379 Assignment 2 Writeup

Kristofer Mitchell - 1213352

Notes and Assumptions

- This is a pdf file, not a .txt (due to the requirement that we have 2 *page* report).
- my program DOES explore subdirectories (though would be a 2 line change to stop this, uncomment lines in store_list in dirdiff.c to change)
- I don't call accept() on connections in the server after MAX_CLIENTS has been reached

I spent a lot of time on this lab assignment. I feel that directory monitoring is not a good assignment to make use of concepts covered in class. A lot of time was spent on edge cases; the directory differencing was fidgety, and the assignment never *requires* that we use any form of concurrency. No semaphores, no monitors, no locks or atomic variables. No inter-process communication is required. It is an exercise on select() and C socket programming, and it seems out of place in an operating systems course. I appreciate the C programming exercise, but I would have preferred something more relevant.

1. Program Architecture

There were 3 main components to this assignment, and I will discuss how I designed each of them:

- The directory monitor
- The client/server protocol
- The client interface

Directory Monitor

The actual directory comparison is done in dirdiff.c . I have several data structures that I use to carry out directory comparisons, defined in dirapp.h. The parent structure is a "dirdiff", which contains buffer lists for "dirstat"s (struct that encapsulates a filename and stat). Directory differencing is done by the function "diffDirectory", which performs a file tree walk and records all dirstats, then compares them to the "old" list using a relatively inefficient (n^2) "pairing" algorithm. Updates are written out to a socket in batches of 254 with the function writeDiffEntries. Since I use a separate process to monitor the directory (to preserve time accuracy, which is not necessarily the case with select() solutions), writeDiffEntries is called with a pipe to the server process. The server process monitors the pipe with select() and writes updates to the client as it receives them.

Client/Server Protocol

I implemented the client and server in separate files, client.c and server.c . The various protocol states are kept stored in the host struct (explained more in Other). A “status” value along with a protocol “step” and an expected number of characters from the server define the protocol state. In the case of the client, I use if’s to select the current status of the protocol, and a switch statement to select the appropriate protocol step. This seemed more elegant than simply having one integer value to represent the protocol state, but it does take up more space. On the server, the protocol is mostly handled in different functions: clientConnect handles connection, writeUpdates() handles the update loop, clientDisconnect handles disconnection and errors. I used a select() loop in both files to poll the active sockets and file descriptors.

Client Interface

The client interface is contained in client.c. I used the readline library to get input from the terminal. Characters are then parsed by execCommand, and the appropriate command is executed. I use functions to modularize each command (addHost, removeHost, list, quit), and strncmp and sscanf to determine whether or not the string given is a valid command.
My client interface turned out to be more complicated than I expected. Initially, I used select() and sscanf/fgets to read information from stdin, but since I wanted updates to be displayed dynamically, I had to use readline. The macro for xprintf (used throughout the code) ensures that when new content (eg. updates from the server) is printed to stdout, the user’s current input is saved and redisplayed after the new content.

Other

I modularized most of my “host” struct into the files host.h and host.c. A host struct is an entity with a hostname, port number, status, protocol specific information (like the step and the expected number of chars), update line buffer, update period, and a socket file descriptor. Not all of these fields are used on both the client and the server; the server doesn’t set a hostname, for example. host.c contains different utility functions used to operate on hosts. clearHosts ensures that a list of hosts starts off zeroed and disconnected. selectHostFD returns the index of the host in a list of hosts with socket file descriptor matching the one provided, or -1 if none is found. selectDiscHost is similar, but returns the index of the first disconnected host in the list. activeHosts returns the number of connected or initializing hosts in a list. terminateClientConnection is used by the server to send the terminating protocol bytes to the client. initHost populates all the necessary fields of a host from provided values. writeToClients is a utility function used by the server to send a message to all clients.

common.c contains simple utility functions. timeF returns a formatted time string, used mostly by dirdiff.c. permOfFile returns directory permissions in a formatted string. die is used to exit with errors. slog is commented out, but normally used to write to a log file. checkPort is used to determine if a port number is valid.

dirapp.c and test.c contain separate main functions for either launching the program, or running the tests.

2. Client UI

My client UI is fairly similar to the TA's sample program, with several improvements:

- Updates will be printed when they are received, and user input is preserved
- Prompting the user for input doesn't lag between commands
- you can use the first letter of a command rather than typing the whole command (eg. a localhost 9000 is equivalent to add localhost 9000)
- command history is enabled; use up or down arrow keys to navigate through history
- updates are displayed with the server name in front of them, and indented
- updates have additional information with them

3. Untested Requirements

- owner change
- owner group change

4. Tested Requirements

Requirements tested manually:

- directory name must match a directory on the server
- server becomes a daemon correctly
- command parsing

    - add
    - remove
    - list
    - quit
- client may only be connected to MAX_SERVERS servers
- server may only have MAX_CLIENTS clients
- add command works
- list command works
- remove command works
- quit command works
- connection protocol
- change in connection protocol causes connection to abort
- disconnection protocol
- change in disconnection protocol causes connection to abort
- error protocol
- directory removal causes server to exit/print error
- SIGHUP received by client
- SIGHUP received by server
- output from multiple servers doesn’t interleave messily

Requirements tested semi-automatically (with the aid of scripts):

- update loop
- more than MAX_SEND (254) entries updated
- symbolic links
- very long update strings

Requirements tested automatically (by scripts entirely):

- port number must be valid to start server/connect as client
- directory differencing with:
    - creation
    - remove
    - size change
    - access mode change
    - multiple changes concatenated with a comma
- host struct functionality

Other programs I tested with:

- TA’s solution