```
 1: // $Id: logstream.h,v 1.6 2021-02-24 15:29:53-08 - - $
 2:
 3: //
 4: // class logstream
 5: // replacement for initial cout so that each call to a logstream
 6: // will prefix the line of output with an identification string
 7: // and a process id.  Template functions must be in header files
 8: // and the others are trivial.
 9: //
10:
11: #ifndef __LOGSTREAM_H__
12: #define __LOGSTREAM_H__
13:
14: #include <cassert>
15: #include <iostream>
16: #include <string>
17: #include <vector>
18: using namespace std;
19:
20: #include <sys/types.h>
21: #include <unistd.h>
22:
23: class logstream {
24:    private:
25:       ostream& out_;
26:       string execname_;
27:    public:
28:
29:       // Constructor may or may not have the execname available.
30:       logstream (ostream& out, const string& execname = ""):
31:                  out_ (out), execname_ (execname) {
32:       }
33:
34:       // First line of main should set execname if logstream is global.
35:       void execname (const string& name) { execname_ = name; }
36:       string execname() { return execname_; }
37:
38:       // First call should be the logstream, not cout.
39:       // Then forward result to the standard ostream.
40:       template <typename T>
41:       ostream& operator<< (const T& obj) {
42:          assert (execname_.size() > 0);
43:          out_ << execname_ << "(" << getpid() << "): " << obj;
44:          return out_;
45:       }
46:
47: };
48:
49: #endif
50:
```

```
 1: // $Id: protocol.h,v 1.9 2020-12-12 22:09:29-08 - - $
 2:
 3: #ifndef __PROTOCOL__H__
 4: #define __PROTOCOL__H__
 5:
 6: #include <cstdint>
 7: using namespace std;
 8:
 9: #include "sockets.h"
10:
11: enum class cxi_command : uint8_t {
12:    ERROR = 0, EXIT, GET, HELP, LS, PUT, RM, FILEOUT, LSOUT, ACK, NAK,
13: };
14: constexpr size_t FILENAME_SIZE = 59;
15: constexpr size_t HEADER_SIZE = 64;
16: struct cxi_header {
17:    uint32_t nbytes {};
18:    cxi_command command {cxi_command::ERROR};
19:    char filename[FILENAME_SIZE] {};
20: };
21:
22: static_assert (sizeof (cxi_header) == HEADER_SIZE);
23:
24: void send_packet (base_socket& socket,
25:                   const void* buffer, size_t bufsize);
26:
27: void recv_packet (base_socket& socket, void* buffer, size_t bufsize);
28:
29: ostream& operator<< (ostream& out, const cxi_header& header);
30:
31: string get_cxi_server_host (const vector<string>& args, size_t index);
32:
33: in_port_t get_cxi_server_port (const vector<string>& args,
34:                                size_t index);
35:
36: #endif
37:
```

```
 1: // $Id: protocol.cpp,v 1.15 2021-03-01 16:08:48-08 - - $
 2:
 3: #include <iomanip>
 4: #include <iostream>
 5: #include <string>
 6: #include <unordered_map>
 7: using namespace std;
 8:
 9: #include "protocol.h"
10:
11: string to_string (cxi_command command) {
12:    switch (command) {
13:       case cxi_command::ERROR  : return "ERROR"  ;
14:       case cxi_command::EXIT   : return "EXIT"   ;
15:       case cxi_command::GET    : return "GET"    ;
16:       case cxi_command::HELP   : return "HELP"   ;
17:       case cxi_command::LS     : return "LS"     ;
18:       case cxi_command::PUT    : return "PUT"    ;
19:       case cxi_command::RM     : return "RM"     ;
20:       case cxi_command::FILEOUT: return "FILEOUT";
21:       case cxi_command::LSOUT  : return "LSOUT"  ;
22:       case cxi_command::ACK    : return "ACK"    ;
23:       case cxi_command::NAK    : return "NAK"    ;
24:       default                  : return "????"   ;
25:    };
26: }
27:
28:
29: void send_packet (base_socket& socket,
30:                   const void* buffer, size_t bufsize) {
31:    const char* bufptr = static_cast<const char*> (buffer);
32:    ssize_t ntosend = bufsize;
33:    do {
34:       ssize_t nbytes = socket.send (bufptr, ntosend);
35:       if (nbytes < 0) throw socket_sys_error (to_string (socket));
36:       bufptr += nbytes;
37:       ntosend -= nbytes;
38:    }while (ntosend > 0);
39: }
40:
41: void recv_packet (base_socket& socket, void* buffer, size_t bufsize) {
42:    char* bufptr = static_cast<char*> (buffer);
43:    ssize_t ntorecv = bufsize;
44:    do {
45:       ssize_t nbytes = socket.recv (bufptr, ntorecv);
46:       if (nbytes < 0) throw socket_sys_error (to_string (socket));
47:       if (nbytes == 0) throw socket_error (to_string (socket)
48:                                            + " is closed");
49:       bufptr += nbytes;
50:       ntorecv -= nbytes;
51:    }while (ntorecv > 0);
52: }
53:
```

```
54:
