```c
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <netdb.h>
#include <dirent.h>

#include <pthread.h>

void* process_control_connection(void *sock);
int process_request(char *buffer, int new_socket, int bytes_received, int *s
int sign_in_thread(char *username);
int pwd(char *cwd, char *data, size_t cwd_size);
int prepare_socket(int port, struct addrinfo *results);
void check_status(int status, const char *error);
int begin_connection(int listening_socket, void *on_create_function);
unsigned long getFileLength(FILE *fp);

// Commands
const char *USER = "USER";
const char *QUIT = "QUIT";
const char *PWD = "PWD";
const char *CWD = "CWD";
const char *PORT = "PORT";
const char *NLST = "NLST";
const char *RETR = "RETR";
const char *TYPE = "TYPE";
const char *SYST = "SYST";
const char *FEAT = "FEAT";
const char *PASV = "PASV";

int NUM_THREADS = 0;
int MAIN_PORT = 5000;
int CURRENT_CONNECTION_PORT = 5000;
const char *ROOT = "/var/folders/r6/mzb0s9jd1639123lkcsv4mf00000gn/T/server"
```

```
40
41 int MAX_NUM_ARGS = 2;
42 int MAX_COMMAND_LENGTH = 50;
43
44 int main(int argc, char *argv[]){
45
46     // Set root directory of server
47     int chdir_status = chdir(ROOT);
48     check_status(chdir_status, "chdir");
49 //     int chroot_status = chroot(ROOT);
50 //     check_status(chroot_status, "chroot");
51
52     // For storing results from creating socket
53     struct addrinfo *results;
54
55     // Create and bind socket to specified port
56     int listening_socket = prepare_socket(MAIN_PORT, results);
57
58     // Listen for connections
59     pid_t pID;
60     int new_socket;
61     while (1) {
62
63         int listen_status = listen(listening_socket, 10);
64         check_status(listen_status, "listen");
65
66         // Accept clients, spawn new thread for each connection
67         new_socket = begin_connection(listening_socket, &process_control_con
68     }
69     freeaddrinfo(results);
70     close(new_socket);
71     close(listening_socket);
72     return 0;
73 }
74
75 // Executed by new thread when server accepts new connection
76 // Receives client requests, calls process_request to parse request and send
77 void *process_control_connection(void *sock) {
78
```

```
79      // Bookkeeping for this thread
80      int signed_in = 0;
81      int data_port = CURRENT_CONNECTION_PORT;
82      int listening_data_socket = 0;
83      int accept_data_socket = 0;
84
85      // Cast void* as int*
86      int *new_socket_ptr = (int *) sock;
87
88      // Get int from int*
89      int new_socket = *new_socket_ptr;
90
91      // Prepare to send and receive data
92      int bufsize = 1024;
93      int *total_bytes_sent = malloc(sizeof(int));
94      char *buffer = malloc(bufsize);
95      int bytes_received;
96
97      // Send message initializing connection
98      char *initial_message;
99      initial_message = "220 Sophia's FTP server (Version 0.0) ready.\n";
100     int bytes_sent = send(new_socket, initial_message, strlen(initial_messag
101     *total_bytes_sent = bytes_sent;
102
103     while (1) {
104         memset(buffer, 0, bufsize);
105         bytes_received = recv(new_socket, buffer, bufsize, 0);
106         check_status(bytes_received, "receive");
107
108         if (bytes_received > 0) {
109             bytes_sent  = process_request(buffer, new_socket, bytes_received
110             check_status(bytes_sent, "send");
111             *total_bytes_sent += bytes_sent;
112         }
113         else { // bytes_recv == 0
114             close(new_socket);
115             printf("Client closed connection.\n");
116             break;
117         }
```

```
118     }
119     printf("Thread closed. %d total bytes sent. Threads still active: %d.\n"
120     free(buffer);
121     free(total_bytes_sent);
122     pthread_exit((void *) total_bytes_sent);
123 }
124 /*    END PROCESS CONTROL CONNECTION    */
125 
126 // Executed by new thread when server accepts new connection
