

```
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 1.12 2021-05-18 01:32:29-07 - - $
2:
3: #ifndef __PROTOCOL__H__
4: #define __PROTOCOL__H__
5:
6: #include <cstdint>
7: using namespace std;
8:
9: #include "socket.h"
10:
11: enum class cxi_command : uint8_t {
12:     ERROR = 0, EXIT, GET, HELP, LS, PUT, RM, FILEOUT, LSOUT, ACK, NAK,
13: };
14:
15: constexpr size_t FILENAME_SIZE = 59;
16: constexpr size_t HEADER_SIZE = 64;
17:
18: struct cxi_header {
19:     uint32_t nbytes {};
20:     cxi_command command {cxi_command::ERROR};
21:     char filename[FILENAME_SIZE] {};
22: };
23:
24: static_assert (sizeof (cxi_header) == HEADER_SIZE);
25:
26: void send_packet (base_socket& socket,
27:                  const void* buffer, size_t bufsize);
28:
29: void recv_packet (base_socket& socket, void* buffer, size_t bufsize);
30:
31: ostream& operator<< (ostream& out, const cxi_header& header);
32:
33: in_port_t get_cxi_server_port (const string& port_arg);
34:
35: #endif
36:
```

```
1: // $Id: protocol.cpp,v 1.17 2021-05-18 01:32:29-07 - - $
2:
3: #include <iomanip>
4: #include <iostream>
5: #include <limits>
6: #include <string>
7: #include <unordered_map>
8: using namespace std;
9:
10: #include "protocol.h"
11:
12: string to_string (cxi_command command) {
13:     switch (command) {
14:         case cxi_command::ERROR : return "ERROR" ;
15:         case cxi_command::EXIT  : return "EXIT"  ;
16:         case cxi_command::GET   : return "GET"   ;
17:         case cxi_command::HELP  : return "HELP"  ;
18:         case cxi_command::LS    : return "LS"    ;
19:         case cxi_command::PUT   : return "PUT"   ;
20:         case cxi_command::RM    : return "RM"    ;
21:         case cxi_command::FILEOUT: return "FILEOUT";
22:         case cxi_command::LSOUT : return "LSOUT" ;
23:         case cxi_command::ACK   : return "ACK"   ;
24:         case cxi_command::NAK   : return "NAK"   ;
25:         default                 : return "????"  ;
26:     };
27: }
28:
29:
30: void send_packet (base_socket& socket,
31:                  const void* buffer, size_t bufsize) {
32:     const char* bufptr = static_cast<const char*> (buffer);
33:     ssize_t ntosend = bufsize;
34:     do {
35:         ssize_t nbytes = socket.send (bufptr, ntosend);
36:         if (nbytes < 0) throw socket_sys_error (to_string (socket));
37:         bufptr += nbytes;
38:         ntosend -= nbytes;
39:     }while (ntosend > 0);
40: }
41:
42: void recv_packet (base_socket& socket, void* buffer, size_t bufsize) {
43:     char* bufptr = static_cast<char*> (buffer);
44:     ssize_t ntorecv = bufsize;
45:     do {
46:         ssize_t nbytes = socket.recv (bufptr, ntorecv);
47:         if (nbytes < 0) throw socket_sys_error (to_string (socket));
48:         if (nbytes == 0) throw socket_error (to_string (socket)
49:                                             + " is closed");
50:         bufptr += nbytes;
51:         ntorecv -= nbytes;
52:     }while (ntorecv > 0);
53: }
54:
```

```
55:
56: string to_hex32_string (uint32_t num) {
57:     ostringstream stream;
58:     stream << "0x" << hex << uppercase << setfill('0') << setw(8) << num;
59:     return stream.str();
60: }
61:
62: ostream& operator<< (ostream& out, const cxi_header& header) {
63:     constexpr size_t WARNING_NBYTES = 1<<20;
64:     uint32_t nbytes = htonl (header.nbytes);
65:     if (nbytes > WARNING_NBYTES) {
66:         out << "WARNING: Payload nbytes " << nbytes << " > "
67:             << WARNING_NBYTES << endl;
68:     }
69:     return out << "{" << to_hex32_string (header.nbytes) << ':' <<
70:         << header.nbytes << ':' << ntohl (header.nbytes) << ", "
71:         << unsigned (header.command)
72:         << "(" << to_string (header.command) << "), \""
73:         << header.filename << "\"}";
74: }
75:
76: in_port_t get_cxi_server_port (const string& port_arg) {
77:     auto error = socket_error (port_arg + ": invalid port number");
78:     try {
79:         constexpr int min = numeric_limits<in_port_t>::min();
80:         constexpr int max = numeric_limits<in_port_t>::max();
81:         int port = stoi (port_arg);
82:         if (port < min or port > max) throw error;
83:         return port;
84:     } catch (invalid_argument&) { // thrown by stoi
85:         throw error;
86:     } catch (out_of_range&) { // thrown by stoi
87:         throw error;
88:     }