55: string to_hex32_string (uint32_t num) {
56:    ostringstream stream;
57:    stream << "0x" << hex << uppercase << setfill('0') << setw(8) << num;
58:    return stream.str();
59: }
60:
61: ostream& operator<< (ostream& out, const cxi_header& header) {
62:    constexpr size_t WARNING_NBYTES = 1<<20;
63:    uint32_t nbytes = htonl (header.nbytes);
64:    if (nbytes > WARNING_NBYTES) {
65:       out << "WARNING: Payload nbytes " << nbytes << " > "
66:           << WARNING_NBYTES << endl;
67:    }
68:    return out << "{" << to_hex32_string (header.nbytes) << ':'
69:               << header.nbytes << ':' << ntohl (header.nbytes) << ","
70:               << unsigned (header.command)
71:               << "(" << to_string (header.command) << "),\""
72:               << header.filename << "\"}";
73: }
74:
75: string get_cxi_server_host (const vector<string>& args, size_t index) {
76:    if (index < args.size()) return args[index];
77:    char* host = getenv ("CIX_SERVER_HOST");
78:    if (host != nullptr) return host;
79:    return "localhost";
80: }
81:
82: in_port_t get_cxi_server_port (const vector<string>& args,
83:                                size_t index) {
84:    string port = "-1";
85:    if (index < args.size()) port = args[index];
86:    else {
87:       char* envport = getenv ("CIX_SERVER_PORT");
88:       if (envport != nullptr) port = envport;
89:    }
90:    return stoi (port);
91: }
92:
```

```
 1: // $Id: sockets.h,v 1.2 2016-05-09 16:01:56-07 - - $
 2:
 3: #ifndef __SOCKET_H__
 4: #define __SOCKET_H__
 5:
 6: #include <cstring>
 7: #include <stdexcept>
 8: #include <string>
 9: #include <vector>
10: using namespace std;
11:
12: #include <arpa/inet.h>
13: #include <netdb.h>
14: #include <netinet/in.h>
15: #include <string>
16: #include <sys/socket.h>
17: #include <sys/types.h>
18: #include <sys/wait.h>
19: #include <unistd.h>
20:
21: //
22: // class base_socket:
23: // mostly protected and not used by applications
24: //
25:
26: class base_socket {
27:    private:
28:       static constexpr size_t MAXRECV = 0xFFFF;
29:       static constexpr int CLOSED_FD = -1;
30:       int socket_fd {CLOSED_FD};
31:       sockaddr_in socket_addr;
32:    protected:
33:       base_socket(); // only derived classes may construct
34:       base_socket (const base_socket&) = delete; // prevent copying
35:       base_socket& operator= (const base_socket&) = delete;
36:       ~base_socket();
37:       // server_socket initialization
38:       void create();
39:       void bind (const in_port_t port);
40:       void listen() const;
41:       void accept (base_socket&) const;
42:       // client_socket initialization
43:       void connect (const string host, const in_port_t port);
44:       // accepted_socket initialization
45:       void set_socket_fd (int fd);
46:    public:
47:       void close();
48:       ssize_t send (const void* buffer, size_t bufsize);
49:       ssize_t recv (void* buffer, size_t bufsize);
50:       void set_non_blocking (const bool);
51:       friend string to_string (const base_socket& sock);
52: };
53:
```

```
54:
55: //
56: // class accepted_socket
57: // used by server when a client connects
58: //
59:
60: class accepted_socket: public base_socket {
61:    public:
62:       accepted_socket() {}
63:       accepted_socket(int fd) { set_socket_fd (fd); }
64: };
65:
66: //
67: // class client_socket
68: // used by client application to connect to server
69: //
70:
71: class client_socket: public base_socket {
72:    public:
73:       client_socket (string host, in_port_t port);
74: };
75:
76: //
77: // class server_socket
78: // single use class by server application
79: //
80:
81: class server_socket: public base_socket {
82:    public:
83:       server_socket (in_port_t port);
84:       void accept (accepted_socket& sock) {
85:          base_socket::accept (sock);
86:       }
87: };
88:
```

```
 89:
 90: //
 91: // class socket_error
 92: // base class for throwing socket errors
 93: //
 94:
 95: class socket_error: public runtime_error {
 96:    public:
 97:       explicit socket_error (const string& what): runtime_error(what){}
 98: };
 99:
100: //
101: // class socket_sys_error
102: // subclass to record status of extern int errno variable
103: //
104:
105: class socket_sys_error: public socket_error {
106:    public:
107:       int sys_errno;
108:       explicit socket_sys_error (const string& what):
109:                socket_error(what + ": " + strerror (errno)),
110:                sys_errno(errno) {}
111: };
