```
 1: // $Id: commands.h,v 1.12 2019-10-27 20:59:20-07 - - $
 2:
 3: #ifndef __COMMANDS_H__
 4: #define __COMMANDS_H__
 5:
 6: #include <unordered_map>
 7: using namespace std;
 8:
 9: #include "file_sys.h"
10: #include "util.h"
11:
12: // A couple of convenient usings to avoid verbosity.
13:
14: using command_fn = void (*)(inode_state& state, const wordvec& words);
15: using command_hash = unordered_map<string,command_fn>;
16:
17: // execution functions -
18:
19: void fn_cat    (inode_state& state, const wordvec& words);//
20: void fn_cd     (inode_state& state, const wordvec& words);//
21: void fn_echo   (inode_state& state, const wordvec& words);
22: void fn_exit   (inode_state& state, const wordvec& words);
23: void fn_ls     (inode_state& state, const wordvec& words);//
24: void fn_lsr    (inode_state& state, const wordvec& words);
25: void fn_make   (inode_state& state, const wordvec& words);//
26: void fn_mkdir  (inode_state& state, const wordvec& words);//
27: void fn_prompt (inode_state& state, const wordvec& words);//
28: void fn_pwd    (inode_state& state, const wordvec& words);//
29: void fn_rm     (inode_state& state, const wordvec& words);
30: void fn_rmr    (inode_state& state, const wordvec& words);
31: void fn_nothing (inode_state& state, const wordvec& words);
32:
33: command_fn find_command_fn (const string& command);
34:
35: // exit_status_message -
36: //    Prints an exit message and returns the exit status, as recorded
37: //    by any of the functions.
38:
39: int exit_status_message();
40: class ysh_exit: public exception {};
41:
42: #endif
```

```
 1: // $Id: commands.cpp,v 1.19 2019-10-27 20:59:20-07 - - $
 2:
 3: #include "commands.h"
 4: #include "debug.h"
 5:
 6: command_hash cmd_hash {
 7:    {"cat"   , fn_cat   },
 8:    {"cd"    , fn_cd    },
 9:    {"echo"  , fn_echo  },
10:    {"exit"  , fn_exit  },
11:    {"ls"    , fn_ls    },
12:    {"lsr"   , fn_lsr   },
13:    {"make"  , fn_make  },
14:    {"mkdir" , fn_mkdir },
15:    {"prompt", fn_prompt},
16:    {"pwd"   , fn_pwd   },
17:    {"rm"    , fn_rm    },
18:    {"rmr"   , fn_rmr   },
19:    {"#"     , fn_nothing},
20: };
21:
22: command_fn find_command_fn (const string& cmd) {
23:    // Note: value_type is pair<const key_type, mapped_type>
24:    // So: iterator->first is key_type (string)
25:    // So: iterator->second is mapped_type (command_fn)
26:    DEBUGF ('c', "[" << cmd << "]");
27:    const auto result = cmd_hash.find (cmd);
28:    if (result == cmd_hash.end()) {
29:       throw command_error (cmd + ": no such function");
30:    }
31:    return result->second;
32: }
33:
34: int exit_status_message() {
35:    int status = exec::status();
36:    cout << exec::execname() << ": exit(" << status << ")" << endl;
37:
38:    return status;
39: }
40:
41: void fn_nothing(inode_state& state, const wordvec& words){
42:    DEBUGF ('c', state);
43:    DEBUGF ('c', words);
44: }
45:
46: void fn_cat (inode_state& state, const wordvec& words){
47:   if(words.size()==1){
48:       err_print("cat", " ", "Incorrect number of arguments");
49:       exec::status(1);
50:   }else{
51:     for(size_t i=1;i<words.size();i++){
52:       try{
53:         state.cat(words[i]);
54:       }catch(command_error& error){
55:         exec::status(1);
56:       }
57:      }
58:    }
```

```
   59:    DEBUGF ('c', state);
   60:    DEBUGF ('c', words);
   61: }
   62:
   63: void fn_cd (inode_state& state, const wordvec& words){
   64:    if(words.size()==1){
   65:       state.cd ("/");
   66:    }
   67:    else if(words.size()>2){
   68:        err_print("cd", words[1], "Incorrect number of arguments");
   69:        exec::status(1);
   70:    }else if (state.cd (words[1])){
   71:        exec::status(1);
   72:    }
   73:    DEBUGF ('c', state);
   74:    DEBUGF ('c', words);
   75: }
   76:
   77: void fn_echo (inode_state& state, const wordvec& words){
   78:    DEBUGF ('c', state);
   79:    DEBUGF ('c', words);
   80:    cout << word_range (words.cbegin() + 1, words.cend()) << endl;
   81: }
   82:
```

```
 83:
 84: void fn_exit (inode_state& state, const wordvec& words){
 85:    DEBUGF ('c', state);
 86:    DEBUGF ('c', words);
 87:    if(words.size() > 1)
 88:     exec::status(stoi(words[1]));
 89:    throw ysh_exit();
 90: }
 91:
 92: void fn_ls (inode_state& state, const wordvec& words){
 93:    if(words.size()==1)
 94:       state.lsprint(".");
 95:    else
 96:      for(size_t i=1;i<words.size();i++)
 97:         if(state.lsprint(words[i]))
 98:             exec::status(1);
 99:    DEBUGF ('c', state);
100:    DEBUGF ('c', words);
101: }
102:
103: void fn_lsr (inode_state& state, const wordvec& words){