89: }
90:
```

```
1: // $Id: socket.h,v 1.4 2021-11-16 13:28:07-08 - - $
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 <sys/socket.h>
16: #include <sys/types.h>
17: #include <sys/wait.h>
18: #include <unistd.h>
19:
20: //
21: // class base_socket:
22: // mostly protected and not used by applications
23: //
24:
25: class base_socket {
26:     private:
27:         static constexpr size_t MAXRECV = 0xFFFF;
28:         static constexpr int CLOSED_FD = -1;
29:         int socket_fd {CLOSED_FD};
30:         sockaddr_in socket_addr;
31:     protected:
32:         base_socket(); // only derived classes may construct
33:         base_socket (const base_socket&) = delete; // prevent copying
34:         base_socket& operator= (const base_socket&) = delete;
35:         ~base_socket();
36:
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:
43:         // client_socket initialization
44:         void connect (const string host, const in_port_t port);
45:
46:         // accepted_socket initialization
47:         void set_socket_fd (int fd);
48:
49:     public:
50:         // data transmission
51:         void close();
52:         ssize_t send (const void* buffer, size_t bufsize);
53:         ssize_t recv (void* buffer, size_t bufsize);
54:         void set_non_blocking (const bool);
55:         friend string to_string (const base_socket& sock);
56: };
57:
```

```
58:
59: //
60: // class accepted_socket
61: // used by server when a client connects
62: //
63:
64: class accepted_socket: public base_socket {
65:     public:
66:         accepted_socket() {}
67:         accepted_socket (int fd) { set_socket_fd (fd); }
68: };
69:
70: //
71: // class client_socket
72: // used by client application to connect to server
73: //
74:
75: class client_socket: public base_socket {
76:     public:
77:         client_socket (string host, in_port_t port);
78: };
79:
80: //
81: // class server_socket
82: // single use class by server application
83: //
84:
85: class server_socket: public base_socket {
86:     public:
87:         server_socket (in_port_t port);
88:         void accept (accepted_socket& sock) {
89:             base_socket::accept (sock);
90:         }
91: };
92:
```

```
93:
94: //
95: // class socket_error
96: // base class for throwing socket errors
97: //
98:
99: class socket_error: public runtime_error {
100:     public:
101:         explicit socket_error (const string& what): runtime_error(what){}
102: };
103:
104: //
105: // class socket_sys_error
106: // subclass to record status of extern int errno variable
107: //
108:
109: class socket_sys_error: public socket_error {
110:     public:
111:         int sys_errno;
112:         explicit socket_sys_error (const string& what):
113:             socket_error(what + ": " + strerror (errno)),
114:             sys_errno(errno) {}
115: };
116:
117: //
118: // class socket_h_error
119: // subclass to record status of extern int h_errno variable
120: //
121:
122: class socket_h_error: public socket_error {
123:     public:
124:         int host_errno;
125:         explicit socket_h_error (const string& what):
126:             socket_error(what + ": " + hstrerror (h_errno)),
127:             host_errno(h_errno) {}
128: };
129:
```

```
130:
131: //
132: // class hostinfo
133: // information about a host given hostname or IPv4 address
134: //
135:
136: class hostinfo {
137:     public:
138:         const string hostname;
139:         const vector<string> aliases;
140:         const vector<in_addr> addresses;
141:         hostinfo (); // localhost
142:         hostinfo (hostent*);
143:         hostinfo (const string& hostname);
144:         hostinfo (const in_addr& ipv4_addr);
145:         friend string to_string (const hostinfo&);
146: };
147:
148: string localhost();
149: string to_string (const in_addr& ipv4_addr);
150:
151: #endif
152:
```



```
1: // $Id: socket.cpp,v 1.5 2021-05-12 21:22:38-07 - - $
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 "socket.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: debug.h,v 1.2 2021-11-08 00:01:44-08 - - $
2:
3: #ifndef __DEBUG_H__
4: #define __DEBUG_H__
5:
6: #include <bitset>
7: #include <climits>
8: #include <string>
9: using namespace std;
10:
11: // debug -
12: //      static class for maintaining global debug flags.
13: // setflags -
14: //      Takes a string argument, and sets a flag for each char in the