112:
113: //
114: // class socket_h_error
115: // subclass to record status of extern int h_errno variable
116: //
117:
118: class socket_h_error: public socket_error {
119:    public:
120:       int host_errno;
121:       explicit socket_h_error (const string& what):
122:                socket_error(what + ": " + hstrerror (h_errno)),
123:                host_errno(h_errno) {}
124: };
125:
```

```
126:
127: //
128: // class hostinfo
129: // information about a host given hostname or IPv4 address
130: //
131:
132: class hostinfo {
133:    public:
134:       const string hostname;
135:       const vector<string> aliases;
136:       const vector<in_addr> addresses;
137:       hostinfo (); // localhost
138:       hostinfo (hostent*);
139:       hostinfo (const string& hostname);
140:       hostinfo (const in_addr& ipv4_addr);
141:       friend string to_string (const hostinfo&);
142: };
143:
144: string localhost();
145: string to_string (const in_addr& ipv4_addr);
146:
147: #endif
148:
```

```
 1: // $Id: sockets.cpp,v 1.3 2019-05-08 11:36:22-07 - - $
 2:
 3: #include <cerrno>
 4: #include <cstring>
 5: #include <iostream>
 6: #include <sstream>
 7: #include <string>
 8: using namespace std;
 9:
10: #include <fcntl.h>
11: #include <limits.h>
12:
13: #include "sockets.h"
14:
15: base_socket::base_socket() {
16:    memset (&socket_addr, 0, sizeof (socket_addr));
17: }
18:
19: base_socket::~base_socket() {
20:    if (socket_fd != CLOSED_FD) close();
21: }
22:
23: void base_socket::close() {
24:    int status = ::close (socket_fd);
25:    if (status < 0) throw socket_sys_error ("close("
26:                            + to_string(socket_fd) + ")");
27:    socket_fd = CLOSED_FD;
28: }
29:
30: void base_socket::create() {
31:    socket_fd = ::socket (AF_INET, SOCK_STREAM, 0);
32:    if (socket_fd < 0) throw socket_sys_error ("socket");
33:    int on = 1;
34:    int status = ::setsockopt (socket_fd, SOL_SOCKET, SO_REUSEADDR,
35:                               &on, sizeof on);
36:    if (status < 0) throw socket_sys_error ("setsockopt");
37: }
38:
39: void base_socket::bind (const in_port_t port) {
40:    socket_addr.sin_family = AF_INET;
41:    socket_addr.sin_addr.s_addr = INADDR_ANY;
42:    socket_addr.sin_port = htons (port);
43:    int status = ::bind (socket_fd,
44:                         reinterpret_cast<sockaddr*> (&socket_addr),
45:                         sizeof socket_addr);
46:    if (status < 0) throw socket_sys_error ("bind(" + to_string (port)
47:                                            + ")");
48: }
49:
50: void base_socket::listen() const {
51:    int status = ::listen (socket_fd, SOMAXCONN);
52:    if (status < 0) throw socket_sys_error ("listen");
53: }
54:
```

```
 55:
 56: void base_socket::accept (base_socket& socket) const {
 57:    int addr_length = sizeof socket.socket_addr;
 58:    socket.socket_fd = ::accept (socket_fd,
 59:                  reinterpret_cast<sockaddr*> (&socket.socket_addr),
 60:                  reinterpret_cast<socklen_t*> (&addr_length));
 61:    if (socket.socket_fd < 0) throw socket_sys_error ("accept");
 62: }
 63:
 64: ssize_t base_socket::send (const void* buffer, size_t bufsize) {
 65:    int nbytes = ::send (socket_fd, buffer, bufsize, MSG_NOSIGNAL);
 66:    if (nbytes < 0) throw socket_sys_error ("send");
 67:    return nbytes;
 68: }
 69:
 70: ssize_t base_socket::recv (void* buffer, size_t bufsize) {
 71:    memset (buffer, 0, bufsize);
 72:    ssize_t nbytes = ::recv (socket_fd, buffer, bufsize, 0);
 73:    if (nbytes < 0) throw socket_sys_error ("recv");
 74:    return nbytes;
 75: }
 76:
 77: void base_socket::connect (const string host, const in_port_t port) {
 78:    struct hostent *hostp = ::gethostbyname (host.c_str());
 79:    if (hostp == NULL) throw socket_h_error ("gethostbyname("
 80:                               + host + ")");
 81:    socket_addr.sin_family = AF_INET;
 82:    socket_addr.sin_port = htons (port);
 83:    socket_addr.sin_addr = *reinterpret_cast<in_addr*> (hostp->h_addr);
 84:    int status = ::connect (socket_fd,
 85:                            reinterpret_cast<sockaddr*> (&socket_addr),
 86:                            sizeof (socket_addr));
 87:    if (status < 0) throw socket_sys_error ("connect(" + host + ":"
 88:                           + to_string (port) + ")");
 89: }
 90:
 91: void base_socket::set_socket_fd (int fd) {
 92:    socklen_t addrlen = sizeof socket_addr;
 93:    int rc = getpeername (fd, reinterpret_cast<sockaddr*> (&socket_addr),