104:    if(words.size()==1)
105:       state.lsr("");
106:    else
107:      for(size_t i=1;i<words.size();i++)
108:         state.lsr(words[i]);
109:
110:    DEBUGF ('c', state);
111:    DEBUGF ('c', words);
112: }
113:
114: void fn_make (inode_state& state, const wordvec& words){
115:    if(words.size()<2){
116:        err_print("make", " ", "Incorrect number of arguments");
117:        exec::status(1);
118:    }else{
119:        string text= ("");
120:        for(size_t i=2;i<words.size();i++){
121:          text.append(words[i]);
122:          if(i!=words.size()-1)
123:              text.append(" ");
124:
125:      }
126:
127:      if(state.make_file(words[1],text)){
128:        exec::status(1);
129:
130:      }
131:    }
132:    DEBUGF ('c', state);
133:    DEBUGF ('c', words);
134: }
135:
136: void fn_mkdir (inode_state& state, const wordvec& words){
137:    if( words.size() != 2){
138:      if(words.size()==1)
139:        err_print("mkdir", " ", "Incorrect number of arguments");
140:      else
```

```
141:        err_print("mkdir", words[1], "Incorrect number of arguments");
142:       exec::status(1);
143:    }else if (state.mkdir_at_cwd(words[1])){
144:       exec::status(1);
145:    }
146:    DEBUGF ('c', state);
147:    DEBUGF ('c', words);
148: }
149:
150: void fn_prompt (inode_state& state, const wordvec& words){
151:    string text= ("");
152:    for(size_t i=1;i<words.size();i++){
153:       text.append(words[i]);
154:       if(i!=words.size()-1)
155:          text.append(" ");
156:    }
157:    state.setprompt(text+" ");
158:    DEBUGF ('c', state);
159:    DEBUGF ('c', words);
160: }
161:
162: void fn_pwd (inode_state& state, const wordvec& words){
163:    if(words.size()>1){
164:      exec::status(1);
165:      err_print("pwd", words[1], "Incorrect number of arguments");
166:    }else if(state.pwd()){
167:      exec::status(1);
168:    }
169:    DEBUGF ('c', state);
170:    DEBUGF ('c', words);
171: }
172:
173: void fn_rm (inode_state& state, const wordvec& words){
174:    if(words.size()!=2){
175:       cerr << "Function rm called with invalid number of args";
176:       return;
177:    }
178:    state.rm(words[1]);
179:    DEBUGF ('c', state);
180:    DEBUGF ('c', words);
181: }
182:
183: void fn_rmr (inode_state& state, const wordvec& words){
184:   if(words.size()!=2){
185:       cerr << "Function rmr called with invalid number of args";
186:       return;
187:    }
188:    state.rmr(words[1]);
189:    DEBUGF ('c', state);
190:    DEBUGF ('c', words);
191: }
```

```
 1: // $Id: debug.h,v 1.11 2019-10-08 13:46:59-07 - - $
 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 1.14 2019-10-27 20:59:20-07 - - $
 2:
 3: #include <climits>
 4: #include <iostream>
 5: #include <vector>
 6:
 7: using namespace std;
 8:
 9: #include "debug.h"
10: #include "util.h"
11:
12: debugflags::flagset debugflags::flags {};
13:
14: void debugflags::setflags (const string& initflags) {
15:    for (const unsigned char flag: initflags) {
16:       if (flag == '@') flags.set();
17:                   else flags.set (flag, true);
18:    }
19: }
20:
21: // getflag -
22: //    Check to see if a certain flag is on.
23:
24: bool debugflags::getflag (char flag) {
25:    // WARNING: Don't TRACE this function or the stack will blow up.
26:    return flags.test (static_cast<unsigned char> (flag));
27: }
28:
29: void debugflags::where (char flag, const char* file, int line,
30:                         const char* pretty_function) {
31:    cout << exec::execname() << ": DEBUG(" << flag << ") "
32:         << file << "[" << line << "] " << pretty_function << endl;
33: }
34:
```

```
 1: // $Id: file_sys.h,v 1.11 2019-10-27 21:05:44-07 - - $
 2:
 3: #ifndef __INODE_H__
 4: #define __INODE_H__
 5:
 6: #include <exception>
 7: #include <iostream>
 8: #include <memory>
 9: #include <map>
10: #include <vector>
11: using namespace std;
12: #include "util.h"
13:
14: // inode_t -
15: //    An inode is either a directory or a plain file.
16:
17: enum class file_type {PLAIN_TYPE, DIRECTORY_TYPE};
18: class inode;
19: class base_file;
20: class plain_file;
21: class directory;
22: using inode_ptr = shared_ptr<inode>;
23: using base_file_ptr = shared_ptr<base_file>;
24: using directory_ptr = shared_ptr<directory>;
25: using file_ptr = shared_ptr<plain_file>;
26: ostream& operator<< (ostream&, file_type);
27:
28: void err_print(const string& cmdname,
29: const string& argname, const string& errname);
30:
```

```
31:
32: // inode_state -
33: //    A small convenient class to maintain the state of the simulated
34: //    process:  the root (/), the current directory (.), and the
35: //    prompt.
