Programming Project #1
CMPS 111, Winter 2014

Purpose

The main goals for this project are to familiarize you with the MINIX3 operating system—how it works, how to use it, and how to compile code for it and 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 the projects overview material (available on the Resources page) **before** you start this project. In it, you'll find valuable information about the MINIX 3 operating system and the x86 emulators 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 hypervisor (VirtualBox, VMWare, QEMU, KVM, etc).

You are provided with two files, `shell.c` and `makeargv.c`. The latter contains a tokenizer function that takes null-terminated C strings of input and delimiters (i.e. all possible characters that separate two tokens) and a pointer to an array of C-strings. The tokenizer returns the number of tokens, and points the pointer to the location in memory holding the array of C-strings it has allocated for you. The former is a simple loop that takes input from `stdin`, passes it to the tokenizer and the prints out the tokens.

You need to compile the code to object files and and then link the object files into one executable. You are also provided with a `Makefile` that accomplishes this basic task, though you are advised to tailor it to your liking as you work on the project.

Details

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()`, `waitpid()`

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 list of commands separated by the UNIX && command.
Example: `ls -l mydir && more mydir/foofile`
Details: The && operator excutes the command after the && symbol if and only if the command before the operator exits successfully.
Concepts: Appropriate error handling.
System calls: `fork()`, `execvp()`, `waitpid()`
7. A command, with or without arguments, run in the background
Example: `ls -r / &`
Details: The & operator executes the command in the background; the user continues to interact with the shell while the command is run. As soon as the command is executed, the shell should print out the process id (PID) on a new line. When the command terminates the system will issue a SIGCHLD to the parent (your shell). You should set up a signal handler to catch this signal. The signal handler should print out the word “Done” followed by a space and the PID of the child.
Concepts: Job control, basic signal handling
System calls: `fork()`,`execvp()`,`sigaction()`

Your shell *must* check and correctly handle *all* return values. This means that you need to read the manual pages for 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.

Your shell should handle at least 10 commands on a single line, with each command having up to 20 arguments. The total size of the command line will not exceed 4096 characters. We strongly suggest using ensuring that you don't have a buffer overflow beyond 4096 characters by using `fgets()` or taking other similar precautions.

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 `README` should also contain running instructions to the TA. Without running instructions, the TA is most likely to return your project to you. Your code and other associated files must be in a single directory; the TA will copy them to her 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. If you have files that you created for testing, it is acceptable (encouraged even) to submit those with your project. Just be sure to make a list of source and test files in your `README` file.

Your design document should be called `DESIGN`, and should reside in the project directory with the rest of your code. Formats other than plain text are not acceptable. 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 on the course Resources page.

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 emulated system—you *can't* crash the whole computer (and if you can, let us know...).
- You may want to edit your code outside of MINIX (using your favorite text editor) and copy it into MINIX to compile and run it. 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 (CVS, SVN, git) to keep multiple revisions of your files.
- Did we mention that you should ***START EARLY!***

We assume that you are already familiar with `Makefile`s 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. To do this, experiment with the existing shell template (if you like), inserting debugging print statements if it'll help. It may be more "fun" to just start coding without a design, but 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....

Project groups

The first project must be done individually; however, later projects (numbers 2–4) may be done in pairs. 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 may pick a partner and work together on the project.