```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <dirent.h>
#include <libgen.h>
#include <pthread.h>

#define PORT "8000"        //The port users will be connecting to.
#define BACKLOG 10          //The maximum number of pending connections queue will hold.
const char **global_argv;   //Global list of command-line arguments.  Need path name, which is argv[1] for handleConnection()
function.

/*
 *ICSI333. System Fundamentals,
 *Spring 2022,
 *TA Sourav,
 *Kiran Aziz,
 *ID: 001440162
 *
 *Project 4 Partner: Courtney Ng
 *
 *Work done by Courtney: Server implementation, client testing, and documentation/research.
 *Work done by Kiran: Server implementation, file handling, thread creation, and commenting.
 *Testing done by both partners on both personal machines using localhosts on port 8000 with telnet.
 */

/*
 *The sigchld_handler function handles child processes when it stops
 *or terminates.  This prevents the accumulation of zombie children
 *since the parent uses wait to free their resources.
 * @param s: Integer representing signal number.
 */
void sigchld_handler(int s){
    (void)s; //Unused variable warning.

    //Saves errno in the case that it changes since the
    //waitpid() might overwrite errno.
    int saved_errno = errno;
    while(waitpid(-1, NULL, WNOHANG) > 0); //Non blocking check with WNOHANG flag.
    errno = saved_errno;
}


/*
 *The get_in_addr function takes in a socket address structure and
 *returns a void pointer, which is an IPv4 or IPv6 address sized value.
 *The function creates a list of addresses available for the specified
 *service name and location.
 * @param sa: struct sockaddr that is the abstract address of the socket.
 */
void *get_in_addr(struct sockaddr *sa){
    //Checks address family.
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }
    return &(((struct sockaddr_in6*)sa)->sin6_addr);
```

```c
}


/*
 *The handleConnection function allows server to accept
 *filenames and reads the file's size and content back to the client. Also
 *acts as the thread function when creating new threads.
 * @param p_client_socket: Pointer to the client socket.
 */
void *handleConnection(void *p_client_socket){
    int new_fd = *((int*)p_client_socket); //Assigns file descriptor to new_fd.
    free(p_client_socket); //Pointer is not needed anymore so it is freed.

    char buffer[100];       //Character array buffer to read message from client.
    char filenameCopy[256]; //Duplicates path to source file to extract file name.
    char *subString;        //Filename from client's message.

    //Buffer reads in client's message.
    if(recv(new_fd, buffer, sizeof(buffer), 0) == -1){
        perror("receive");
    }


    strcpy(filenameCopy, buffer); //Copies content from buffer.

    //File name path is between two whitespaces.
    subString = strtok(filenameCopy," "); //Finds the first space.
    subString = strtok(NULL," ");   //Finds the second space.

    //Constructs full path to the source file using path and filename.
    char fullPath[strlen(subString) + 50];
    strcpy(fullPath, global_argv[1]); //Path name is from command-line argument.
    strcat(fullPath, subString); //Filename is from client's message.

 //Describes success or failure for system call open()
 //Returns -1 if there is an error and writes an error message.
    int sourceFD = open(fullPath, O_RDONLY);
    if (sourceFD < 0) {
        fprintf(stderr, "Error: Cannot open or find source file.\n");
        perror("open");

        //If file is not found, 404 message is sent out to the client.
        char *errorMessage = "\nHTTP/1.0 404 File Not Found\r\n";
        send(new_fd, errorMessage, strlen(errorMessage), 0);

        close(sourceFD); //Close source file.
        close(new_fd); //Close socket.
        return NULL;
    }


    //Reads data from source file to the buffer.
    //Using BUFSIZ macro to determine the size of the buffer.
    char buf[BUFSIZ];
    int numOfBytes;

    //Determines size of file, which is the number of bytes.
    struct stat st;
    stat(fullPath, &st);
    int size = st.st_size;

    //If file is found, 200 message is sent out to the client.
    char *response = "\nHTTP/1.0 200 OK\r\nContent-Length:";
    write(new_fd, response, strlen(response));

    //Convert file's number of bytes to a string value.
    char contentLengthString[10];
    snprintf(contentLengthString, 10, "%d", size);

    //Send length of source file to the client.
    write(new_fd, contentLengthString, strlen(contentLengthString));
    write(new_fd,"\r\n\r\n", 4);

    //Writes data from the buffer to the clientside.
