Instructions

Programming Project #1: Shell

The main goals for this project are to familiarize you with the MINIX3 operating system—how it works, how to use it and to compile code for it. We also want to give you an opportunity to learn how to use system calls. To do this, you're going to implement a Unix shell program. A shell is simply a program that conveniently allows you to run other programs; your shell will resemble the shell that you're familiar with from logging into unix.ic or other Unix computers.

Basics

You should read over the general project information page *before* you start this project. In it, you'll find valuable information about the MINIX 3 operating system and the virtual machines available to run it as well as general guidelines and hints for projects in this class. Before going on to the rest of the assignment, get MINIX running in your choice of VM.

Details

Parsing an input line

You're going to need to parse an input line by breaking it up into *tokens*. There are two types of tokens: words and meta-characters. Meta-characters are the characters >, <, and |. Words are composed of all other characters*except* meta-characters and white space. In addition, the combination of \ (backslash) followed by a meta-character or whitespace is considered a word character.

A word is a sequence of one or more word characters, and is terminated by either whitespace or a meta-character (again, recall that spaces and meta-characters preceded by a backslash are word characters).

Your job in parsing the input line is to go through the line and record all of the words and metacharacters in separate entries in an array. We recommend simply having a function that looks like this:

```
parseline (char *line, char *buffer, char **tokens)
```

where `buffer` is an (empty) array about twice as large as `line` and `tokens` is an array of pointers to characters that gets filled in with a pointer to each token. Then, all that `parseline` has to do is run through the `line` array, copying characters to `buffer`. If a backslash is encountered, copy the next character to the buffer verbatim. As soon as a token (word or meta-character) ends, drop a NULL (\0) into the buffer, and add an entry to the tokens array. Note that it might be easier to add a pointer to the *following* entry, since you know where that is (immediately following the NULL); if you do this, make sure you add a pointer to the start of the buffer into the first entry in the token array. For example, the string

```
echo hello_there another\ word |wc
```

would generate the tokens

```
echo
hello_there
another word
|
wc
```

Each token would be NULL-terminated.

parseline isn't that hard to write, as long as you go through the pseudocode *first* and make sure it seems to work. If you try to write it without designing it first, it may take longer—do the design *first*, then code.

Shell commands

Your shell must support the following:

1. The internal shell command exit which terminates the shell.
   Concepts: shell commands, exiting the shell
   System calls: exit()
2. A command with no arguments.
   Example: ls
   Details: Your shell must block until the command completes and, if the return code is abnormal, print out a message to that effect. This holds for *all* command strings in this assignment.
   Concepts: Forking a child process, waiting for it to complete, synchronous execution.
   System calls: fork(), execvp(), exit(), wait()
3. A command with arguments.
   Example: ls -l
   Details: Argument zero is the name of the command other arguments follow in sequence.
   Concepts: Command-line parameters.
4. A command, with or without arguments, whose output is redirected to a file.
   Example: ls -l > file
   Details: This takes the output of the command and puts it in the named file.
   Concepts: File operations, output redirection.
   System calls: close(), dup()
5. A command, with or without arguments, whose input is redirected from a file.
   Example: sort < scores
   Details: This uses the named file as input to the command.
   Concepts: Input redirection, file operations.
   System calls: close(), dup()
6. A command, with or without arguments, whose output is piped to the input of another command.
   Example: ls -l | more
   Details: This takes the output of the first command and makes it the input to the second command.
   Concepts: Pipes, synchronous operation
   System calls: pipe(), close(), dup()

Your shell *must* check and correctly handle *all* return values. This means that you need to read the manual pagesfor each function and system call to figure out what the possible return values are, what errors they indicate, and what you must do when you get that error.

Your shell should support any combination of these characters on a single line, as long as it makes sense. For example,
ls -l | sort > result.txt
should run the output of the first command into the input of the second and redirect the output of the second command into result.txt. Remember that input redirection only applies to the *first* command on the line, and output redirection only applies to the *last* command; pipe symbols separate two commands and send output from the preceding command to the following command.

