Time/date formatting function (for blog function)

#include <pthread.h>
#include <unistd.h>     // Standard system calls
#include <signal.h>     // Signal handling system calls (sigaction(2))


// Generic log-to-stdout logging routine
void blog(const char *fmt, ...);

#endif
```

```
// ------------------------------------------------------------------------
----------
//  File: utils.c
// Author: Carter Shean  Login: cshea892  Class: CpS 320
//  Desc:   This file contains the blog method taken from webserver.c
// ------------------------------------------------------------------------
--------

#include "utils.h"

// Generic log-to-stdout logging routine
// Message format: "timestamp:pid:user-defined-message"
void blog(const char *fmt, ...) {
    // Convert user format string and variadic args into a fixed string buffer
    char user_msg_buff[256];
    va_list vargs;
    va_start(vargs, fmt);
    vsnprintf(user_msg_buff, sizeof(user_msg_buff), fmt, vargs);
    va_end(vargs);

    // Get the current time as a string
    time_t t = time(NULL);
    struct tm *tp = localtime(&t);
    char timestamp[64];
    strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S", tp);

    // Print said string to STDOUT prefixed by our timestamp and pid indicators
    printf("%s:%d:%s\n", timestamp, getpid(), user_msg_buff);
}
```

```
# Remove the "-DSHOW_LOG_ERROR" to eliminate the
# verbose error logging messages generated by EZNet
CFLAGS=-g -std=gnu11 -Wall -Werror -DSHOW_LOG_ERROR

all: webserver

webserver: webserver.c eznet.o utils.o
	$(CC) $(CFLAGS) -owebserver webserver.c eznet.o utils.o -lpthread

eznet.o: eznet.c eznet.h
	$(CC) -c $(CFLAGS) eznet.c

utils.o: utils.c utils.h
	$(CC) -c $(CFLAGS) utils.c

clean:
	rm webserver *.o
```