Shell Part 1

CS453: Operating systems

Overview

In this project, we will be extending the functionality of the simple shell example that was discussed in class. We will call our shell **mydash** (for My Dead Again SHell). We will add the capability to accept arguments to commands and add some basic built-in commands like **exit** and **cd**.

All the programming assignments for Linux will be graded on the onyx cluster (onyx.boisestate.edu). Please test your programs on onyx as differences between different Linux distros (CentOS, Fedora, Ubuntu) can cause compilation issues when using simple Makefiles. To get around this problem you would normally use a build system such as Autotools or Cmake which detects what system you are building on, however for this class using a meta-build system is overly complex and unnecessary when we have a shared cluster. Finally we will be using git (backpack) for version control and submission of projects.

Starter Code

In your git repository you will have a folder named **shell** with all the relevant build and header files. Additionally, during development you should **not commit any other generated files** such as **.o**, **.d** into source control.

Project Management

As this is a relatively complex project specification, we will be using the web-based Trello.com tool for project management. Please create a free account for yourself on Trello and create a board for this project. As you read through this specification, create cards for the tasks you need to finish. For each card, you can also create a checklist to further decompose the project into manageable chunks. You will be submitting a screenshot of the Trello board for the project.

Specification

Filename completion and command history

The GNU readline library allow a program control over the input line. The **readline()** function allows the user to edit the input line, use TAB key for filename completion, the use of up arrow, down arrow, left arrow, right arrow keys to access the history of commands typed in by the user. The following code sample shows the usage of the readline library.

```
/*
 *  Filename: test-readline.c
 *  Compile:  gcc -Wall test-readline.c -lreadline -lncurses
 */
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>

int main() {

    char *line;
    char *prompt = "mydash>";

    using_history(); /* enable readline history mechanism */
    while ((line=readline(prompt)))
    {
        printf("%s\n",line);
        add_history(line); /* add current line to history list */
        free(line);
    }
    exit(0);
}
```

NOTE: You may need to install the readline packages for the above code to compile.
The readline documentation is a good starting point on how to use the readline library. Pay close attention to memory ownership. Remember that C does not have a garbage collector.

Command execution

The mini-shell accepts one command per line with arguments. It should accept up to 2048 arguments to any command. The shell will parse each command that is entered and then execute any valid command typed in with arguments (using the **execvp()** system call). The execvp() system call performs a search for the command using the PATH environment variable. This will simplify your programming since you do not have to search for the location of the command. You will need to consult the man pages for the fork() ,exec(), wait() and other related system calls.

Empty commands

If the user just presses the Enter key, then the shell displays another prompt. If the user types just spaces and then presses the Enter key, then the shell displays another prompt as this is also an empty command. Empty commands should not cause a segfault or memory leak.

Handing EOF

The mini-shell terminates normally on receiving the end of input (Under Linux and bash, this would normally be **Ctl-d** for you to test your mini-shell).

Exit command

Include a built-in commands **exit** that terminates the shell normally.

Change Directory Command

Add a built-in command **cd** to allow an user to change directories. You will need to use the **chdir()** system call. The **cd** command without any arguments should change the working directory to the user's home directory. You must first use **getenv** and if getenv returns NULL your program should fall back to the system calls **getuid()** and **getpwuid()** to find out the home directory of the user. Make sure to print an error message if the **cd** command fails.

Prompt

The default prompt for mydash may be anything you like. However the shell checks for an environment variable **DASH_PROMPT** using the **getenv()** system call. If the environment variable is set, then it uses the value as the prompt. Set the environment variable using the export command in the terminal, as shown next.

```
export DASH_PROMPT=”myprompt>”
```

See the man page for **getenv()** for more details.

Return value on exit

Your mini-shell should return a status of 0 when it terminates normally and a non-zero status otherwise.

Show version

When the shell is started with a command line argument -v, it prints out the version of the project and then quits.

```
[user@localhost p1(master)]$ ./mydash -v
mydash: Version 1: Revision 9e0501   (author: User user@localhost.com)
[user@localhost p1(master)]$
```

For larger projects in industry, you would often have a CI (Continuous Integration) system setup to handle this aspect but we won't use something that involved for our project.

We will use the first 6 characters of the git SHA code (the long hash code that uniquely represents a version of your project repo). You will have your **Makefile** auto-generate a small function **git_version()** that will return the first 6 characters of the git SHA code.

Valgrind

You must use valgrind and resolve memory errors and leaks as much as possible. Invoke valgrind as follows:

```
valgrind --leak-check=full ./mydash
```

Note that the readline library gives many valgrind errors that are not relevant. You can suppress these errors. See the man page or online documentation for valgrind for instructions on suppressing external errors. A sample suppression file named **valgrind.supp** has been provided for you in your repo.

Documentation

README.md

Use the `README.md` template to document your overall project. It must contain a *reflection* and *self assessment* section.

Doxygen docs

Use the software documentation tool doxygen to generate the documentation for the code. A sample useful config file for doxygen is provided in the file `doxygen-config` in backpack. Make sure to examine it and modify to customize to your project. Then the documentation can be generated with the command:

```
doxygen doxygen-config
```

The doxygen tool is available on the lab machines and can be installed on CentOS using the following command:

```
yum install doxygen
```

Testing Plan

The backpack.sh script in the project folder contains simple smoke tests that you can run for a quick sanity check. This would be your starting point. At the minimum, you must provide a comprehensive list of test cases. This should be in separate file. Please include a reference to this file in the manifest in your README.md file. A test harness or unit tests would be even better but not required at this point.

Submission

There are two places to submit for this project 1) you must submit your code to backpack and 2) you must submit the SHA1 hash to blackboard.

Submit to Backpack

Required files to submit through git (backpack)

1. Makefile
2. mydash.c
3. mydash.h
4. README.md

5. valgrind.supp
6. **Any other files required to build and run the project**

Push your code to a branch for grading

Run the following commands

1. make clean
2. git add <file …> (on each file!)
3. git commit -am "Finished project"
4. git branch shell_p1
5. git checkout shell_p1
6. git push origin shell_p1
7. git checkout master

Check to make sure you have pushed correctly

- Use the command **git branch -r** and you should see your branch listed (see the example below)

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/shell_p1
```

Submit to Blackboard

You must submit the sha1 hash of your final commit on the correct branch. Your instructor needs this in order to troubleshoot any problems with submission that you may have.

- git rev-parse HEAD

Grading Rubric

Provided via backpack.