ASH Project

## **Unix Shell**

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project. There are three specific objectives to this assignment:

• To further familiarize yourself with the Unix/Linux programming environment.

• To learn how processes are created, destroyed, and managed.

• To gain exposure to the necessary functionality in shells.

## **Overview**

In this assignment, you will implement a command line interpreter (CLI) or, as it is more commonly known, a shell. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished. The shells you implement will be similar to, but simpler than, the one you run every day in your Operating System. **One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.**

## **Program Specifications**

Basic Shell: ash Your basic shell, called ash (short for **A**ggie **SH**ell, naturally), is simply an interactive loop: it repeatedly prints a prompt dash> (note the space after the greater-than sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types exit. The name of your final executable should be ash. The shell can be invoked with either no arguments or a single argument; anything else is an error. Here is the no-argument way: At this point, dash is running, and ready to accept commands. Type away! The mode above is called interactive mode, and allows the user to type commands directly. The shell also supports a batch mode, which instead reads input from a batch file and executes commands from therein. Here is how you run the shell with a batch file named batch.txt: prompt> ./ash batch.txt One difference between batch and interactive modes: in interactive mode, a prompt is printed (ash>). In batch mode, no prompt should be printed. You should structure your shell such that it creates a process for each new command (the exception are built-in commands, discussed below). Your basic shell should be able to parse a command and run the program corresponding to the command. For example, if the user types ls -la /tmp, your shell should run the program /bin/ls with the given arguments -la and /tmp (how does the shell know to run /bin/ls? It's something called the shell path; more on this below).

## **Structure**

Basic Shell The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command exit, at which point it exits. That's it! For reading lines of input, you should use getline(). This allows you to obtain arbitrarily long input lines with ease. Generally, the shell will be run in interactive mode, where the user types a command (one at a time) and the shell acts on it.

Your shell will also support batch mode, in which the shell is given an input file of commands; in this case, the shell should not read user input (from stdin) but rather from this file to get the commands to execute. In either mode, if you hit the end-of-file marker (EOF), you should call exit(0) and exit gracefully. To parse the input line into constituent pieces, you might want to use strtok() (or, if doing nested tokenization, use strtok\_r()). Read the man page (carefully) for more details. To execute commands, look into fork(), exec(), and wait()/waitpid(). See the man pages for these functions, and also read the relevant book chapter for a brief overview. You will note that there are a variety of commands in the exec family; for this project, you must use execv. **You should not use the system()** library function call to run a command. Remember that if execv() is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified.

## **Paths**

In our example above, the user typed ls but the shell knew to execute the program /bin/ls. How does your shell know this? It turns out that the user must specify a path variable to describe the set of directories to search for executables; the set of directories that comprise the path are sometimes called the search path of the shell. The path variable contains the list of all directories to search, in order, when the user types a command. Important: Note that the shell itself does not implement ls or other commands (except built-ins). All it does is find those executables in one of the directories specified by path and create a new process to run them. To check if a particular file exists in a directory and is executable, consider the access() system call. For example, when the user types ls, and path is set to include both /bin and /usr/bin, try access("/bin/ls", X\_OK). If that fails, try "/usr/bin/ls". If that fails too, it is an error. Your initial shell path should contain one directory: `/bin' Note: Most shells allow you to specify a binary specifically without using a search path, using either absolute paths or relative paths. For example, a user could type the absolute path /bin/ls and execute the ls binary without a search path being needed. A user could also specify a relative path which starts with the current working directory and specifies the executable directly, e.g., ./main. In this project, you do not have to worry about these features.

## **Built-in Commands**

Whenever your shell accepts a command, it should check whether the command is a built-in command or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For example, to implement the exit built-in command, you simply call exit(0); in your dash source code, which then will exit the shell. In this project, you should implement exit, cd, and path as built-in commands.

• exit: When the user types exit, your shell should simply call the exit system call with 0 as a parameter. It is an error to pass any arguments to exit. Upon exit you list your name, along with your project name (Aggie Shell).