    //System call write() returns -1 if it fails.
    while((numOfBytes = read(sourceFD, buf, (size_t) BUFSIZ)) > 0 ){
        if(write(new_fd, &buf, numOfBytes) == -1){
            perror("write");
        }
    }


    write(new_fd,"\r\n", 2);
    close(sourceFD); //Closes source file.
    close(new_fd); //Closes socket.
    return NULL;
}



/*
 *The main function runs a simple web server that connects
 *to a port, listens for requests, and sends the appropriate file
 *to the requestor. The web server implements the GET request.
 *The web server also uses TCP connection to receive a request.
 *
 * @param argc: Integer holding the number of command line arguments.
 * @param argv: Array of string values, with each value being an command line argument
 *              that is passed onto the main function.
 */
int main(int argc, char const* argv[]){
    global_argv = argv; //Refers to global list of command-line arguments.
    int sockfd, new_fd; //Listen is on sock_fd, while the new connection is on new_fd.

    struct addrinfo hints, *servinfo, *p; //addrinfo structures that hold information about protocol
                                          //family, socket type, and protocol.
    struct sockaddr_storage their_addr; //Connector's address information.
    socklen_t sin_size;                 //Unsigned integer to indicate size of address.
    struct sigaction sa;                //A "disposition", or a way to express what to do when the
                                        //given signal is received.
    int yes = 1;    //Integer used to set flag SO_REUSEADDR to true/yes under socket options.

    char s[INET6_ADDRSTRLEN]; //Character array to hold address.
    int rv;                   //Integer used to represent status code, either success or failure, of getaddrinfo() function;

    //Loads up address structs with getaddrinfo().
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;     //No specification for type of address (IPv4 or IPv6).
    hints.ai_socktype = SOCK_STREAM; //SOCK_STREAM indicates TCP connection, which is a stream of data.
    hints.ai_flags = AI_PASSIVE;     //Fills in my IP.


    //If the server does not take a path as a command-line
    //argument, then the program will fail with a message.
    if(argc != 2){
        fprintf(stderr, "Arguments error: Need a directory path for web server to serve files.\n");
        exit(1);
    }

    //Gets my available interfaces.
    //Using address "127.0.0.1" for localhost.
    //Using port 8000.
    //hints is an addrinfo structure that specifies what we want.
```

```c
//servinfo is a pointer to a pointer to a linked list of addrinfo structures.
if ((rv = getaddrinfo("127.0.0.1", PORT, &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
}

//Loops through all the results and binds to the first we can.
for(p = servinfo; p != NULL; p = p->ai_next) {
    //Creates socket.
    if ((sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    }

    //Sets the current value for a socket option associated with a socket
    //of any type, in any state.
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }

    //Binds previously made socket to the port we want.
    //In this case, the port is 8000.
    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("server: bind");
        continue;
    }

    break;
}


freeaddrinfo(servinfo); //All done using this structure.

//If socket fails to bind, error message is thrown and
//program ends.
if (p == NULL) {
    fprintf(stderr, "server: failed to bind\n");
    exit(1);
}

//Marks a connection-mode socket as accepting connections.
//Backlog: provides hint to implementation on queue size, or
//         how big to make the queue.
if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
}

//Constructing the new disposition.
sa.sa_handler = sigchld_handler; //Reaps all dead processes.
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;

//sigaction() system call changes the action
//taken by a process on receipt of a specific signal.
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    perror("sigaction");
    exit(1);
}

printf("server: waiting for connections...\n");

//Infinite loop that accepts new connections
while(1){
        sin_size = sizeof their_addr; //Sets size of address.
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size); //Accepts new connection.

        if (new_fd == -1) {
            perror("accept");
        }

        //Converts Internet address in binary to text format.
        inet_ntop(their_addr.ss_family, get_in_addr((struct sockaddr *)&their_addr), s, sizeof s);
        //Prints connection message.
        printf("server: got connection from %s\n", s);

        //Handles each request using threads with pthreads.
        int *pclient = malloc(sizeof(int));
        *pclient = new_fd;

        pthread_t t; //Identifier for a thread.
        pthread_create(&t, NULL, handleConnection, pclient); //Creates a new thread.

    }
    return 0;
}
```