36:
37: class inode_state {
38:    friend class inode;
39:    friend class plain_file;
40:    friend class base_file;
41:    friend class directory;
42:    friend ostream& operator<< (ostream& out, const inode_state&);
43:    private:
44:       inode_ptr root {nullptr};
45:       inode_ptr cwd {nullptr};
46:       string prompt_ {"% "};
47:       void rmrecur(inode_ptr target);
48:       void lsrecur(const string& path);
49:    public:
50:       ~inode_state();
51:       inode_state (const inode_state&) = delete; // copy ctor
52:       inode_state& operator= (const inode_state&) = delete; // op=
53:       inode_state();
54:       void rmr(const string& path);
55:       void rm(const string& path);
56:       bool lsprint(const string& path);
57:       void lsr(const string& path);
58:       bool pwd();
59:       bool mkdir_at_cwd(const string& filename);
60:       bool make_file(const string& filename, const string& text);
61:       bool cat(const string& filename);
62:       bool cd(const string& path);
63:       inode_ptr pathDecode(const string& path);
64:       const string& prompt() const;
65:       void setprompt(const string& newprompt);
66: };
67:
68: // class inode -
69: // inode ctor -
70: //    Create a new inode of the given type.
71: // get_inode_nr -
72: //    Retrieves the serial number of the inode.  Inode numbers are
73: //    allocated in sequence by small integer.
74: // size -
75: //    Returns the size of an inode.  For a directory, this is the
76: //    number of dirents.  For a text file, the number of characters
77: //    when printed (the sum of the lengths of each word, plus the
78: //    number of words.
79: //
80:
81: class inode {
82:    friend class inode_state;
83:    friend class plain_file;
84:    friend class base_file;
85:    friend class directory;
86:    private:
87:       static int next_inode_nr;
88:       int inode_nr;
```

```
89:       base_file_ptr contents;
90:       string fileType;
91:    public:
92:       inode (file_type);
93:       int get_inode_nr() const;
94:       int size() const;
95:       string type() const;
96: };
97:
```

```
 98:
 99: // class base_file -
100: // Just a base class at which an inode can point.  No data or
101: // functions.  Makes the synthesized members useable only from
102: // the derived classes.
103:
104: class base_file {
105:    friend class inode_state;
106:    friend class inode;
107:    friend class plain_file;
108:    friend class directory;
109:    protected:
110:       base_file() = default;
111:       virtual const string error_file_type() const = 0;
112:    public:
113:       virtual ~base_file() = default;
114:       base_file (const base_file&) = delete;
115:       base_file& operator= (const base_file&) = delete;
116:       virtual size_t size() const = 0;
117:       virtual const string& readfile() const;
118:       virtual void writefile (const string& newdata);
119:       virtual void remove (const string& filename);
120:       virtual inode_ptr mkdir (const string& dirname);
121:       virtual inode_ptr mkfile (const string& filename);
122: };
123:
124: class file_error: public runtime_error {
125:    public:
126:       explicit file_error (const string& what);
127: };
128:
```

```
129:
130: // class plain_file -
131: // Used to hold data.
132: // synthesized default ctor -
133: //    Default vector<string> is a an empty vector.
134: // readfile -
135: //    Returns a copy of the contents of the wordvec in the file.
136: // writefile -
137: //    Replaces the contents of a file with new contents.
138:
139: class plain_file: public base_file {
140:    friend class inode_state;
141:    friend class inode;
142:    friend class base_file;
143:    friend class directory;
144:    private:
145:       string data;
146:       virtual const string error_file_type() const override {
147:          return "plain file";
148:       }
149:    public:
150:       virtual size_t size() const override;
151:       virtual const string& readfile() const override;
152:       virtual void writefile (const string& newdata) override;
153: };
154:
155: // class directory -
156: // Used to map filenames onto inode pointers.
157: // default ctor -
158: //    Creates a new map with keys "." and "..".
159: // remove -
160: //    Removes the file or subdirectory from the current inode.
161: //    Throws an file_error if this is not a directory, the file
162: //    does not exist, or the subdirectory is not empty.
163: //    Here empty means the only entries are dot (.) and dotdot (..).
164: // mkdir -
165: //    Creates a new directory under the current directory and
166: //    immediately adds the directories dot (.) and dotdot (..) to it.
167: //    Note that the parent (..) of / is / itself.  It is an error
168: //    if the entry already exists.
169: // mkfile -
170: //    Create a new empty text file with the given name.  Error if
171: //    a dirent with that name exists.