Your shell should handle at least 20 commands on a single line, with each command having up to 50 arguments. The total size of the command line will not exceed 1024 characters. We strongly suggest using ensuring that you don't have a buffer overflow beyond 1024 characters by using fgets() or taking other similar precautions. It's OK to chop off lines after 1024 characters, but your shell *must not crash* if the user tries to do something they shouldn't.

Deliverables

You must hand in a compressed tar file of your project directory, including your design document. You must do a "make clean" before creating the tar file. In addition, you should include a README file to explain anything unusual to the teaching assistant. Your code and other associated files must be in a single directory; the TA will copy them to his MINIX installation and compile and run them there.

Do not submit object files, assembler files, or executables. Every file in the tar file that could be generated automatically by the compiler or assembler will result in a 5 point deduction from your programming assignment grade. (Note that, while there *are* programs that can generate a Makefile for you, we *do* expect you to turn one in.)

Your design document should be called design.txt (if plain ASCII text, with a maximum line length of 75 characters) or design.pdf (if in PDF), and should reside in the project directory with the rest of your code. Formats other than plain text or PDF are not acceptable; please convert other formats (MS Word, LaTeX, HTML, etc.) toPDF. Your design document should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs. A sample design document is available.

Hints

- ***START EARLY!*** You should start with your design, and check it over with the course staff.
- Build your program a piece at a time. Get one type of command working before tackling another.
- Experiment! You're running in an virtual machine—you *can't* crash the whole computer (and if you manage to do it, let us know...).
- You'll probably want to edit your code outside of MINIX (using your favorite text editor) and copy it into MINIX to compile and run it. You can use rsync or other methods to keep your files synchronized between your host machine and MINIX. This has several advantages:
    - Crashes in MINIX don't harm your source code (by not writing changes to disk, perhaps).
    - Most OSes have better editors than what's available in MINIX.
- ***START EARLY!***
- Test your shell. You might want to write up a set of test lines that you can cut and paste (or at least type) into your shell to see if it works. This approach has two advantages: it saves you time (no need to make up new commands) and it gives you a set of tests you can use every time you add features. Your tests might include:
    - Different sample commands with the features listed above
    - Commands with errors: command not found, non-existent input file, etc.
    - Malformed command lines (e.g., ls -l >| foo)
- Use a version control system such as svn, git, or cvs to keep multiple revisions of your files. Version-control systems are very space-efficient, and allow you to keep multiple coherent versions of your source code and other files (such as Makefiles and design documents). You can also check in your files before trying something that might not work, and revert back to a previous version if you want. No more commenting things out!
- Did we mention that you should **START EARLY!**

We assume that you are already familiar with Makefiles and debugging techniques from earlier classes such as CMPS 101 or from the sections held the first week of class. If not, this will be a considerably more difficult project because you will have to learn to use these tools as well.

This project doesn't require a lot of coding (typically fewer than 200 lines of code), but does require that you understand how to use MINIX and how to use basic system calls. You're encouraged to go to the class discussion section or talk with the course staff during office hours to get help if you need it.

You should do your design ***first***, before writing your code. Start with the simple code shell we went over the first week of class, and figure out how you might need to add to it to get all the functionality added. Don't worry if you can't figure out (at first) how to get a particular piece of functionality working. You may also want to experiment with small pieces of code (such as the code you'll need to split a line into tokens) as you write the design document. It's OK to (for example) say that you need a function to split a line into tokens and then figure out how to do that separately. While it may be more fun to just start coding without a design, it'll also result in spending more time than you need to on the project.

IMPORTANT: As with all of the projects this quarter, the key to success is starting early. You can always take a break if you finish early, but it's impossible to complete a 20 hour project in the remaining 12 hours before it's due....

Extra credit

Extra credit will involve supporting background processes. Details will be available shortly.

Extra credit will *only* be applied to projects with a raw score of 85 or higher. In other words, don't even think of trying the extra credit until you get the basic functionality working (and that includes a strong design document).

Project groups

The first project must be done individually; however, later projects (numbers 2–4) will be done in randomly-assigned groups of three. It's vital that every student in the class get familiar with how to use the MINIX system; the best way to do that is to do the first project yourself. For the second, third, and fourth projects, you'll have partners assigned well in advance of the project assignment date.