

```
1: // $Id: logstream.h,v 1.5 2020-12-12 22:09:29-08 - - $
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: using namespace std;
18:
19: class logstream {
20:     private:
21:         ostream& out_;
22:         string execname_;
23:     public:
24:
25:         // Constructor may or may not have the execname available.
26:         logstream (ostream& out, const string& execname = ""):
27:             out_ (out), execname_ (execname) {
28:         }
29:
30:         // First line of main should set execname if logstream is global.
31:         void execname (const string& name) { execname_ = name; }
32:         string execname() { return execname_; }
33:
34:         // First call should be the logstream, not cout.
35:         // Then forward result to the standard ostream.
36:         template <typename T>
37:         ostream& operator<< (const T& obj) {
38:             assert (execname_.size() > 0);
39:             out_ << execname_ << "(" << getpid() << "): " << obj;
40:             return out_;
41:         }
42:
43: };
44:
45: #endif
46:
```

[illegible]

```
1: // $Id: protocol.cpp,v 1.14 2020-12-12 22:09:29-08 - - $
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:     return out << "{" << to_hex32_string (header.nbytes) << ':'
63:         << header.nbytes << ':' << ntohl (header.nbytes) << ", "
64:         << unsigned (header.command)
65:         << "(" << to_string (header.command) << "), \""
66:         << header.filename << "\"}";
67: }
68:
69: string get_cxi_server_host (const vector<string>& args, size_t index) {
70:     if (index < args.size()) return args[index];
71:     char* host = getenv ("CIX_SERVER_HOST");
72:     if (host != nullptr) return host;
73:     return "localhost";
74: }
75:
76: in_port_t get_cxi_server_port (const vector<string>& args,
77:                                size_t index) {
78:     string port = "-1";
79:     if (index < args.size()) port = args[index];
80:     else {
81:         char* envport = getenv ("CIX_SERVER_PORT");
82:         if (envport != nullptr) port = envport;
83:     }
84:     return stoi (port);
85: }
86:
```

```
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 1.1 2020-11-22 16:51:43-08 - - $
2:
3: #include <iostream>
4: #include <memory>
5: #include <string>
6: #include <unordered_map>
7: #include <vector>
8: using namespace std;
9:
10: #include <libgen.h>
11: #include <sys/types.h>
12: #include <unistd.h>
13:
14: #include "protocol.h"
15: #include "logstream.h"
16: #include "sockets.h"
17:
18: logstream outlog (cout);
19: struct cxi_exit: public exception {};
20:
21: unordered_map<string,cxi_command> command_map {
22:     {"exit", cxi_command::EXIT},
23:     {"help", cxi_command::HELP},
24:     {"ls" , cxi_command::LS },
25: };
26:
27: static const char help[] = R"|(
28: exit      - Exit the program.  Equivalent to EOF.
29: get filename - Copy remote file to local host.
30: help      - Print help summary.
31: ls        - List names of files on remote server.
32: put filename - Copy local file to remote host.
33: rm filename - Remove file from remote server.
34: )|";
35:
36: void cxi_help() {
37:     cout << help;
38: }
39:
40: void cxi_ls (client_socket& server) {
41:     cxi_header header;
42:     header.command = cxi_command::LS;
43:     outlog << "sending header " << header << endl;
44:     send_packet (server, &header, sizeof header);
45:     recv_packet (server, &header, sizeof header);
46:     outlog << "received header " << header << endl;
47:     if (header.command != cxi_command::LSOUT) {
48:         outlog << "sent LS, server did not return LSOUT" << endl;
49:         outlog << "server returned " << header << endl;
50:     }else {
51:         size_t host_nbytes = ntohl (header.nbytes);
52:         auto buffer = make_unique<char[]> (host_nbytes + 1);
53:         recv_packet (server, buffer.get(), host_nbytes);
54:         outlog << "received " << host_nbytes << " bytes" << endl;
55:         buffer[host_nbytes] = '\0';
56:         cout << buffer.get();
57:     }
58: }
```