 94:                          &addrlen);
 95:    if (rc < 0) throw socket_sys_error ("set_socket_fd("
 96:                        + to_string (fd) + "): getpeername");
 97:    socket_fd = fd;
 98:    if (socket_addr.sin_family != AF_INET)
 99:       throw socket_error ("address not AF_INET");
100: }
101:
102: void base_socket::set_non_blocking (const bool blocking) {
103:    int opts = ::fcntl (socket_fd, F_GETFL);
104:    if (opts < 0) throw socket_sys_error ("fcntl");
105:    if (blocking) opts |= O_NONBLOCK;
106:             else opts &= ~ O_NONBLOCK;
107:    opts = ::fcntl (socket_fd, F_SETFL, opts);
108:    if (opts < 0) throw socket_sys_error ("fcntl");
109: }
110:
```

```
111:
112: client_socket::client_socket (string host, in_port_t port) {
113:    base_socket::create();
114:    base_socket::connect (host, port);
115: }
116:
117: server_socket::server_socket (in_port_t port) {
118:    base_socket::create();
119:    base_socket::bind (port);
120:    base_socket::listen();
121: }
122:
123: string to_string (const hostinfo& info) {
124:    return info.hostname + " (" + to_string (info.addresses[0]) + ")";
125: }
126:
127: string to_string (const in_addr& ipv4_addr) {
128:    char buffer[INET_ADDRSTRLEN];
129:    const char *result = ::inet_ntop (AF_INET, &ipv4_addr,
130:                                      buffer, sizeof buffer);
131:    if (result == NULL) throw socket_sys_error ("inet_ntop");
132:    return result;
133: }
134:
135: string to_string (const base_socket& sock) {
136:    hostinfo info (sock.socket_addr.sin_addr);
137:    return info.hostname + " (" + to_string (info.addresses[0])
138:           + ") port " + to_string (ntohs (sock.socket_addr.sin_port));
139: }
140:
```

```
141:
142: string init_hostname (hostent* host) {
143:    if (host == nullptr) throw socket_h_error ("gethostbyname");
144:    return host->h_name;
145: }
146:
147: vector<string> init_aliases (hostent* host) {
148:    if (host == nullptr) throw socket_h_error ("gethostbyname");
149:    vector<string> init_aliases;
150:    for (char** alias = host->h_aliases; *alias != nullptr; ++alias) {
151:       init_aliases.push_back (*alias);
152:    }
153:    return init_aliases;
154: }
155:
156: vector<in_addr> init_addresses (hostent* host) {
157:    vector<in_addr> init_addresses;
158:    if (host == nullptr) throw socket_h_error ("gethostbyname");
159:    for (in_addr** addr =
160:                   reinterpret_cast<in_addr**> (host->h_addr_list);
161:         *addr != nullptr; ++addr) {
162:       init_addresses.push_back (**addr);
163:    }
164:    return init_addresses;
165: }
166:
167: hostinfo::hostinfo (hostent* host):
168:    hostname (init_hostname (host)),
169:    aliases (init_aliases (host)),
170:    addresses (init_addresses (host)) {
171: }
172:
173: hostinfo::hostinfo(): hostinfo (localhost()) {
174: }
175:
176: hostinfo::hostinfo (const string& hostname_):
177:           hostinfo (::gethostbyname (hostname_.c_str())) {
178: }
179:
180: hostinfo::hostinfo (const in_addr& ipv4_addr):
181:           hostinfo (::gethostbyaddr (&ipv4_addr, sizeof ipv4_addr,
182:                                      AF_INET)) {
183: }
184:
185: string localhost() {
186:    char hostname[HOST_NAME_MAX] {};
187:    int rc = gethostname (hostname, sizeof hostname);
188:    if (rc < 0) throw socket_sys_error ("gethostname");
189:    return hostname;
190: }
191:
```

```
 1: // $Id: cxi.cpp,v 1.5 2021-06-01 20:01:20-07 - - $
 2:
 3: #include <iostream>
 4: #include <memory>
 5: #include <string>
 6: #include <unordered_map>
 7: #include <vector>
 8: #include <fstream>
 9:
10: using namespace std;
11:
12: #include <libgen.h>
13: #include <sys/types.h>
14: #include <sys/stat.h>
15: #include <unistd.h>
16:
17: #include "protocol.h"
18: #include "logstream.h"
19: #include "sockets.h"
20:
21: logstream outlog (cout);
22: struct cxi_exit: public exception {};
23:
24: unordered_map<string,cxi_command> command_map {
25:    {"exit", cxi_command::EXIT},
26:    {"help", cxi_command::HELP},
27:    {"ls"  , cxi_command::LS  },
28:    {"get" , cxi_command::GET },
29:    {"put" , cxi_command::PUT },
30:    {"rm"  , cxi_command::RM  },
31: };
32:
33: static const char help[] = R"||(
34: exit         - Exit the program.  Equivalent to EOF.
35: get filename - Copy remote file to local host.
36: help         - Print help summary.
37: ls           - List names of files on remote server.
38: put filename - Copy local file to remote host.
39: rm filename  - Remove file from remote server.