127 // Receives client requests, calls process_request to parse request and send
128 void *process_data_connection(void *sock) {
129 
130     // Cast void* as int*
131     int *listening_data_socket_ptr = (int *) sock;
132 
133     // Get int from int*
134     int listening_data_socket = *listening_data_socket_ptr;
135 
136     int new_socket;
137     char *initial_message;
138 //     while (1) {
139 
140         int listen_status = listen(listening_data_socket, 10);
141         check_status(listen_status, "listen");
142 
143         // Accept clients
144         struct sockaddr_storage client;
145         socklen_t addr_size = sizeof(client);
146         printf("\nin loop about to accept?\n");
147 
148         new_socket = accept(listening_data_socket, (struct sockaddr *) &clie
149         check_status(new_socket, "accept");
150         printf("\naccepted, connection began\n");
151 
152         // Send message initializing connection
153 //         initial_message = "220 Sophia's FTP server DATA CONNECTION (Versi
154 //         int bytes_sent = send(new_socket, initial_message, strlen(initial
155 //         check_status(bytes_sent, "send");
156 //
```

```
157 //          close(listening_data_socket);
158 //          close(new_socket);
159     pthread_exit((void *) initial_message);
160 }
161 /*    END PROCESS DATA CONNECTION    */
162 
163 // Handles FTP client requests and sends appropriate responses
164 int process_request(char *buffer, int new_socket, int bytes_received, int *s
165     size_t data_size = 1024*sizeof(char);
166     char *data = malloc(data_size);
167     memset(data, '\0', data_size);
168 
169     // TODO make some of these globals
170     int bytes_sent;
171 
172     int already_sent = 0;
173 
174     // Assuming all commands contain less than two words (TODO: otherwise, e
175     MAX_NUM_ARGS = 2;
176     char parsed[MAX_NUM_ARGS][MAX_COMMAND_LENGTH];
177 
178     printf("\n-----------------------------------------\n");
179     printf("\nServer received: %s (%i bytes)\n", buffer, bytes_received);
180 
181     // TODO -- GETCHAR OF BUFFER???
182     // Parse buffer, splitting on spaces, tabs, nl, cr
183     int j = 0;
184     char * pch;
185     pch = strtok(buffer," \t\n\r");
186     while (pch != NULL && j < MAX_NUM_ARGS) {
187         memset(parsed[j], '\0', MAX_COMMAND_LENGTH);
188         snprintf(parsed[j], MAX_COMMAND_LENGTH, "%s", pch);
189         pch = strtok(NULL, " \t\n\r");
190         j++;
191     }
192 
193     int z = 0;
194     while(z < MAX_NUM_ARGS) {
195         printf("command %zd is %s\n", z, parsed[z]);
```

```
196         int a = 0;
197         for(a = 0; a < MAX_COMMAND_LENGTH; a++) {
198             if (parsed[z][a] == '\0') {
199                 printf("NULL-");
200             }
201             else {
202               printf("%c-", parsed[z][a]);
203             }
204         }
205         z++;
206         printf("\n");
207     }
208 
209     // Find appropriate response based on client's commands
210     if (strcmp(parsed[0], USER) == 0) {
211         *sign_in_status = sign_in_thread(parsed[1]);
212         if (*sign_in_status == 1) {
213             snprintf(data, data_size, "%s", "230 User signed in. Using binary
214         }
215         else {
216             snprintf(data, data_size, "%s", "530 Sign in failure.\n");
217         }
218     }
219     else if (strcmp(parsed[0], QUIT) == 0) {
220         NUM_THREADS--;
221         send(new_socket, "221 \n", 5, 0);
222         return 0;
223     }
224     else if (*sign_in_status == 1) {
225         if (strcmp(parsed[0], PWD) == 0) {
226 
227             char *cwd = malloc(data_size);
228             int pwd_status = pwd(cwd, data, data_size);
229             free(cwd);
230         }
231         else if (strcmp(parsed[0], CWD) == 0) {
232             ////////////////////////////////////////
233             // THREADS SHARE WORKING DIRECTORY, DAMMIT.