**02/17/21
18:20:58**

\$cse111-wm/Assignments/asg4-client-server/code
cxi.cpp

2/3

59:

```
60:
61: void usage() {
62:     cerr << "Usage: " << outlog.execname() << " [host] [port]" << endl;
63:     throw cxi_exit();
64: }
65:
66: int main (int argc, char** argv) {
67:     outlog.execname (basename (argv[0]));
68:     outlog << "starting" << endl;
69:     vector<string> args (&argv[1], &argv[argc]);
70:     if (args.size() > 2) usage();
71:     string host = get_cxi_server_host (args, 0);
72:     in_port_t port = get_cxi_server_port (args, 1);
73:     outlog << to_string (hostinfo()) << endl;
74:     try {
75:         outlog << "connecting to " << host << " port " << port << endl;
76:         client_socket server (host, port);
77:         outlog << "connected to " << to_string (server) << endl;
78:         for (;;) {
79:             string line;
80:             getline (cin, line);
81:             if (cin.eof()) throw cxi_exit();
82:             outlog << "command " << line << endl;
83:             const auto& itor = command_map.find (line);
84:             cxi_command cmd = itor == command_map.end()
85:                 ? cxi_command::ERROR : itor->second;
86:             switch (cmd) {
87:                 case cxi_command::EXIT:
88:                     throw cxi_exit();
89:                     break;
90:                 case cxi_command::HELP:
91:                     cxi_help();
92:                     break;
93:                 case cxi_command::LS:
94:                     cxi_ls (server);
95:                     break;
96:                 default:
97:                     outlog << line << ": invalid command" << endl;
98:                     break;
99:             }
100:         }
101:     } catch (socket_error& error) {
102:         outlog << error.what() << endl;
103:     } catch (cxi_exit& error) {
104:         outlog << "caught cxi_exit" << endl;
105:     }
106:     outlog << "finishing" << endl;
107:     return 0;
108: }
109:
```

```
1: // $Id: cxid.cpp,v 1.3 2020-12-12 22:09:29-08 - - $
2:
3: #include <iostream>
4: #include <string>
5: #include <vector>
6: using namespace std;
7:
8: #include <libgen.h>
9: #include <sys/types.h>
10: #include <unistd.h>
11:
12: #include "protocol.h"
13: #include "logstream.h"
14: #include "sockets.h"
15:
16: logstream outlog (cout);
17: struct cxi_exit: public exception {};
18:
19: void reply_ls (accepted_socket& client_sock, cxi_header& header) {
20:     const char* ls_cmd = "ls -l 2>&1";
21:     FILE* ls_pipe = popen (ls_cmd, "r");
22:     if (ls_pipe == NULL) {
23:         outlog << ls_cmd << ": " << strerror (errno) << endl;
24:         header.command = cxi_command::NAK;
25:         header.nbytes = htonl (errno);
26:         send_packet (client_sock, &header, sizeof header);
27:         return;
28:     }
29:     string ls_output;
30:     char buffer[0x1000];
31:     for (;;) {
32:         char* rc = fgets (buffer, sizeof buffer, ls_pipe);
33:         if (rc == nullptr) break;
34:         ls_output.append (buffer);
35:     }
36:     int status = pclose (ls_pipe);
37:     if (status < 0) outlog << ls_cmd << ": " << strerror (errno) << endl;
38:         else outlog << ls_cmd << ": exit " << (status >> 8)
39:             << " signal " << (status & 0x7F)
40:             << " core " << (status >> 7 & 1) << endl;
41:     header.command = cxi_command::LSOUT;
42:     header.nbytes = htonl (ls_output.size());
43:     memset (header.filename, 0, FILENAME_SIZE);
44:     outlog << "sending header " << header << endl;
45:     send_packet (client_sock, &header, sizeof header);
46:     send_packet (client_sock, ls_output.c_str(), ls_output.size());
47:     outlog << "sent " << ls_output.size() << " bytes" << endl;
48: }
49:
```



```
50:
51: void run_server (accepted_socket& client_sock) {
52:     outlog.execname (outlog.execname() + "*");
53:     outlog << "connected to " << to_string (client_sock) << endl;
54:     try {
55:         for (;;) {
56:             cxi_header header;
57:             recv_packet (client_sock, &header, sizeof header);
58:             outlog << "received header " << header << endl;
59:             switch (header.command) {
60:                 case cxi_command::LS:
61:                     reply_ls (client_sock, header);
62:                     break;