40: )||";
41:
42: void cxi_help() {
43:    cout << help;
44: }
45:
46: void cxi_ls (client_socket& server) {
47:    cxi_header header;
48:    header.command = cxi_command::LS;
49:    outlog << "sending header " << header << endl;
50:    send_packet (server, &header, sizeof header);
51:    recv_packet (server, &header, sizeof header);
52:    outlog << "received header " << header << endl;
53:    if (header.command != cxi_command::LSOUT) {  //if!LSOUT
54:       outlog << "sent LS, server did not return LSOUT" << endl;
55:       outlog << "server returned " << header << endl;
56:    }else {
57:       size_t host_nbytes = ntohl (header.nbytes);//setnbytes
58:       auto buffer = make_unique<char[]> (host_nbytes + 1);
```

```
   59:       recv_packet (server, buffer.get(), host_nbytes);
   60:       outlog << "received " << host_nbytes << " bytes" << endl;
   61:       buffer[host_nbytes] = '\0';
   62:       cout << buffer.get();
   63:    }
   64: }
   65: void cxi_get (client_socket& server, vector<string>& splitvec) {
   66:    cxi_header header;
   67:    header.command = cxi_command::GET;
   68:    strcpy(header.filename,splitvec[1].c_str());
   69:
   70:    outlog << "sending header " << header << endl;
   71:    send_packet (server, &header, sizeof header);
   72:    recv_packet (server, &header, sizeof header);
   73:    outlog << "received header " << header << endl;
   74:    //if(!fileout) error
   75:    if (header.command != cxi_command::FILEOUT) {
   76:       outlog << "sent GET, server did not return FILEOUT" << endl;
   77:       outlog << "server returned " << header << endl;
   78:    }
   79:    //else set nbytes, declare buffer, recv, writefile,log<<success
   80:    else {
   81:       size_t host_nbytes = ntohl (header.nbytes);
   82:       auto buffer = make_unique<char[]> (host_nbytes + 1);
   83:       recv_packet (server, buffer.get(), host_nbytes);
   84:       buffer[host_nbytes] = '\0';
   85:       cout << buffer.get();
   86:    //create ofstream and write
   87:    ofstream write_file (header.filename, ios::out |ios::binary);
   88:    write_file.write(buffer.get(),host_nbytes);
   89:    write_file.close();
   90:    write_file << "GET sucess" << endl;
   91:    }
   92:
   93: }
   94: void cxi_put (client_socket& server, vector<string>& splitvec) {
   95:    cxi_header header;
   96:    header.command = cxi_command::PUT;
   97:
   98:    strcpy(header.filename,splitvec[1].c_str());
   99:    //if !file.exist ->error
  100:    struct stat stat_buf;
  101:    int status = stat(header.filename, &stat_buf);
  102:    if(status !=0){   //check if this works lol
  103:       cerr<< "Cannot put file. File: "<< header.filename <<
  104:              " does not exist" << endl;
  105:       return;
  106:    }
  107:    auto buffer = make_unique<char[]> (stat_buf.st_size);
  108:    //send payload
  109:    ifstream read_file (header.filename, ios::in | ios::binary);
  110:    read_file.read(buffer.get(), stat_buf.st_size);
  111:    header.command = cxi_command::PUT;//send PUT
  112:    send_packet (server, &header, sizeof header);
  113:    send_packet (server, buffer.get(), stat_buf.st_size);
  114:    recv_packet (server, &header, sizeof header);
  115:
  116:    if(header.command == cxi_command::ACK){
```

```
117:        cout << "PUT: ACK, sucess" << endl;
118:     }
119:     if(header.command == cxi_command::NAK){
120:        cout << "PUT: NAK, failure" << endl;
121:     }
122:     read_file.close();
123: }
```

```
124:
125: void cxi_rm (client_socket& server, vector<string>& splitvec) {
126:    cxi_header header;
127:    header.command = cxi_command::RM;//send RM
128:    strcpy(header.filename,splitvec[1].c_str());
129:
130:    send_packet (server, &header, sizeof header);
131:    recv_packet (server, &header, sizeof header);
132:    if(header.command == cxi_command::ACK){
133:       cout << "RM: ACK, sucess" << endl;
134:    }
135:    if(header.command == cxi_command::NAK){
136:       cout << "RM: NAK, failure" << endl;
137:    }
138: }
139:
140: void usage() {
141:    cerr << "Usage: " << outlog.execname() << " [host] [port]" << endl;
142:    throw cxi_exit();
143: }
144: //taken from split function in asg2
145: vector<string> split (const string& line, const string& delimiters) {
146:    vector<string> words;
147:    size_t end = 0;
148:     for (;;) {
149:       size_t start = line.find_first_not_of (delimiters, end);