15: //      string. As a special case, '@', sets all flags.
16: // getflag -
17: //      Used by the DEBUGF macro to check to see if a flag has been set.
18: //      Not to be called by user code.
19:
20: class debugflags {
21:     private:
22:         using flagset_ = bitset<UCHAR_MAX + 1>;
23:         static flagset_ flags_;
24:     public:
25:         static void setflags (const string& optflags);
26:         static bool getflag (char flag);
27:         static void where (char flag, const char* file, int line,
28:                           const char* pretty_function);
29: };
30:
```

```
31:
32: // DEBUGF -
33: //     Macro which expands into trace code.  First argument is a
34: //     trace flag char, second argument is output code that can
35: //     be sandwiched between <<.  Beware of operator precedence.
36: //     Example:
37: //         DEBUGF ('u', "foo = " << foo);
38: //     will print two words and a newline if flag 'u' is on.
39: //     Traces are preceded by filename, line number, and function.
40:
41: #ifdef NDEBUG
42: #define DEBUGF(FLAG, CODE) ;
43: #define DEBUGS(FLAG, STMT) ;
44: #else
45: #define DEBUGF(FLAG, CODE) { \
46:     if (debugflags::getflag (FLAG)) { \
47:         debugflags::where (FLAG, __FILE__, __LINE__, \
48:             __PRETTY_FUNCTION__); \
49:         cerr << CODE << endl; \
50:     } \
51: }
52: #define DEBUGS(FLAG, STMT) { \
53:     if (debugflags::getflag (FLAG)) { \
54:         debugflags::where (FLAG, __FILE__, __LINE__, \
55:             __PRETTY_FUNCTION__); \
56:         STMT; \
57:     } \
58: }
59: #endif
60:
61: #endif
62:
```

```
1: // $Id: debug.cpp,v 1.3 2021-11-08 00:01:44-08 - - $
2:
3: #include <climits>
4: #include <iostream>
5: #include <vector>
6:
7: using namespace std;
8:
9: #include "debug.h"
10:
11: debugflags::flagset_ debugflags::flags_ {};
12:
13: void debugflags::setflags (const string& initflags) {
14:     for (const unsigned char flag: initflags) {
15:         if (flag == '@') flags_.set();
16:         else flags_.set (flag, true);
17:     }
18: }
19:
20: // getflag -
21: //     Check to see if a certain flag is on.
22:
23: bool debugflags::getflag (char flag) {
24:     // WARNING: Don't TRACE this function or the stack will blow up.
25:     return flags_.test (static_cast<unsigned char> (flag));
26: }
27:
28: void debugflags::where (char flag, const char* file, int line,
29:                        const char* pretty_function) {
30:     cout << "DEBUG(" << flag << ") "
31:          << file << "[" << line << "]" " << endl
32:          << "... " << pretty_function << endl;
33: }
34:
```

```
1: // $Id: cxi.cpp,v 1.6 2021-11-08 00:01:44-08 - - $
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 "debug.h"
15: #include "logstream.h"
16: #include "protocol.h"
17: #include "socket.h"
18:
19: logstream outlog (cout);
20: struct cxi_exit: public exception {};
21:
22: unordered_map<string,cxi_command> command_map {
23:     {"exit", cxi_command::EXIT},
24:     {"help", cxi_command::HELP},
25:     {"ls" , cxi_command::LS },
26: };
27:
28: static const char help[] = R"|(
29: exit          - Exit the program.  Equivalent to EOF.
30: get filename  - Copy remote file to local host.
31: help         - Print help summary.
32: ls           - List names of files on remote server.
33: put filename  - Copy local file to remote host.
34: rm filename   - Remove file from remote server.
35: )|";
36:
37: void cxi_help() {
38:     cout << help;
39: }
40:
41: void cxi_ls (client_socket& server) {
42:     cxi_header header;
43:     header.command = cxi_command::LS;
44:     DEBUGF ('h', "sending header " << header << endl);
45:     send_packet (server, &header, sizeof header);
46:     recv_packet (server, &header, sizeof header);
47:     DEBUGF ('h', "received header " << header << endl);
48:     if (header.command != cxi_command::LSOUT) {
49:         outlog << "sent LS, server did not return LSOUT" << endl;
50:         outlog << "server returned " << header << endl;
51:     }else {
52:         size_t host_nbytes = ntohl (header.nbytes);
53:         auto buffer = make_unique<char[]> (host_nbytes + 1);
54:         recv_packet (server, buffer.get(), host_nbytes);
55:         DEBUGF ('h', "received " << host_nbytes << " bytes");
56:         buffer[host_nbytes] = '\0';
57:         cout << buffer.get();
58:     }
```