234             ////////////////////////////////////////
```

```
235
236             int chdir_status = chdir(parsed[1]);
237             if (chdir_status == 0) {
238                 snprintf(data, data_size, "%s", "250 CWD successful\n");
239             }
240             else {
241                 perror("CWD");
242                 snprintf(data, data_size, "%s", "550 CWD error\n");
243             }
244         }
245         else if (strcmp(parsed[0], NLST) == 0) {
246             // Thanks to http://stackoverflow.com/questions/4204666/how-to-l
247             int bytes_data_written = 0;
248             int bytes_response_code_written = 0;
249             DIR *d;
250             struct dirent *dir;
251             d = opendir(".");
252             char data_over_second_connection[data_size];
253             bytes_response_code_written = snprintf(data, data_size, "%s", "1
254             if (d && *accept_data_socket > 0) {
255                 while ((dir = readdir(d)) != NULL) {
256                     bytes_data_written = bytes_data_written + snprintf(data_
257                 }
258                 closedir(d);
259
260                 // Send directory information immediately over data socket, 
261                 bytes_sent += send(*accept_data_socket, data_over_second_con
262                 close(*accept_data_socket);
263                 *accept_data_socket = 0;
264
265                 snprintf(data + bytes_response_code_written, data_size, "226
266             }
267             else {
268                 snprintf(data, data_size, "%s", "550 NLST error\n");
269             }
270         }
271         else if (strcmp(parsed[0], RETR) == 0) {
272
273             FILE *fp;
```

```
274             fp = fopen(parsed[1], "rb");
275 
276             int bytes_data_written = 0;
277             int bytes_response_code_written = 0;
278 
279             char data_over_second_connection[data_size];
280             bytes_response_code_written = snprintf(data, data_size, "150 Ope
281             if ((fp != NULL) && *accept_data_socket > 0) {
282 
283 //                  char fileBuf[1000];
284 
285                 unsigned long fileLength = getFileLength(fp); // Only works 
286 
287                 char data_over_second_connection[fileLength + 1];
288 //                  strncpy(data_over_second_connection,"", 1);
289 
290                 rewind(fp);
291 //                  while (fgets(fileBuf, 1000, fp) != NULL) { // while we ha
292 //                      bytes_data_written = bytes_data_written + snprintf(da
293 //                  }
294 
295 
296                 unsigned char fileBuffer[fileLength + 1];
297 //                  buffer = (unsigned char *)malloc(fileLen);
298                 if (!fileBuffer) {
299                     fprintf(stderr, "Memory error!");
300                     fclose(fp);
301                     return 1;
302                 }
303 
304                 fread(fileBuffer, sizeof(unsigned char), fileLength, fp);
305 
306                 int i;
307                 for (i = 0; i < (fileLength + 1); i++) {
308                     bytes_sent += send(*accept_data_socket, (const void *) &
309                 }
310 //                  bytes_sent += send(*accept_data_socket, fileBuffer, fileL
311 
312 
```

```
313
314
315
316
317
318                 fclose(fp);
319 //                  printf("%s", data_over_second_connection);
320                 printf("%s", fileBuffer);
321
322
323
324 //                  int a;
325 //                  char *letter;
326 //                  for (a = 0; a < (strlen(data_over_second_connection) + 1
327 //                      putchar(data_over_second_connection[a]);
328 //                      letter = &data_over_second_connection[a];
329 //                      putchar(letter);
330 //                      bytes_sent += send(*accept_data_socket, letter, 1, 0
331 //                  }
332
333                 // Send directory information immediately over data socket,
334 //                  bytes_sent += send(*accept_data_socket, data_over_second
335                 close(*accept_data_socket);
336                 *accept_data_socket = 0;
337
338                 snprintf(data + bytes_response_code_written, data_size, "220
339
340             }
341             else {
342                 snprintf(data, data_size, "%s", "550 RETR error\n");
343             }
344         }
345         else if (strcmp(parsed[0], TYPE) == 0) {
346             if (strcmp(parsed[1], "I") == 0) {
347                 snprintf(data, data_size, "%s", "200 Using binary mode to tr
348             }
349             else {
350                 snprintf(data, data_size, "%s", "502 I only work with binary
351             }
```

```
352         }
353         else if (strcmp(parsed[0], SYST) == 0) {
354             snprintf(data, data_size, "%s", "215 MACOS Sophia's Server\n");
355         }
356         else if (strcmp(parsed[0], FEAT) == 0) {
357             snprintf(data, data_size, "%s", "211 end\n");
358         }
359         // 200 host and port address
360         else if (strcmp(parsed[0], PASV) == 0) {
361             CURRENT_CONNECTION_PORT++;
362             *data_port = CURRENT_CONNECTION_PORT;
363             struct addrinfo *data_results;
364             *listening_data_socket = prepare_socket(*data_port, data_results