150:       if (start == string::npos) break;
151:       end = line.find_first_of (delimiters, start);
152:       words.push_back (line.substr (start, end - start));
153:    }
154:    return words;
155: }
156:
157: int main (int argc, char** argv) {
158:    outlog.execname (basename (argv[0]));
159:    outlog << "starting" << endl;
160:    vector<string> args (&argv[1], &argv[argc]);
161:    if (args.size() > 2) usage();
162:    string host = get_cxi_server_host (args, 0);
163:    in_port_t port = get_cxi_server_port (args, 1);
164:    outlog << to_string (hostinfo()) << endl;
165:    try {
166:       outlog << "connecting to " << host << " port " << port << endl;
167:       client_socket server (host, port);
168:       outlog << "connected to " << to_string (server) << endl;
169:       for (;;) {
170:          string line;
171:          getline (cin, line); //split line after if
172:          if (cin.eof()) throw cxi_exit();
173:          vector<string> splitline = split(line," ");//makes new vector
174:          outlog << "command " << line << endl;
175:          const auto& itor = command_map.find (splitline[0]);
176:          cxi_command cmd = itor == command_map.end()
177:                          ? cxi_command::ERROR : itor->second;
178:          switch (cmd) { //added appropriate switch cases
179:             case cxi_command::EXIT:
180:                throw cxi_exit();
181:                break;
```

```
182:             case cxi_command::HELP:
183:                cxi_help();
184:                break;
185:             case cxi_command::LS:
186:                cxi_ls (server);
187:                break;
188:             case cxi_command::GET:
189:                cxi_get (server, splitline);
190:                break;
191:             case cxi_command::PUT:
192:                cxi_put (server, splitline);
193:                break;
194:             case cxi_command::RM:
195:                cxi_rm (server, splitline);
196:                break;
197:             default:
198:                outlog << line << ": invalid command" << endl;
199:                break;
200:          }
201:       }
202:    }catch (socket_error& error) {
203:       outlog << error.what() << endl;
204:    }catch (cxi_exit& error) {
205:       outlog << "caught cxi_exit" << endl;
206:    }
207:    outlog << "finishing" << endl;
208:    return 0;
209: }
210:
```

```
 1: // $Id: cxid.cpp,v 1.6 2021-06-01 20:01:20-07 - - $
 2:
 3: #include <iostream>
 4: #include <string>
 5: #include <vector>
 6: #include <fstream>
 7: #include <memory>
 8: using namespace std;
 9:
10: #include <libgen.h>
11: #include <sys/types.h>
12: #include <sys/stat.h>
13: #include <unistd.h>
14:
15: #include "protocol.h"
16: #include "logstream.h"
17: #include "sockets.h"
18:
19: logstream outlog (cout);
20: struct cxi_exit: public exception {};
21:
22: void reply_ls (accepted_socket& client_sock, cxi_header& header) {
23:    const char* ls_cmd = "ls -l 2>&1";
24:    FILE* ls_pipe = popen (ls_cmd, "r");
25:    if (ls_pipe == NULL) { //gets ls output
26:       outlog << ls_cmd << ": " << strerror (errno) << endl;
27:       header.command = cxi_command::NAK;//if err return NAK
28:       header.nbytes = htonl (errno);
29:       send_packet (client_sock, &header, sizeof header);
30:       return;
31:    }
32:    string ls_output;
33:    char buffer[0x1000];//else create buffer
34:    for (;;) {
35:       char* rc = fgets (buffer, sizeof buffer, ls_pipe);
36:       if (rc == nullptr) break;
37:       ls_output.append (buffer);//make output
38:    }
39:    int status = pclose (ls_pipe);//close status
40:    if (status < 0) outlog << ls_cmd << ": " << strerror (errno) << endl;
41:               else outlog << ls_cmd << ": exit " << (status >> 8)
42:                           << " signal " << (status & 0x7F)
43:                           << " core " << (status >> 7 & 1) << endl;
44:    header.command = cxi_command::LSOUT;//sends LSOUT
45:    header.nbytes = htonl (ls_output.size());//sets nbytes
46:    memset (header.filename, 0, FILENAME_SIZE);//memset
47:    outlog << "sending header " << header << endl;
48:    send_packet (client_sock, &header, sizeof header);//sends lsout
49:    send_packet (client_sock, ls_output.c_str(), ls_output.size());
50:    outlog << "sent " << ls_output.size() << " bytes" << endl;
51: }
52: void reply_get (accepted_socket& client_sock, cxi_header& header) {
53:    struct stat stat_buf;   //or file?