11/16/21
13:28:07

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

2/4

```
59: }  
60:
```

```
61:
62: void usage() {
63:     cerr << "Usage: " << outlog.execname() << " host port" << endl;
64:     throw cxi_exit();
65: }
66:
67: pair<string,in_port_t> scan_options (int argc, char** argv) {
68:     for (;;) {
69:         int opt = getopt (argc, argv, "@:");
70:         if (opt == EOF) break;
71:         switch (opt) {
72:             case '@': debugflags::setflags (optarg);
73:                 break;
74:         }
75:     }
76:     if (argc - optind != 2) usage();
77:     string host = argv[optind];
78:     in_port_t port = get_cxi_server_port (argv[optind + 1]);
79:     return {host, port};
80: }
81:
82: int main (int argc, char** argv) {
83:     outlog.execname (basename (argv[0]));
84:     outlog << to_string (hostinfo()) << endl;
85:     try {
86:         auto host_port = scan_options (argc, argv);
87:         string host = host_port.first;
88:         in_port_t port = host_port.second;
89:         outlog << "connecting to " << host << " port " << port << endl;
90:         client_socket server (host, port);
91:         outlog << "connected to " << to_string (server) << endl;
92:         for (;;) {
93:             string line;
94:             getline (cin, line);
95:             if (cin.eof()) throw cxi_exit();
96:             outlog << "command " << line << endl;
97:             const auto& itor = command_map.find (line);
98:             cxi_command cmd = itor == command_map.end()
99:                 ? cxi_command::ERROR : itor->second;
100:             switch (cmd) {
101:                 case cxi_command::EXIT:
102:                     throw cxi_exit();
103:                     break;
104:                 case cxi_command::HELP:
105:                     cxi_help();
106:                     break;
107:                 case cxi_command::LS:
108:                     cxi_ls (server);
109:                     break;
110:                 default:
111:                     outlog << line << ": invalid command" << endl;
112:                     break;
113:             }
114:         }
115:     } catch (socket_error& error) {
116:         outlog << error.what() << endl;
117:     } catch (cxi_exit& error) {
118:         DEBUGF ('x', "caught cxi_exit");
```