365 
366 
367             // Client expects (a1,a2,a3,a4,p1,p2), where port = p1*256 + p2
368             int p1 = *data_port / 256;
369             int p2 = *data_port % 256;
370 
371             int listen_status = listen(*listening_data_socket, 10);
372             check_status(listen_status, "listen");
373 
374             // Accept clients
375             struct sockaddr_storage client;
376             socklen_t addr_size = sizeof(client);
377             printf("\nin loop about to accept?\n");
378 
379             snprintf(data, data_size, "%s =127,0,0,1,%i,%i\n", "227 Enterin
380             bytes_sent += send(new_socket, data, strlen(data), 0);
381             already_sent = 1;
382 
383             *accept_data_socket = accept(*listening_data_socket, (struct so
384             check_status(*accept_data_socket, "accept");
385             printf("\naccepted, DATA connection began\n");
386 
387             freeaddrinfo(data_results);
388         }
389         else if (strcmp(parsed[0], PORT) == 0) {
390             snprintf(data, data_size, "%s\n", "200 Port command ok");
```

```
391         }
392         else {
393             snprintf(data, data_size, "%s", "500 Syntax error, command unre
394         }
395     }
396     else {
397         snprintf(data, data_size, "%s", "530 User not logged in.\n");
398     }
399     if (already_sent == 0) {
400         bytes_sent = send(new_socket, data, strlen(data), 0);
401     }
402     printf("Data sent: %s\n", data);
403 
404     free(data);
405     return bytes_sent;
406 }
407 /*    END PROCESS REQUEST    */
408 
409 int sign_in_thread(char *username) {
410     if (strcmp(username, "anonymous") == 0) {
411         return 1;
412     }
413     else {
414         return -1;
415     }
416 }
417 
418 // Copies working directory into "data," along with appropriate success cod
419 int pwd(char *cwd, char *data, size_t cwd_size) {
420     if (getcwd(cwd, cwd_size) != NULL) {
421         snprintf(data, cwd_size, "257 \"%s\" \n", cwd);
422         return 1;
423     }
424     else {
425         perror("getcwd() error");
426         return -1;
427         exit(1);
428     }
429 }
```

```
430
431 int prepare_socket(int port, struct addrinfo *results) {
432
433     int listening_socket;
434
435     // Socket address information:
436     struct sockaddr_in address_in; // sockaddr_in contains IPv4 information
437     address_in.sin_family = AF_INET; // IPv4
438     address_in.sin_port = htons(port);
439     address_in.sin_addr.s_addr = INADDR_ANY; // expects 4-byte IP address (
440
441     struct addrinfo address; // addrinfo contains info about the socket
442     memset(&address, 0, sizeof(address));
443     address.ai_socktype = SOCK_STREAM;
444     address.ai_protocol = 0; // 0 = choose correct protocol for stream vs da
445     address.ai_addr = (struct sockaddr *) &address_in;
446     address.ai_flags = AI_PASSIVE; // fills in IP automatically
447
448     char port_str[5];
449     sprintf(port_str, "%d", port);
450
451     int status = getaddrinfo(NULL, port_str, &address, &results); // results
452     char message[50];
453     sprintf(message, "getaddrinfo error: %s\n", gai_strerror(status));
454     check_status(status, message);
455
456     // Create socket
457     listening_socket = socket(results->ai_family, results->ai_socktype, resu
458     if (listening_socket > 0) {
459         printf("Socket created, listening on port %s.\n", port_str);
460     }
461
462     // Allow reuse of port -- from Beej
463     int yes = 1;
464     int sockopt_status = setsockopt(listening_socket, SOL_SOCKET, SO_REUSEAD
465     check_status(sockopt_status, "setsockopt");
466
467     // Bind socket to address
468     // socket id, *sockaddr struct w address info, length (in bytes) of addr
```

```
469     int bind_status = bind(listening_socket, (struct sockaddr *) &address_i
470     check_status(bind_status, "bind");
471     printf("Binding socket...\n");
472 
473     return listening_socket;
474 }
475 
476 // Sets appropriate error message if status indicates error
477 void check_status(int status, const char *error) {
478     if (status < 0) {
479         char message[100];
480         sprintf(message, "server: %s", error);
481         perror(message);
482         exit(1);
483     }
484 }
485 
486 // Accept connection and spawn new thread
487 int begin_connection(int listening_socket, void *on_create_function) {
488     // Make a new socket specifically for sending/receiving data w this cli
489     int new_socket;
490 
491     // Info about incoming connection goes into sockaddr_storage struct
492     struct sockaddr_storage client;
493     socklen_t addr_size = sizeof(client);
494 
495     new_socket = accept(listening_socket, (struct sockaddr *) &client, &add
496     check_status(new_socket, "accept");
497 
498     if (new_socket > 0) {
499         pthread_t tid;
500         NUM_THREADS++;
501         pthread_create(&tid, NULL, on_create_function, &new_socket);
502         printf("\nA client has connected (socket %d), new thread created. T
503     }
504     return new_socket;
505 }
506 
507 unsigned long getFileLength(FILE *fp) {
```

```
508     fseek(fp, 0, SEEK_END);
509     unsigned long length = ftell(fp);
510     return length;
511 }
512
513
514
```