# Assignment 1

# Goals

The primary goal of this assignment is to further warm up your C programming skills, and to become familiar with the system call interface and processes. A secondary goal is to use some of the programming tools provided in FreeBSD. In this assignment you are to implement a simple FreeBSD shell program. A shell is simply a program that conveniently allows you to run other programs.

# Basics

**BEFORE YOU DO ANYTHING ELSE**, create and switch to the branch for Assignment 1 (name the branch asgn1) in your local repository. In order to check out a different branch, all of your changes in the current branch must be committed. You may switch between branches using git checkout at any time; changes in each branch are preserved.

You are provided with the files **shell.l** and **argshell.c** (find them under canvas files) that contain some code that calls get_args(), a function provided by shell.l that reads and parses a line of input. You can use get_args() in your program rather than parsing text input yourself. The argshell.c file contains an example of a shell that receives an input string and then parses it. (you can base your shell on this example and extend it to support the functionality of an actual shell that we present later.)

The get_args() function returns an array of pointers to character strings. Each string is either a word containing letters, numbers, period (.), dash (-), underscore (_) and forward slash (/), or a character string containing only the special characters: ( ) < > | & ;, which have special meaning to the shell. Note that some of the sequences below use *two or more* special characters in a row.

To compile shell.l, you have to use the flex command. This produces a file called (by default) lex.yy.c. You must compile and link lex.yy.c and the .c file(s) that contain your shell code in order to get a running program. This includes a compile step first using the cc command to compile each c file. Then, you perform the link step to link the object files (.o files) that resulted from the compile step. In the link step, you also have to use -lfl to get everything to work properly. Your link step should look like this:
cc -o argshell argshell.o lex.yy.o -lfl
Note that -lfl has to come *last*, after your .o files. Also, all of your code building instructions should be in a makefile, arranged so that make with no arguments builds your shell. You should have gotten some practice writing makefiles in the first assignment, but there are plenty of tutorials online. FreeBSD doesn't use GNU make, so makefile might be a bit different.
After compiling the provided files you should have a an argshell executable that performs the following:

```
 $ ./argshell
Command ('exit' to quit): ls -l > outfile
Argument 0: ls
Argument 1: -l
Argument 2: >
Argument 3: outfile
Command ('exit' to quit): exit
$
```

# Details

## Supported features

Your shell must support the following:

1. The internal shell command exit
2. **Details:** The exit command terminates the shell
3. **Concepts:** shell commands, exiting the shell
4. **Library calls:** exit()

5. A command with no arguments.
6. *Example:* ls
7. **Details:** This runs the command, which can be *any* executable in your default search path (the one that execvp() searches). Your shell must block until the command completes and, if the return code is abnormal (non-zero), print out a message to that effect, perhaps using perror(). This holds for all command strings in this assignment.
8. **Concepts:** Forking a child process, waiting for it to complete, synchronous execution.
9. **System calls:** fork(), execvp(), wait()
10. A command with one or more arguments.
11. *Example*: ls -l foo
12. **Details:** Argument 0 is the name of the command; other arguments follow in sequence. A special character ends the command, which means that I/O redirection (see below) has to come *after* all the arguments to a command.
13. **Concepts:** Command-line parameters
14. **System calls:** fork(), execvp(), wait()
15. A command, with or without arguments whose input is redirected *from* a file.
16. *Example:* sort -nr < scores
17. **Details:** This takes the named file as input to the command.
18. **Concepts:** Input redirection, file operations.
19. **System calls:** open(), close(), dup()
20. A command, with or without arguments, whose output is redirected *to* a file.
21. *Example:* ls -l > file
22. *Example:* ls -l >> file
23. **Details:** This takes the output of the command and appends it to the named file. If the > form is used, the command output overwrites what is in the output file. If the >> form is used, the command output is *appended* to the end of the

output file, creating it if necessary. If the output file doesn't already exist, > and >> do the same thing.

24. **Concepts:** File operations, output redirection.
25. **System calls:** open(), close(), dup()
26. A command, with or without arguments, whose output is piped to the input of another command.
27. *Example:* ls -l | less
28. **Details:** This takes the output of the first command and makes it the input to the second command.
29. **Concepts:** Pipes, synchronous operation
30. **System calls:** pipe(), close(), dup()
31. Redirection of standard error.
32. *Example:* ls -l >& foo
33. *Example:* ls -l >>& foo
34. *Example:* ls -l |& wc
35. **Details:** If any output redirection operator includes an & character at the end, standard error (file descriptor 2) is redirected in addition to standard output, and both go to the same file / pipe.
36. **Concepts:** File operations, pipes, output redirection
37. **System calls:** pipe(), open(), close(), dup()
38. Two or more command sequences, possibly involving I/O redirection, separated by a semicolon.
39. *Example:* cd /home/elm/cmps111 ; sort -nr < grades ; ls -l > foo
40. **Details:** The shell runs the first command, and waits for it to finish. When it finishes, the next command is run, regardless of any error encountered by the previous sequence. I/O redirections apply only to their specific command, not to *all* previous commands (or future commands). In the example above, the output of sort would *not* be redirected, but the output of ls would be.
41. **Concepts:** Multiple commands on a single line.
42. **System calls:** wait()

43. The internal shell command cd
44. *Example:* cd /usr
45. *Example:* cd
46. **Details:** This shell command sets the working directory to the directory provided in the command. If no directory is provided, the working directory is set to whatever the working directory was when the shell was started (so you might want to get this directory when your program starts and save it...).
47. **Concepts:** Working directory
48. **System / library calls:** chdir(), getcwd()

# Error handling

You 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.

# Deliverables

As with Assignment 0, you'll be writing your code in a subdirectory of the source tree you checked out. You can use the same repository as you did for Assignment 0, but you need to put your files into the asgn1 subdirectory under the repository root. As before, you'll be using git to commit changes as you write your code, and will need to push your commits to your repository for grading.

At the time of submission, your asgn1 subdirectory must include your source code files, your makefile, and a design document. In addition, include a README file to explain anything unusual to the TA. Your design document should be called design.txt (if in plain text), or design.pdf (if in Adobe 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

(Word, LaTeX, HTML, ...) to PDF. **Don't submit a Word file—it's not a text file.** 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 canvas.

# Committing and submitting

You should do regular commits of your code to git; you will lose points if you don't do regular git commits. You need not push your commits to the server every time you commit, but you should push on a regular basis.

Do not commit object files, assembler files, or executables to your git repository. Each file in the submit directory that could be generated automatically by the compiler
or flex (including lex.yy.c!) will result in a deduction from your programming assignment grade.

You'll submit your program the same way you submitted assignment 0. Please read the instruction if you need to refresh your memory.

# Hints

- The first target in a makefile is the one that's built if no arguments are provided. So, the first target in
  your makefile should be myshell, or at least build it.
- Write your design document *first*. Go through it step by step to ensure that your design is sound. Yes, it means you'll spend more time before writing code, but it means it'll take you a lot less time to write it and (especially) debug it. We're going to ask for your design document if you want help debugging your code.

- Use a debugger. FreeBSD includes the gdb debugger, and you can ask the compiler to create debugging symbols with the -g option.

# Extra credit

**10 points: Chaining multiple commands via pipes.**

The default assignment only requires you to be able to chain two commands together. For this extra credit, you need to be able to chain as many commands together, via pipes, as the user wants to use. For example, consider this classic pipe sequence:

cat mypaper.tr | pic | tbl | eqn | groff -mm -Tps | ps2pdf - > mypaper.pdf

This sequence requires that each command pipe its output as input to the next one. Note that commands in the pipe may themselves have arguments.