54:    int status = stat(header.filename, &stat_buf);
55:    if(status !=0){   //check if this works lol
56:       cerr<< "Cannot get file. File: " << header.filename <<
57:              " does not exist" << endl;
58:       header.command = cxi_command::NAK;  //send NAK
```

```
 59:       header.nbytes = htonl (errno);
 60:       send_packet (client_sock, &header, sizeof header);
 61:       return;
 62:    }
 63:    //check size of file, declare buf, set nbytes
 64:    auto buffer = make_unique<char[]> (stat_buf.st_size);
 65:    ifstream read_file (header.filename, ios::in | ios::binary);
 66:    read_file.read(buffer.get(), stat_buf.st_size);
 67:    read_file.close();
 68:    header.command = cxi_command::FILEOUT;
 69:    header.nbytes = htonl (stat_buf.st_size);//set nbytes
 70:    //sends FILEOUT
 71:    send_packet (client_sock, &header, sizeof header);
 72:    //sends output
 73:    send_packet (client_sock, buffer.get(), stat_buf.st_size);
 74: }
 75: void reply_put (accepted_socket& client_sock, cxi_header& header) {
 76:    struct stat stat_buf;
 77:    int status = stat(header.filename, &stat_buf);
 78:    if(status !=0){
 79:       cerr<< "Cannot get file. File: " << header.filename <<
 80:              " does not exist" << endl;
 81:       header.command = cxi_command::NAK;  //send NAK
 82:       header.nbytes = htonl (errno);
 83:       send_packet (client_sock, &header, sizeof header);
 84:       return;
 85:    }
 86:    size_t host_nbytes = ntohl (header.nbytes);//setnbytes
 87:    auto buffer = make_unique<char[]> (host_nbytes+1);//set buffer
 88:
 89:    recv_packet (client_sock, buffer.get(), host_nbytes);
 90:    buffer[host_nbytes] = '\0';
 91:    ofstream write_file (header.filename, ios::out | ios::binary);
 92:    write_file.write(buffer.get(),host_nbytes);
 93:    header.command = cxi_command::ACK;  //send ACK
 94:    send_packet (client_sock, &header, sizeof header);//sends FILEOUT
 95:
 96:    write_file.close();
 97:
 98:
 99: }
100: void reply_rm (accepted_socket& client_sock, cxi_header& header) {
101:    int status = unlink(header.filename);
102:    if(status !=0){
103:       header.command = cxi_command::NAK;  //send NAK
104:       header.nbytes = htonl (errno);
105:       send_packet (client_sock, &header, sizeof header);
106:    }
107:    header.command = cxi_command::ACK;  //send ACK
108:    send_packet (client_sock, &header, sizeof header);
109: }
110:
```

```
//edit
  112: void run_server (accepted_socket& client_sock) {
  113:    outlog.execname (outlog.execname() + "*");
  114:    outlog << "connected to " << to_string (client_sock) << endl;
  115:    try {
  116:       for (;;) {
  117:          cxi_header header;
  118:          recv_packet (client_sock, &header, sizeof header);
  119:          outlog << "received header " << header << endl;
  120:          switch (header.command) {
  121:             case cxi_command::LS:   //put, rm, get
  122:                reply_ls (client_sock, header);
  123:                break;
  124:             case cxi_command::GET:
  125:                reply_get (client_sock, header);
  126:                break;
  127:             case cxi_command::PUT:
  128:                reply_put (client_sock, header);
  129:                break;
  130:             case cxi_command::RM:
  131:                reply_rm (client_sock, header);
  132:                break;
  133:             default:
  134:                outlog << "invalid client header:" << header << endl;
  135:                break;
  136:          }
  137:       }
  138:    }catch (socket_error& error) {
  139:       outlog << error.what() << endl;
  140:    }catch (cxi_exit& error) {
  141:       outlog << "caught cxi_exit" << endl;
  142:    }
  143:    outlog << "finishing" << endl;
  144:    throw cxi_exit();
  145: }
  146:
  147: void fork_cxiserver (server_socket& server, accepted_socket& accept) {
  148:    pid_t pid = fork();
  149:    if (pid == 0) { // child
  150:       server.close();
  151:       run_server (accept);
  152:       throw cxi_exit();
  153:    }else {
  154:       accept.close();
  155:       if (pid < 0) {
  156:          outlog << "fork failed: " << strerror (errno) << endl;
  157:       }else {
  158:          outlog << "forked cxiserver pid " << pid << endl;
  159:       }
  160:    }
  161: }
  162:
```

```
163:
164: void reap_zombies() {
165:    for (;;) {
166:       int status;
167:       pid_t child = waitpid (-1, &status, WNOHANG);
168:       if (child <= 0) break;
169:       outlog << "child " << child
170:              << " exit " << (status >> 8)
171:              << " signal " << (status & 0x7F)
172:              << " core " << (status >> 7 & 1) << endl;
173:    }