172:
173: class directory: public base_file {
174:    friend class inode_state;
175:    friend class inode;
176:    friend class plain_file;
177:    friend class base_file;
178:    private:
179:       // Must be a map, not unordered_map, so printing is lexicographic
180:       map<string,inode_ptr> dirents;
181:       virtual const string error_file_type() const override {
182:          return "directory";
183:       }
184:    public:
185:       virtual size_t size() const override;
186:       virtual void remove (const string& filename) override;
```

```
187:       virtual inode_ptr mkdir (const string& dirname) override;
188:       virtual inode_ptr mkfile (const string& filename) override;
189: };
190:
191: #endif
```

```
 1: // $Id: file_sys.cpp,v 1.8 2019-10-27 20:59:20-07 - - $
 2:
 3: #include <iostream>
 4: #include <stdexcept>
 5: #include <unordered_map>
 6:
 7: using namespace std;
 8:
 9: #include "debug.h"
10: #include "file_sys.h"
11: #include <cstring>
12: int inode::next_inode_nr {1};
13:
14: struct file_type_hash {
15:    size_t operator() (file_type type) const {
16:       return static_cast<size_t> (type);
17:    }
18: };
19:
20: void err_print(const string& cmdname,
21: const string& argname, const string& errname){
22:   throw command_error (cmdname + ": " + argname + ": " + errname);
23:   // cerr << cmdname << ": " << argname << ": " << errname << endl;
24: }
25:
26: void inode_state::setprompt(const string& newprompt){
27:   prompt_ =newprompt;
28: }
29:
30: ostream& operator<< (ostream& out, file_type type) {
31:    static unordered_map<file_type,string,file_type_hash> hash {
32:       {file_type::PLAIN_TYPE, "PLAIN_TYPE"},
33:       {file_type::DIRECTORY_TYPE, "DIRECTORY_TYPE"},
34:    };
35:    return out << hash[type];
36: }
37:
38: inode_state::inode_state() {
39:    root= make_shared<inode>(file_type::DIRECTORY_TYPE);
40:    directory_ptr dir = dynamic_pointer_cast<directory>(root->contents);
41:    dir -> dirents.insert(pair<string,inode_ptr>(".",root));
42:    dir -> dirents.insert(pair<string,inode_ptr>("..",root));
43:    cwd=root;
44:
45:    DEBUGF ('i', "root = " << root << ", cwd = " << cwd
46:           << ", prompt = \"" << prompt() << "\"");
47: }
48:
49: inode_state::~inode_state() {
50:    rmrecur(root);
51:    root = nullptr;
52:    cwd = nullptr;
53: }
54:
55: bool inode_state::lsprint(const string& path){
56:
57:    inode_ptr temp = pathDecode(path);
58:
```

```
   59:    directory_ptr dir= dynamic_pointer_cast<directory>(temp->contents);
   60:    string a = ("");
   61:    while(dir->dirents.at("..")!=temp){
   62:       inode_ptr temp2 =dir->dirents.at("..");
   63:       directory_ptr dir2 =
   64:       dynamic_pointer_cast<directory>(temp2->contents);
   65:       for (auto it=dir2->dirents.begin(); it!=dir2->dirents.end(); ++it)
   66:         if(it->second==temp){
   67:                a.insert(0,it->first);
   68:                a.insert(0,"/");
   69:             }
   70:           temp = dir->dirents.at("..");
   71:           dir= dynamic_pointer_cast<directory>(temp->contents);
   72:    }
   73:    if(a.length()>0){
   74:        cout <<a<<":"<<endl;
   75:    }else{
   76:        cout <<"/"<<":"<<endl;
   77:    }
   78:     inode_ptr result_path;
   79:    if(path.compare(".")==0||path.length()==0){
   80:       result_path= cwd;
   81:    }else{
   82:       result_path= pathDecode(path);
   83:    }
   84:    if(result_path ==nullptr){
   85:      err_print("ls", path, "Path not found");
   86:      return true;
   87:    }
   88:    // Printing everything in the directory
   89:    directory_ptr dir3 =
   90:    dynamic_pointer_cast<directory>(result_path->contents);
   91:    for (auto it=dir3->dirents.begin(); it!=dir3->dirents.end(); ++it){
   92:
   93:        cout << "    "<< it -> second-> inode_nr
   94:        << "    " << it -> second->size() << "  ";
   95:        cout << it->first;
   96:        if(it -> second-> contents->error_file_type()==
   97:        "directory"&&(it->first!="."&&it->first!=".." )){
   98:          cout << "/";
   99:        }
  100:        cout << '\n';
  101:     }
  102:    return false;
  103: }
  104:
  105: void inode_state::lsr(const string& path){
  106:     inode_ptr temp;
  107:     if(path.find('/') == string::npos)
  108:         temp = cwd;
  109:     else
  110:         temp = pathDecode(path);
  111:     // Building absolute path to the target directory
  112:    directory_ptr dir = dynamic_pointer_cast<directory>(temp->contents);
  113:    string absPath = ("");
  114:    while(dir->dirents.at("..") != temp){
  115:       inode_ptr temp2 = dir->dirents.at("..");
  116:       directory_ptr dir2 =
```

```
117:       dynamic_pointer_cast<directory>(temp2->contents);
118:       for (auto it = dir2->dirents.begin();
119:       it != dir2->dirents.end(); ++it)
120:         if(it->second == temp){
121:                absPath.insert(0,it->first);
122:                absPath.insert(0,"/");
123:             }
124:           temp = dir->dirents.at("..");
125:           dir = dynamic_pointer_cast<directory>(temp->contents);
126:    }
127:    if(absPath.empty())
128:       absPath = "/" + path;
129:    lsrecur(absPath);
130: }
131:
132: void inode_state::lsrecur(const string& path){
133:     lsprint(path);
134:     inode_ptr target = pathDecode(path);
135:     if((*target).size() <= 2){
136:         return;
137:     }
138:     directory_ptr dir =