• cd: cd always take one argument (0 or >1 args should be signaled as an error). To change directories, use the chdir()system call with the argument supplied by the user; if chdir fails, that is also an error.

• path: The path command takes 0 or more arguments, with each argument separated by whitespace from the others. A typical usage would be like this: dash> path /bin /usr/bin, which would add /bin and /usr/bin to the search path of the shell. If the user sets path to be empty, then the shell should not be able to run any programs (except built-in commands). The path command always overwrites the old path with the newly specified path.

## **Redirection**

Many times, a shell user prefers to send the output of a program to a file rather than to the screen. Usually, a shell provides this nice feature with the > character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should also include this feature, but with a slight twist (explained below). For example, if a user types ls -la /tmp > output, nothing should be printed on the screen. Instead, the standard output of the ls program should be rerouted to the file output. In addition, the standard error output of the file should be rerouted to the file output (the twist is that this is a little different than standard redirection). If the output file exists before you run your program, you should simple overwrite it (after truncating it). The exact format of redirection is a command (and possibly some arguments) followed by the redirection symbol followed by a filename. Multiple redirection operators or multiple files to the right of the redirection sign are errors. Note: don’t worry about redirection for built-in commands (e.g., we will not test what happens when you type path /bin > file).

## **Parallel Commands**

There will be no Concurrent commands. The shell will operate interactively on one command at a time.

## **Program Errors**

char error\_message[30] = "An error has occurred\n"; write(STDERR\_FILENO, error\_message, strlen(error\_message));

The one and only error message. You should print this one and only error message whenever you encounter an error of any type: The error message should be printed to stderr (standard error). After most errors, your shell should simply continue processing after printing the one and only error message. However, if the shell is invoked with more than one file, or if the shell is passed a bad batch file, it should exit by calling exit(1). There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there are any program-related errors (e.g., invalid arguments to ls when you run it, for example), the shell does not have to worry about that (rather, the program will print its own error messages and exit).

## **Miscellaneous Hints**

Remember to get the basic functionality of your shell working before worrying about all of the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as ls). Next, add built-in commands. Then, try working on redirection. Each of these requires a little more effort on parsing, but they should not be too hard to implement.

At some point, you should make sure your code is robust to white space of various kinds, including spaces () and tabs (\t). In general, the user should be able to put variable amounts of white space before and after commands, arguments, and various operators; however, the operators (redirection and parallel commands) do not require whitespace. Check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. It's also just good programming sense. Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of different inputs at it and make sure the shell behaves well. Good code comes through testing; you must run many different tests to make sure things work as desired. Don't be gentle -- other users certainly won't be. Finally, keep versions of your code. More advanced programmers will use a source control system such as git. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

## **Testing**

There should not be any error messages and warning during compilation. Create your own test cases and test thoroughly.

# Check Point 1

1. Prompt “ash > “is displayed
2. Built in “exit”

**Due date CP1: September 24th by 11:59 PM**

# Check Point 2

1. All Check Point 1
2. All built in should be done
3. Cursor should reflect the current path “ash /usr/bin >” if you are in the usr/bin directory. “ash >” if you are in your home directory.

**Due date CP2: October 15th, by 11:59 PM**

# Check Point 3

1. All Check Point 2
2. Interactively execute single commands with or with out arguments
3. Redirection

**Due date CP3: November 5th, by 11:59 PM**

# Final

1. All Check Point 3
2. Batch processing
3. All project requirements are due.

**Due date CP4: November 26th, by 11:59 PM**

**Grading**  
The following tasks are how the groups will be graded each by each Checkpoint due date. You will need to create a directory (folder) in your workspace, called “ash”. The project will be saved in a file called “ash.cpp”

1. In order to have the Check Point graded ash.cpp must compile.
2. 70% of the grade comes from if the code accomplishes the Check Point tasks and the code being error/warning free.

* Are Check Point Tasks completed
* Compilation warnings
* Unexpected termination
* Not handling a carriage return or a null carriage return.

1. 30% of the grade comes from using good coding style

* white space usage
* comments/documentation
* indention…etc.