63:                 default:
64:                     outlog << "invalid client header:" << header << endl;
65:                     break;
66:             }
67:         }
68:     } catch (socket_error& error) {
69:         outlog << error.what() << endl;
70:     } catch (cxi_exit& error) {
71:         outlog << "caught cxi_exit" << endl;
72:     }
73:     outlog << "finishing" << endl;
74:     throw cxi_exit();
75: }
76:
77: void fork_cxiserver (server_socket& server, accepted_socket& accept) {
78:     pid_t pid = fork();
79:     if (pid == 0) { // child
80:         server.close();
81:         run_server (accept);
82:         throw cxi_exit();
83:     } else {
84:         accept.close();
85:         if (pid < 0) {
86:             outlog << "fork failed: " << strerror (errno) << endl;
87:         } else {
88:             outlog << "forked cxiserver pid " << pid << endl;
89:         }
90:     }
91: }
92:
```

```
93:
94: void reap_zombies() {
95:     for (;;) {
96:         int status;
97:         pid_t child = waitpid (-1, &status, WNOHANG);
98:         if (child <= 0) break;
99:         outlog << "child " << child
100:             << " exit " << (status >> 8)
101:             << " signal " << (status & 0x7F)
102:             << " core " << (status >> 7 & 1) << endl;
103:     }
104: }
105:
106: void signal_handler (int signal) {
107:     outlog << "signal_handler: caught " << strsignal (signal) << endl;
108:     reap_zombies();
109: }
110:
111: void signal_action (int signal, void (*handler) (int)) {
112:     struct sigaction action;
113:     action.sa_handler = handler;
114:     sigfillset (&action.sa_mask);
115:     action.sa_flags = 0;
116:     int rc = sigaction (signal, &action, nullptr);
117:     if (rc < 0) outlog << "sigaction " << strsignal (signal)
118:         << " failed: " << strerror (errno) << endl;
119: }
120:
```

```
121:
122: int main (int argc, char** argv) {
123:     outlog.execname (basename (argv[0]));
124:     outlog << "starting" << endl;
125:     vector<string> args (&argv[1], &argv[argc]);
126:     signal_action (SIGCHLD, signal_handler);
127:     in_port_t port = get_cxi_server_port (args, 0);
128:     try {
129:         server_socket listener (port);
130:         for (;;) {
131:             outlog << to_string (hostinfo()) << " accepting port "
132:                 << to_string (port) << endl;
133:             accepted_socket client_sock;
134:             for (;;) {
135:                 try {
136:                     listener.accept (client_sock);
137:                     break;
138:                 }catch (socket_sys_error& error) {
139:                     switch (error.sys_errno) {
140:                         case EINTR:
141:                             outlog << "listener.accept caught "
142:                                 << strerror (EINTR) << endl;
143:                             break;
144:                         default:
145:                             throw;
146:                     }
147:                 }
148:             }
149:             outlog << "accepted " << to_string (client_sock) << endl;
150:             try {
151:                 fork_cxiserver (listener, client_sock);
152:                 reap_zombies();
153:             }catch (socket_error& error) {
154:                 outlog << error.what() << endl;
155:             }
156:         }
157:     }catch (socket_error& error) {
158:         outlog << error.what() << endl;
159:     }catch (cxi_exit& error) {
160:         outlog << "caught cxi_exit" << endl;
161:     }
162:     outlog << "finishing" << endl;
163:     return 0;
164: }
165:
```

```
1: # $Id: Makefile,v 1.19 2021-02-17 18:20:54-08 - - $
2:
3: MKFILE      = Makefile
4: DEPPFILE    = ${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: ${DEPPFILE} ${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} ${DEPPFILE}
48:     - pkill -g 0 gv || true
49:     mkpspdf ${LISTING} ${ALLSOURCE} ${DEPPFILE}
50:
51: clean:
52:     - rm ${LISTING} ${LISTING:.ps=.pdf} ${CLEANOBJS} core
53:
54: spotless: clean
55:     - rm ${EXECBINS} ${DEPPFILE}
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:     ${GMAKE} spotless dep ci all lis
68:
69: ifeq (${NEEDINCL}, )
70: include ${DEPFILE}
71: endif
72:
```

```
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
```