11/16/21
13:28:07

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

4/4

```
119:    }  
120:    return 0;  
121: }  
122:
```

```
1: // $Id: cxid.cpp,v 1.9 2021-11-08 00:01:44-08 - - $
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 "debug.h"
13: #include "logstream.h"
14: #include "protocol.h"
15: #include "socket.h"
16:
17: logstream outlog (cout);
18: struct cxi_exit: public exception {};
19:
20: void reply_ls (accepted_socket& client_sock, cxi_header& header) {
21:     static const char ls_cmd[] = "ls -l 2>&1";
22:     FILE* ls_pipe = popen (ls_cmd, "r");
23:     if (ls_pipe == nullptr) {
24:         outlog << ls_cmd << ": " << strerror (errno) << endl;
25:         header.command = cxi_command::NAK;
26:         header.nbytes = htonl (errno);
27:         send_packet (client_sock, &header, sizeof header);
28:         return;
29:     }
30:     string ls_output;
31:     char buffer[0x1000];
32:     for (;;) {
33:         char* rc = fgets (buffer, sizeof buffer, ls_pipe);
34:         if (rc == nullptr) break;
35:         ls_output.append (buffer);
36:     }
37:     pclose (ls_pipe);
38:     header.command = cxi_command::LSOUT;
39:     header.nbytes = htonl (ls_output.size());
40:     memset (header.filename, 0, FILENAME_SIZE);
41:     DEBUGF ('h', "sending header " << header);
42:     send_packet (client_sock, &header, sizeof header);
43:     send_packet (client_sock, ls_output.c_str(), ls_output.size());
44:     DEBUGF ('h', "sent " << ls_output.size() << " bytes");
45: }
46:
```

```
47:
48: void run_server (accepted_socket& client_sock) {
49:     outlog.execname (outlog.execname() + "*");
50:     outlog << "connected to " << to_string (client_sock) << endl;
51:     try {
52:         for (;;) {
53:             cxi_header header;
54:             recv_packet (client_sock, &header, sizeof header);
55:             DEBUGF ('h', "received header " << header);
56:             switch (header.command) {
57:                 case cxi_command::LS:
58:                     reply_ls (client_sock, header);
59:                     break;
60:             default:
61:                 outlog << "invalid client header:" << header << endl;
62:                 break;
63:             }
64:         }
65:     } catch (socket_error& error) {
66:         outlog << error.what() << endl;
67:     } catch (cxi_exit& error) {
68:         DEBUGF ('x', "caught cxi_exit");
69:     }
70:     throw cxi_exit();
71: }
72:
73: void fork_cxiserver (server_socket& server, accepted_socket& accept) {
74:     pid_t pid = fork();
75:     if (pid == 0) { // child
76:         server.close();
77:         run_server (accept);
78:         throw cxi_exit();
79:     } else {
80:         accept.close();
81:         if (pid < 0) {
82:             outlog << "fork failed: " << strerror (errno) << endl;
83:         } else {
84:             outlog << "forked cxiserver pid " << pid << endl;
85:         }
86:     }
87: }
88:
```

```
89:
90: void reap_zombies() {
91:     for (;;) {
92:         int status;
93:         pid_t child = waitpid (-1, &status, WNOHANG);
94:         if (child <= 0) break;
95:         if (status != 0) {
96:             outlog << "child " << child
97:                 << " exit " << (status >> 8)
98:                 << " signal " << (status & 0x7F)
99:                 << " core " << (status >> 7 & 1) << endl;
100:        }
101:    }
102: }
103:
104: void signal_handler (int signal) {
105:     DEBUGF ('s', "signal_handler: caught " << strsignal (signal));
106:     reap_zombies();
107: }
108:
109: void signal_action (int signal, void (*handler) (int)) {
110:     struct sigaction action;
111:     action.sa_handler = handler;
112:     sigfillset (&action.sa_mask);
113:     action.sa_flags = 0;
114:     int rc = sigaction (signal, &action, nullptr);
115:     if (rc < 0) outlog << "sigaction " << strsignal (signal)
116:         << " failed: " << strerror (errno) << endl;
117: }
118:
```

```
119:
120:
121: void usage() {
122:     cerr << "Usage: " << outlog.execname() << " port" << endl;
123:     throw cxi_exit();
124: }
125:
126: in_port_t scan_options (int argc, char** argv) {
127:     for (;;) {
128:         int opt = getopt (argc, argv, "@:");
129:         if (opt == EOF) break;
130:         switch (opt) {
131:             case '@': debugflags::setflags (optarg);
132:                 break;
133:         }
134:     }
135:     if (argc - optind != 1) usage();
136:     return get_cxi_server_port (argv[optind]);
137: }
138:
139: int main (int argc, char** argv) {
140:     outlog.execname (basename (argv[0]));
141:     signal_action (SIGCHLD, signal_handler);
142:     try {
143:         in_port_t port = scan_options (argc, argv);
144:         server_socket listener (port);
145:         for (;;) {
146:             outlog << to_string (hostinfo()) << " accepting port "
147:                 << to_string (port) << endl;
148:             accepted_socket client_sock;
149:             for (;;) {
150:                 try {
151:                     listener.accept (client_sock);
152:                     break;
153:                 } catch (socket_sys_error& error) {
154:                     switch (error.sys_errno) {
155:                         case EINTR:
156:                             outlog << "listener.accept caught "
157:                                 << strerror (EINTR) << endl;
158:                             break;
159:                         default:
160:                             throw;
161:                     }
162:                 }
163:             }
164:             outlog << "accepted " << to_string (client_sock) << endl;
165:             try {
166:                 fork_cxiserver (listener, client_sock);
167:                 reap_zombies();
168:             } catch (socket_error& error) {
169:                 outlog << error.what() << endl;
170:             }
171:         }
172:     } catch (socket_error& error) {
173:         outlog << error.what() << endl;
174:     } catch (cxi_exit& error) {
175:         DEBUGF ('x', "caught cxi_exit");
176:     }
```

11/16/21
13:28:07

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

5/5

```
177:     return 0;  
178: }  
179:
```



```
1: # $Id: Makefile,v 1.22 2021-11-08 00:01:44-08 - - $
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++2a -g -O0 ${GPPOPTS}
12: MAKEDEPCPP  = g++ -std=gnu++2a -MM ${GPPOPTS}
13: UTILBIN     = /afs/cats.ucsc.edu/courses/cse111-wm/bin
14:
15: MODULES     = logstream protocol socket debug
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 socket.h
3: socket.o: socket.cpp socket.h
4: debug.o: debug.cpp debug.h
5: cxi.o: cxi.cpp debug.h logstream.h protocol.h socket.h
6: cxid.o: cxid.cpp debug.h logstream.h protocol.h socket.h
```