139:     dynamic_pointer_cast<directory>(target->contents);
140:     auto it = dir->dirents.begin();
141:     ++it;
142:     ++it;
143:     for (; it != dir->dirents.end(); it++)
144:         // Grab the value from the iterator (inode_ptr) and recur
145:         if((*(it->second)).type() == "directory")
146:             lsrecur(path + "/" + it->first);
147: }
148:
149: void inode_state::rm(const string& path){
150:   // Get necessary pointers
151:   inode_ptr parent = cwd;
152:   string filename = path;
153:    if(path.find('/') != string::npos){
154:       size_t index_of_last = filename.find_last_of("/");
155:       filename = path.substr(index_of_last + 1);
156:       parent = pathDecode(filename.substr(0,index_of_last));
157:    }
158:    inode_ptr target = pathDecode(path);
159:    // Error checking
160:    if(target == nullptr){
161:     err_print("rm", path, "Target not found");
162:     return;
163:    }
164:    else if(target == parent){
165:     err_print("rm", path, "Called on root");
166:     return;
167:    }
168:    else if((*target).type() == "directory" && (*target).size() > 2){
169:     err_print("rm", path, "Called on directory with children");
170:     return;
171:    }
172:    // Erase the reference
173:   directory_ptr dir = dynamic_pointer_cast<directory>(parent->contents);
174:   dir->dirents.erase(filename);
```

```
175: }
176:
177: void inode_state::rmr(const string& path){
178:     // Get necessary pointers
179:   inode_ptr parent = cwd;
180:   string filename = path;
181:    if(path.find('/') != string::npos){
182:       size_t index_of_last = filename.find_last_of("/");
183:       filename = path.substr(index_of_last + 1);
184:       parent = pathDecode(filename.substr(0,index_of_last));
185:    }
186:    inode_ptr target = pathDecode(path);
187:    // Error checking
188:    if(target == nullptr){
189:     err_print("rm", path, "Target not found");
190:     return;
191:    }
192:    else if(target == parent){
193:     err_print("rm", path, "Called on root");
194:     return;
195:    }
196:    rmrecur(target);
197:    // Erase the reference
198:    directory_ptr dir =
199:    dynamic_pointer_cast<directory>(parent->contents);
200:    dir->dirents.erase(filename);
201: }
202:
203: void inode_state::rmrecur(inode_ptr target){
204:   if((*target).type() == "directory" && (*target).size() > 2){
205:     directory_ptr dir =
206:     dynamic_pointer_cast<directory>(target->contents);
207:     auto next = dir->dirents.begin();
208:     ++next;
209:     ++next;
210:     auto current = next;
211:     while(next != dir->dirents.end()){
212:       ++next;
213:       rmrecur(current->second);
214:       dir->dirents.erase(current->first);
215:       current = next;
216:     }
217:     dir->dirents.erase("..");
218:     dir->dirents.erase(".");
219:   }
220: }
221:
222: bool inode_state::pwd(){
223:    inode_ptr temp = cwd;
224:    directory_ptr dir= dynamic_pointer_cast<directory>(temp->contents);
225:    string a= ("");
226:    while(dir->dirents.at("..")!=temp){
227:       inode_ptr temp2 =dir->dirents.at("..");
228:       directory_ptr dir2 =
229:       dynamic_pointer_cast<directory>(temp2->contents);
230:       for (auto it=dir2->dirents.begin(); it!=dir2->dirents.end(); ++it)
231:         if(it->second==temp){
232:                a.insert(0,it->first);
```

```
233:                a.insert(0,"/");
234:                  //cout <<it->first;
235:             }
236:          temp = dir->dirents.at("..");
237:          dir= dynamic_pointer_cast<directory>(temp->contents);
238:    }
239:    if(a.length()>0){
240:        cout <<a<<endl;
241:    }else{
242:        cout <<"/"<<endl;
243:    }
244:    return false;
245: }
246:
247: bool inode_state::make_file(const string& filename, const string& text){
248:    inode_ptr path= cwd;
249:    string file;
250:    if(filename.find("/") != string::npos){
251:       size_t index_of_last = filename.find_last_of("/");
252:       path = pathDecode(filename.substr(0,index_of_last));
253:       file =filename.substr(index_of_last+1);
254:       if(path ==nullptr){
255:       err_print("make", filename, "Path to file not found");
256:         return true;
257:       }
258:
259:    }else{
260:       file= filename;
261:    }
262:    directory_ptr dir = dynamic_pointer_cast<directory>(path->contents);
263:    if(dir->mkfile(file)==nullptr){
264:      err_print("make",filename,  "Path to file already exists");
265:      return true;
266:    }
267:    file_ptr ptr_to_file =
268:    dynamic_pointer_cast<plain_file>(dir->dirents.at(file)->contents);
269:
270:    ptr_to_file->writefile(text);
271:    return false;
272:
273: }
274:
275: bool inode_state::mkdir_at_cwd(const string& filename){
276:    inode_ptr path = cwd;
277:    string file;
278:    if(filename.find('/') != string::npos){
279:       size_t index_of_last = filename.find_last_of("/");
280:
281:       path = pathDecode(filename.substr(0,index_of_last));
282:       file =filename.substr(index_of_last+1);
283:
284:       if(path ==nullptr){
285:         err_print("mkdir", filename, "Path to directory not found");
286:         return true;
287:       }
288:    }else{
289:       file= filename;
290:    }
```

```
291:    directory_ptr dir= dynamic_pointer_cast<directory>(path->contents);
292:    if(dir->mkdir(file)==nullptr){
293:      err_print("mkdir", filename, "File already exists");
294:      return true;
295:    }
