SIENA*college*
Computer Science

Computer Science 330
Operating Systems
Siena College
Fall 2022

Programming Project 3: Basic Unix Shell Implementation

Due: 11:59 PM, Friday, November 11, 2022

In this programming project, you will write the first version of a C program called the Roger Bacon Shell (`rbsh`), a mini command shell interpreter. `rbsh` is similar to familiar Unix shells such as the Bourne shell (`sh`) the Bourne-Again shell (`bash`), and C shell (`csh, tcsh`). By the time you get through the final version, you will learn about process creation, pipes, input/output redirection, background process management, signals, and interrupt handling, and gain extensive experience with C.

You may work alone or in groups of size 2 or 3 on this programming project. However, in order to make sure you learn the material and are well-prepared for the exams, those who work in a group should either collaborate closely while completing the project.

There is a significant amount of work to be done here. It will be difficult if not impossible to complete the assignment if you wait until the last minute. You will need to ask questions. A slow and steady approach will be much more effective.

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `basicshell-proj-yourgitname`, for this programming project. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 9:20 AM, Monday, November 7, 2022. This applies to those who choose to work alone as well!

Requirements

Like the Unix shells you use every day, `rbsh` should issue a prompt (below, it is "`shell#`"), at which it reads commands from the user and executes them.

Your shell should interpret the following commands and provide the following functionality:

- `exit`: exit from the shell.
- `help`: display a message listing usage of all commands.
- Execute a command (program on the file system). As in the shells you use, the command should be located according to the `PATH` environment variable. Appropriate choice of `exec`

function will simplify this for you. The arguments following the command should be passed as arguments to the command.

For example,

```
shell# cat cat.c
```

should execute `cat` with one argument, `cat.c`.

- Implement a builtin `cd` command. It must be given 0 or 1 command-line parameters. If no command-line parameter is given, it returns the current directory to the user's home directory (note: you can get this with the function call `getenv("HOME")`). Otherwise, it changes to the directory given as the command-line parameter. It is an error to have more than one command-line parameter, and it should be reported as such.
- Errors should be reported meaningfully.

Notes

- The `Makefile` provided in the repository is a GNU makefile, not a standard makefile, so if you are running on noreaster, you will need to use the `gmake` command to run the GNU version of make.
- The repository includes a C file `rbsh.c` that provides your basic input loop using the `readline(3)` function that gives you some nicer keyboard input handling than a standard `scanf` call. Readline is in the Standard C Library. Note that this will require an extra link flag "`-lreadline`", and on some systems it also requires "`-lncurses`".
- When using the provided `Makefile`, you can specify `DEBUG=1` on the command line to compile with debugging flags enabled. These can be used to include debugging output between `#ifdef` and `#endif` directives. You can see an example of this in the starter code. Note that there are two preprocessor symbols defined: `PARSE_DEBUG` and `PROC_CONTROL_DEBUG`. You should protect your debugging that has to do with parsing the text read into the `cmd_line` variable into its parts (a task that is already non-trivial in this version and will become a significant challenge later when you add pipes and I/O redirection) with `PARSE_DEBUG`. Use printouts protected by `PROC_CONTROL_DEBUG` to report when processes are created and complete. Again, this will become much more complicated once you have pipes and background execution later. Good use of this verbose/debugging mode is essential for, well, debugging. And it's part of the grading, so do it.
- Some string utility functions that were useful in the reference solution are included in `stringfuncs.c` and `stringfuncs.h`. You are welcome to use them or ignore them as you see fit.
- The `system(3)` system call must **not** be used.

- The system calls that you should use are `fork(2)`, a variant of `exec(3)`,`waitpid(2)` and `chdir(2)`.
- You may find the `strsep(3)` function to be useful to break your command down into tokens.
- You may assume the builtin commands `exit` and `help` ignore any command-line parameters given after those commands.
- The builtin commands run in the shell's process. There is no `fork`ing needed when processing those commands.
- A shell may run for a long time, and can be susceptible to memory leaks. Be careful about memory management.
- `gcc`'s `-Wall` flag will report additional compiler warnings. This can help you find some of the more subtle bugs that are so common in C code before they have a chance to cause their subtle problems. If you aren't absolutely certain that a given warning is harmless, fix it! Your submitted code should have no warnings reported when compiled with `-Wall`.
- Take good advantage of git and GitHub to facilitate collaboration. Commit and push often.
- Executables, both standard and with debugging outputs turned on, for the reference solution are in `/home/cs330/basicshell` on noreaster.

Submission

Commit and push!

For this project, an in-person demonstration and code review is required for each group following submission. This can occur during office hours or by appointment.

Grading

This assignment will be graded out of 60 points.

Feature	Value	Score
Documentation	10	
Verbose mode output	5	
`exit` and `help` commands	2	
Running a command with no arguments	15	
Running a command with arguments	20	
Builtin `cd` command	3	
In-person demo and code review	5	
Total	60	