CMPS 111
Assignment 1 Design Document - Minix Shell

Melanie Dickinson - mldickin@ucsc.edu 858-248-3720
Neil Killeen - nkilleen@ucsc.edu 925-719-3846
Dave Lazell - dlazell@ucsc.edu 803-464-4428
Desmond Vehar - dvehar@ucsc.edu 619-861-9209

Behavior:

This program implements a simple shell, supporting background execution and file redirection.

User input is defined as follows:

- The maximum number of tokens entered at once has been limited to 19. Any tokens past the 19th are not processed. The 20th position is reserved for the sentinel.
- Each word is contains letters, numbers, period, minus, and slash, or a single special character from “<>&”. Any of the words in the user input that don’t conform to this definition will be ignored.
- Commands should take one of the following forms (blue text is description). If they do not then the shell will print an error message and wait for the next command.
  - Nothing
  - exit
    Just “exit” and nothing else
  - cmd_name [argument…]
    A command name that is optionally followed by arguments
  - cmd_name [argument...] [> outfile] [< infile] [&]
    A command name that is optionally followed by arguments which are optionally followed by redirects (with a single mandatory redirect location specified) which are optionally followed by an ampersand.
  - cmd_name [argument...] [< infile] [> outfile] [&]
    A command name that is optionally followed by arguments which are optionally followed by redirects (with a single mandatory redirect location specified) which are optionally followed by an ampersand.

All system calls, if returning with an error, will cause an error message to be displayed.

On program exit, all commands running in the background will continue to run until their normal exit. They will not be shut down when the main shell exits.

Implementation:

Basically there is a main loop in which we loop forever, doing the following:

1) Print the prompt

2) Get an array of tokens from the command line. Tokens include “<”, “>”, and “&”, and other alphanumeric strings separated by whitepace.

3) exit the program if we have an “exit” token

4) determine if we are executing in the background. This is a separate function that checks to see if the last token is the “&” token. It also removes the “&” from the token array.

5) determine if we are redirecting, and to what file. Also remove the “<” or “>” token and the filename from the token array, and returns 1 if we are redirecting (direction specified as an argument), or 0 if we aren’t.

6) fork. If we are the child process, we will set up redirection based on the results of #4 above (using freopen). We will then execute the command using execvp, then return the redirection to normal with fclose.

If instead we are the parent, we will wait on the child process from the fork() call with waitpid(), and only if we are not executing the command in the background as in #3 above.

In addition to the main loop above, main() sets up a signal handler before entering the main loop. This handler catches the SIGCHLD signal in order to reap the child processes.

Tricky issues encountered:

1) In order to avoid zombie children, a separate function was created to setup a callback signal-handler function, via sigaction. This signal-handler function just calls wait() on the exited child in order to reap it.

2) The function to get the input and tokenize it had to be modified to use feof() in order to determine if the result of the getc() call was truly an eof, or errored out due to being interrupted by a signal.

Testing:

Testing the program will be to run it and test each individual case as well as combing test cases (example: “func -lol red > out1 < in1 &”). Various functions were tested by putting printf statements into the code at strategic points, and then removed for the final code.