296:    return false;
297: }
298:
299: bool inode_state::cat(const string& filename){
300:    inode_ptr path = pathDecode(filename);
301:    if(path ==nullptr){
302:      err_print("cat",filename, "Path to file chosen not valid");
303:      return true;
304:    }
305:    if (path -> contents->error_file_type()=="plain file"){
306:       file_ptr ptr_to_file =
307:       dynamic_pointer_cast<plain_file>(path->contents);
308:       cout << ptr_to_file-> readfile()<<endl;
309:    }else{
310:     err_print("cat", filename, "Target is a directory");
311:      return true;
312:    }
313:    return false;
314: }
315:
316: inode_ptr inode_state::pathDecode(const string& path){
317:   inode_ptr ret = cwd;
318:
319:   if(path.at(0)==('/'))
320:     ret = root;
321:
322:     //absolutePath
323:   wordvec words = split (path, "/");
324:
325:
326:   bool foundall= true;
327:
328:   for(string& subdir : words){
329:      directory_ptr dir= dynamic_pointer_cast<directory>(ret->contents);
330:        try {
331:            ret = dir->dirents.at(subdir);
332:        }
333:        catch (const out_of_range& err) {
334:            foundall= false;
335:            break;
336:        }
337:   }
338:   if(!foundall){
339:     return nullptr;
340:   }
341:   return ret;
342: }
343:
344: bool inode_state::cd (const string& path){
345:    inode_ptr newCWD = pathDecode(path);
346:    if(newCWD==nullptr){
347:       err_print("cd", path, "Directory not found");
348:      return true;
```

```
349:    }else if((*newCWD).type() == "plain"){
350:       err_print("cd", path, "Target is a plain file");
351:       return true;
352:    }
353:    else {
354:      cwd= newCWD;
355:      return false;
356:    }
357: }
358:
359: const string& inode_state::prompt() const { return prompt_; }
360:
361: ostream& operator<< (ostream& out, const inode_state& state) {
362:    out << "inode_state: root = " << state.root
363:        << ", cwd = " << state.cwd;
364:    return out;
365: }
366:
367: inode::inode(file_type type): inode_nr (next_inode_nr++) {
368:    switch (type) {
369:       case file_type::PLAIN_TYPE:
370:            fileType = "plain";
371:            contents = make_shared<plain_file>();
372:            break;
373:       case file_type::DIRECTORY_TYPE:
374:            fileType = "directory";
375:            contents = make_shared<directory>();
376:            break;
377:    }
378:    DEBUGF ('i', "inode " << inode_nr << ", type = " << type);
379: }
380:
381: int inode::get_inode_nr() const {
382:    DEBUGF ('i', "inode = " << inode_nr);
383:    return inode_nr;
384: }
385:
386: int inode::size() const {
387:   DEBUGF ('i', "size = " << (*contents).size());
388:   return (*contents).size();
389: }
390:
391: string inode::type() const {
392:   DEBUGF ('i', "type = " << fileType);
393:   return fileType;
394: }
395:
```

```
396:
397: file_error::file_error (const string& what):
398:             runtime_error (what) {
399: }
400:
401: const string& base_file::readfile() const {
402:    throw file_error ("is a " + error_file_type());
403: }
404:
405: void base_file::writefile (const string&) {
406:    throw file_error ("is a " + error_file_type());
407: }
408:
409: void base_file::remove (const string&) {
410:    throw file_error ("is a " + error_file_type());
411: }
412:
413: inode_ptr base_file::mkdir (const string&) {
414:    throw file_error ("is a " + error_file_type());
415: }
416:
417: inode_ptr base_file::mkfile (const string&) {
418:    throw file_error ("is a " + error_file_type());
419: }
420:
```

```
421:
422: size_t plain_file::size() const {
423:    size_t size {data.length()};
424:    DEBUGF ('i', "size = " << size);
425:    return size;
426: }
427:
428: const string& plain_file::readfile() const {
429:    DEBUGF ('i', data);
430:    return data;
431: }
432:
433: void plain_file::writefile (const string& words) {
434:    data= words;
435:    DEBUGF ('i', words);
436: }
437:
438: size_t directory::size() const {
439:    size_t size {dirents.size()};
440:    DEBUGF ('i', "size = " << size);
441:    return size;
442: }
443:
444: void directory::remove (const string& filename) {
445:    DEBUGF ('i', filename);
446: }
447:
448: inode_ptr directory::mkdir (const string& dirname) {
449:    auto iter = dirents.find(dirname);
450:    if( iter!=dirents.end()){
451:       return nullptr;
452:    }
453:    inode_ptr node = make_shared<inode>(file_type::DIRECTORY_TYPE);
454:    base_file_ptr fil=node->contents;
455:    directory_ptr dir= dynamic_pointer_cast<directory>(fil);
456:    dir -> dirents.insert(pair<string,inode_ptr>(".",node));
457:    dir -> dirents.insert(pair<string,inode_ptr>("..",dirents.at(".")));
458:    dirents.insert(pair<string,inode_ptr>(dirname,node));
459:    DEBUGF ('i', dirname);
460:    return node;
461: }
462:
463: inode_ptr directory::mkfile (const string& filename) {
464:   auto iter = dirents.find(filename);
465:   if( iter!=dirents.end()){
466:      return nullptr;
467:   }
468:    inode_ptr node = make_shared<inode>(file_type::PLAIN_TYPE);
469:    base_file_ptr fil=node->contents;
470:    file_ptr file= dynamic_pointer_cast<plain_file>(fil);
471:    dirents.insert(pair<string,inode_ptr>(filename,node));
472:    DEBUGF ('i', filename);
473:    return node;
474: }
```

```
 1: // $Id: util.h,v 1.14 2019-10-27 20:59:20-07 - - $
 2:
 3: // util -
 4: //    A utility class to provide various services not conveniently
 5: //    included in other modules.