174: }
175:
176: void signal_handler (int signal) {
177:    outlog << "signal_handler: caught " << strsignal (signal) << endl;
178:    reap_zombies();
179: }
180:
181: void signal_action (int signal, void (*handler) (int)) {
182:    struct sigaction action;
183:    action.sa_handler = handler;
184:    sigfillset (&action.sa_mask);
185:    action.sa_flags = 0;
186:    int rc = sigaction (signal, &action, nullptr);
187:    if (rc < 0) outlog << "sigaction " << strsignal (signal)
188:                       << " failed: " << strerror (errno) << endl;
189: }
190:
```

```
191:
192: int main (int argc, char** argv) {
193:    outlog.execname (basename (argv[0]));
194:    outlog << "starting" << endl;
195:    vector<string> args (&argv[1], &argv[argc]);
196:    signal_action (SIGCHLD, signal_handler);
197:    in_port_t port = get_cxi_server_port (args, 0);
198:    try {
199:       server_socket listener (port);
200:       for (;;) {
201:          outlog << to_string (hostinfo()) << " accepting port "
202:              << to_string (port) << endl;
203:          accepted_socket client_sock;
204:          for (;;) {
205:             try {
206:                listener.accept (client_sock);
207:                break;
208:             }catch (socket_sys_error& error) {
209:                switch (error.sys_errno) {
210:                   case EINTR:
211:                      outlog << "listener.accept caught "
212:                          << strerror (EINTR) << endl;
213:                      break;
214:                   default:
215:                      throw;
216:                }
217:             }
218:          }
219:          outlog << "accepted " << to_string (client_sock) << endl;
220:          try {
221:             fork_cxiserver (listener, client_sock);
222:             reap_zombies();
223:          }catch (socket_error& error) {
224:             outlog << error.what() << endl;
225:          }
226:       }
227:    }catch (socket_error& error) {
228:       outlog << error.what() << endl;
229:    }catch (cxi_exit& error) {
230:       outlog << "caught cxi_exit" << endl;
231:    }
232:    outlog << "finishing" << endl;
233:    return 0;
234: }
235:
```

```
 1: # $Id: Makefile,v 1.24 2021-06-01 20:01:20-07 - - $
 2:
 3: MKFILE      = Makefile
 4: DEPFILE     = ${MKFILE}.dep
 5: NOINCL      = ci clean spotless
 6: NEEDINCL    = ${filter ${NOINCL}, ${MAKECMDGOALS}}
 7: GMAKE       = ${MAKE} --no-print-directory
 8:
 9: GPPWARN     = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
10: GPPOPTS     = ${GPPWARN} -fdiagnostics-color=never
11: COMPILECPP  = g++ -std=gnu++17 -g -O0 ${GPPOPTS}
12: MAKEDEPCPP  = g++ -std=gnu++17 -MM ${GPPOPTS}
13: UTILBIN     = /afs/cats.ucsc.edu/courses/cse111-wm/bin
14:
15: MODULES     = logstream protocol sockets
16: EXECBINS    = cxi cxid
17: ALLMODS     = ${MODULES} ${EXECBINS}
18: SOURCELIST  = ${foreach MOD, ${ALLMODS}, ${MOD}.h ${MOD}.tcc ${MOD}.cpp}
19: CPPSOURCE   = ${wildcard ${MODULES:=.cpp} ${EXECBINS:=.cpp}}
20: ALLSOURCE   = ${wildcard ${SOURCELIST}} ${MKFILE}
21: CPPLIBS     = ${wildcard ${MODULES:=.cpp}}
22: OBJLIBS     = ${CPPLIBS:.cpp=.o}
23: CXIOBJS     = cxi.o ${OBJLIBS}
24: CXIDOBJS    = cxid.o ${OBJLIBS}
25: CLEANOBJS   = ${OBJLIBS} ${CXIOBJS} ${CXIDOBJS}
26: LISTING     = Listing.ps
27:
28: export PATH := ${PATH}:/afs/cats.ucsc.edu/courses/cse110a-wm/bin
29:
30: all: ${DEPFILE} ${EXECBINS}
31:
32: cxi: ${CXIOBJS}
33:         ${COMPILECPP} -o $@ ${CXIOBJS}
34:
35: cxid: ${CXIDOBJS}
36:         ${COMPILECPP} -o $@ ${CXIDOBJS}
37:
38: %.o: %.cpp
39:         - checksource $<
40:         - cpplint.py.perl $<
41:         ${COMPILECPP} -c $<
42:
43: ci: ${ALLSOURCE}
44:         cid -is ${ALLSOURCE}
45:         - checksource ${ALLSOURCE}
46:
47: lis: all ${ALLSOURCE} ${DEPFILE}
48:         - pkill -g 0 gv || true
49:         mkpspdf ${LISTING} ${ALLSOURCE} ${DEPFILE}
50:
51: clean:
52:         - rm ${LISTING} ${LISTING:.ps=.pdf} ${CLEANOBJS} core
53:
54: spotless: clean
55:         - rm ${EXECBINS} ${DEPFILE}
56:
```

```
57:
58: dep: ${ALLCPPSRC}
59:         @ echo "# ${DEPFILE} created $(LC_TIME=C date)" >${DEPFILE}
60:         ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}
61:
62: ${DEPFILE}:
63:         @ touch ${DEPFILE}
64:         ${GMAKE} dep
65:
66: again: ${ALLSOURCE}
67:         ${GMAKE} spotless dep ci all lis
68:
69: submit:
70:         submit cse111-wm.s21 asg4 *.cpp *.h Makefile README
71:
72: ifeq (${NEEDINCL}, )
73: include ${DEPFILE}
74: endif
75:
```

```
1: # Makefile.dep created
2: protocol.o: protocol.cpp protocol.h sockets.h
3: sockets.o: sockets.cpp sockets.h
4: cxi.o: cxi.cpp protocol.h sockets.h logstream.h
5: cxid.o: cxid.cpp protocol.h sockets.h logstream.h
```