 6:
 7: #ifndef __UTIL_H__
 8: #define __UTIL_H__
 9:
10: #include <iostream>
11: #include <stdexcept>
12: #include <string>
13: #include <vector>
14: using namespace std;
15:
16: // Convenient type using to allow brevity of code elsewhere.
17:
18: template <typename iterator>
19: using range_type = pair<iterator,iterator>;
20:
21: using wordvec = vector<string>;
22: using word_range = range_type<decltype(declval<wordvec>().cbegin())>;
23:
24: // want_echo -
25: //    We want to echo all of cin to cout if either cin or cout
26: //    is not a tty.  This helps make batch processing easier by
27: //    making cout look like a terminal session trace.
28:
29: bool want_echo();
30:
31: //
32: // main -
33: //    Keep track of execname and exit status.  Must be initialized
34: //    as the first thing done inside main.  Main should call:
35: //       main::execname (argv[0]);
36: //    before anything else.
37: //
38:
39: class exec {
40:    private:
41:       static string execname_;
42:       static int status_;
43:       static void execname (const string& argv0);
44:       friend int main (int, char**);
45:    public:
46:       static void status (int status);
47:       static const string& execname() {return execname_; }
48:       static int status() {return status_; }
49: };
50:
```

```
 51:
 52: // split -
 53: //    Split a string into a wordvec (as defined above).  Any sequence
 54: //    of chars in the delimiter string is used as a separator.  To
 55: //    Split a pathname, use "/".  To split a shell command, use " ".
 56:
 57: wordvec split (const string& line, const string& delimiter);
 58:
 59: // complain -
 60: //    Used for starting error messages.  Sets the exit status to
 61: //    EXIT_FAILURE, writes the program name to cerr, and then
 62: //    returns the cerr ostream.  Example:
 63: //       complain() << filename << ": some problem" << endl;
 64:
 65: ostream& complain();
 66:
 67: // operator<< (vector) -
 68: //    An overloaded template operator which allows vectors to be
 69: //    printed out as a single operator, each element separated from
 70: //    the next with spaces.  The item_t must have an output operator
 71: //    defined for it.
 72:
 73: template <typename item_t>
 74: ostream& operator<< (ostream& out, const vector<item_t>& vec) {
 75:    string space = "";
 76:    for (const auto& item: vec) {
 77:       out << space << item;
 78:       space = " ";
 79:    }
 80:    return out;
 81: }
 82:
 83: template <typename iterator>
 84: ostream& operator<< (ostream& out, range_type<iterator> range) {
 85:    for (auto itor = range.first; itor != range.second; ++itor) {
 86:       if (itor != range.first) out << " ";
 87:       out << *itor;
 88:    }
 89:    return out;
 90: }
 91:
 92: // command_error -
 93: //    Extend runtime_error for throwing exceptions related to this
 94: //    program.
 95:
 96: class command_error: public runtime_error {
 97:    public:
 98:       explicit command_error (const string& what);
 99: };
100:
101: #endif
102:
```

```
 1: // $Id: util.cpp,v 1.15 2019-10-27 20:59:20-07 - - $
 2:
 3: #include <cstdlib>
 4: #include <unistd.h>
 5:
 6: using namespace std;
 7:
 8: #include "util.h"
 9: #include "debug.h"
10:
11: bool want_echo() {
12:    constexpr int CIN_FD {0};
13:    constexpr int COUT_FD {1};
14:    bool cin_is_not_a_tty = not isatty (CIN_FD);
15:    bool cout_is_not_a_tty = not isatty (COUT_FD);
16:    DEBUGF ('u', "cin_is_not_a_tty = " << cin_is_not_a_tty
17:           << ", cout_is_not_a_tty = " << cout_is_not_a_tty);
18:    return cin_is_not_a_tty or cout_is_not_a_tty;
19: }
20:
21: string exec::execname_; // Must be initialized from main().
22: int exec::status_ = EXIT_SUCCESS;
23:
24: string basename (const string &arg) {
25:    return arg.substr (arg.find_last_of ('/') + 1);
26: }
27:
28: void exec::execname (const string& argv0) {
29:    execname_ = basename (argv0);
30:    cout << boolalpha;
31:    cerr << boolalpha;
32:    DEBUGF ('u', "execname = " << execname_);
33: }
34:
35: void exec::status (int status) {
36:    if (status_ < status) status_ = status;
37: }
38:
```

```
39:
40: wordvec split (const string& line, const string& delimiters) {
41:    wordvec words;
42:    size_t end = 0;
43:
44:    // Loop over the string, splitting out words, and for each word
45:    // thus found, append it to the output wordvec.
46:    for (;;) {
47:       size_t start = line.find_first_not_of (delimiters, end);
48:       if (start == string::npos) break;
49:       end = line.find_first_of (delimiters, start);
50:       words.push_back (line.substr (start, end - start));
51:    }
52:    DEBUGF ('u', words);
53:    return words;
54: }
55:
56: ostream& complain() {
57:    exec::status (EXIT_FAILURE);
58:    cerr << exec::execname() << ": ";
59:    return cerr;
60: }
61:
62: command_error::command_error (const string& what):
63:             runtime_error (what) {
64: }
```

```
 1: // $Id: main.cpp,v 1.11 2019-10-27 20:59:20-07 - - $
 2:
 3: #include <cstdlib>
 4: #include <iostream>
 5: #include <string>
 6: #include <utility>
 7: #include <unistd.h>
 8:
 9: using namespace std;
10:
11: #include "commands.h"
12: #include "debug.h"
13: #include "file_sys.h"
14: #include "util.h"
15:
16: // scan_options
17: //    Options analysis:  The only option is -Dflags.
18:
19: void scan_options (int argc, char** argv) {
20:    opterr = 0;
21:    for (;;) {
22:       int option = getopt (argc, argv, "@:");
23:       if (option == EOF) break;
24:       switch (option) {
25:          case '@':
26:             debugflags::setflags (optarg);
27:             break;
28:          default:
29:             complain() << "-" << static_cast<char> (option)
30:                        << ": invalid option" << endl;
31:             break;
32:       }
33:    }
34:    if (optind < argc) {
35:       complain() << "operands not permitted" << endl;
36:    }
37: }
38:
```

```
39:
40: // main -
41: //    Main program which loops reading commands until end of file.
42:
43: int main (int argc, char** argv) {
44:    exec::execname (argv[0]);
45:    cout << boolalpha;  // Print false or true instead of 0 or 1.
46:    cerr << boolalpha;
47:    cout << argv[0] << " build " << __DATE__ << " " << __TIME__ << endl;
48:    scan_options (argc, argv);
49:    bool need_echo = want_echo();
50:    inode_state state;
51:    try {
52:       for (;;) {
53:          try {
54:             // Read a line, break at EOF, and echo print the prompt
55:             // if one is needed.
56:             cout << state.prompt();
57:             string line;
58:             getline (cin, line);
59:             if (cin.eof()) {
60:
61:                if (need_echo) cout << "^D";
62:                cout << endl;
63:                DEBUGF ('y', "EOF");
64:                break;
65:             }
66:             if (need_echo) cout << line << endl;
67:
68:             // Split the line into words and lookup the appropriate
69:             // function.  Complain or call it.
70:             wordvec words = split (line, " \t");
71:             DEBUGF ('y', "words = " << words);
72:             command_fn fn = find_command_fn (words.at(0));
73:             fn (state, words);
74:          }catch (command_error& error) {
75:             // If there is a problem discovered in any function, an
76:             // exn is thrown and printed here.
77:             complain() << error.what() << endl;
78:          }
79:       }
80:    } catch (ysh_exit&) {
81:       // This catch intentionally left blank.
82:    }
83:
84:    return exit_status_message();
85: }
86:
```

```
 1: # $Id: Makefile,v 1.34 2019-10-27 20:59:20-07 - - $
 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: GPPWARN     = -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
 9: GPPOPTS     = ${GPPWARN} -fdiagnostics-color=never
10: COMPILECPP  = g++ -std=gnu++17 -g -O0 ${GPPOPTS}
11: MAKEDEPCPP  = g++ -std=gnu++17 -MM ${GPPOPTS}
12: UTILBIN     = /afs/cats.ucsc.edu/courses/cse111-wm/bin
13:
14: MODULES     = commands debug file_sys util
15: CPPHEADER   = ${MODULES:=.h}
16: CPPSOURCE   = ${MODULES:=.cpp} main.cpp
17: EXECBIN     = yshell
18: OBJECTS     = ${CPPSOURCE:.cpp=.o}
19: MODULESRC   = ${foreach MOD, ${MODULES}, ${MOD}.h ${MOD}.cpp}
20: OTHERSRC    = ${filter-out ${MODULESRC}, ${CPPHEADER} ${CPPSOURCE}}
21: ALLSOURCES  = ${MODULESRC} ${OTHERSRC} ${MKFILE}
22: LISTING     = Listing.ps
23:
24: all : ${EXECBIN}
25:
26: ${EXECBIN} : ${OBJECTS}
27:         ${COMPILECPP} -o $@ ${OBJECTS}
28:
29: %.o : %.cpp
30:         - ${UTILBIN}/cpplint.py.perl $<
31:         - ${UTILBIN}/checksource $<
32:         ${COMPILECPP} -c $<
33:
34: ci : ${ALLSOURCES}
35:         - ${UTILBIN}/checksource ${ALLSOURCES}
36:         ${UTILBIN}/cid + ${ALLSOURCES}
37:
38: lis : ${ALLSOURCES}
39:         ${UTILBIN}/mkpspdf ${LISTING} ${ALLSOURCES} ${DEPFILE}
40:
41: clean :
42:         - rm ${OBJECTS} ${DEPFILE} core ${EXECBIN}.errs
43:
44: spotless : clean
45:         - rm ${EXECBIN} ${LISTING} ${LISTING:.ps=.pdf}
46:
```

```
47:
48: dep : ${CPPSOURCE} ${CPPHEADER}
49:         @ echo "# ${DEPFILE} created `LC_TIME=C date`" >${DEPFILE}
50:         ${MAKEDEPCPP} ${CPPSOURCE} >>${DEPFILE}
51:
52: ${DEPFILE} : ${MKFILE}
53:         @ touch ${DEPFILE}
54:         ${GMAKE} dep
55:
56: again :
57:         ${GMAKE} spotless dep ci all lis
58:
59: ifeq (${NEEDINCL}, )
60: include ${DEPFILE}
61: endif
62:
```

```
1: # Makefile.dep created Sun Oct 27 21:05:44 PDT 2019
2: commands.o: commands.cpp commands.h file_sys.h util.h debug.h
3: debug.o: debug.cpp debug.h util.h
4: file_sys.o: file_sys.cpp debug.h file_sys.h util.h
5: util.o: util.cpp util.h debug.h
6: main.o: main.cpp commands.h file_sys.h util.h debug.h
```