SIENA*college*
Computer Science

Computer Science 330
Operating Systems
Siena College
Fall 2020

Programming Project 3: The Roger Bacon Shell

Part 1 Due: 4:00 PM, Friday, October 23, 2020
Part 2 Due: 4:00 PM, Friday, October 30, 2020

In this programming project, you are to write 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`). 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 in groups of size 2 or 3. Collaboration within a group is, of course, unrestricted. You may discuss the program with members of other groups, but what you turn in must be your own group's work. Groups must be formed no later than 4:00 PM, Friday, October 16, 2020, and be confirmed by all group members having write access to the group's repository on GitHub and all names in the `README.md` file. All group members will be assigned the same grade for the lab. There are many subtasks that can be carved off and assigned to group members, so everyone is encouraged to join a group.

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. A slow and steady approach will be much more effective.

To help encourage this approach, there are two parts for your work on this project. The basic functionality to run a single command is part 1, and is due at 4:00 PM, Friday, October 23, 2020. The remainder is due at 4:00 PM, Friday, October 30, 2020. Please refer to the grading guidelines at the end of this document for the specific functionality required for your part 1 submission.

Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository, which will be named `shell-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 4:00 PM, Friday, October 16, 2020. 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 disk). 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`

  - *Input and output redirection* should be implemented.
    For example,

    ```
    shell# cat < cat.c > myfile.c
    ```

    should cause the `cat` program to read from `cat.c` and write to the file `myfile.c`.

    ```
    shell# ls -l >> dirlistings.txt
    ```

    should cause the output of `ls -l` to be appended to the end of the existing file `dirlistings.txt`.
    An individual command may only redirect input once and output once, but those redirections may be specified in any order.

    ```
    shell# > alines grep -i < Makefile a
    ```

    should be interpreted the same as the more usual

    ```
    shell# grep -i a < Makefile > alines
    ```

  - *Pipes* should be implemented.
    - For example,

      ```
      shell# cat cat.c | wc > count.txt
      ```

      should cause the output of `cat cat.c` to be the input of `wc > count.txt`.
    - You should allow a sequence of pipes to be specified:

      ```
      shell# ls -l *.c | grep "Oct 31" | wc -l
      ```

      The program should be able to handle a sequence of pipes of any length.
    - Only the first command in a pipeline may have input redirection. Only the last command in a pipeline may have output redirection. Redirection of other commands should be reported as an ambiguous command line.

  - Typing <ctrl-c> should abort a command being run, but not cause `rbsh` to terminate.

- A list of commands separated by semicolons should be executed in sequence.
- As with the familiar Unix shells, appending an `&` to the end of your command line will execute the command in the background.
  - An `&` may be anywhere in a command line, but it is treated like a `;` in that anything on the command line following an `&` should be treated as a new command.

    For example,

    ```
    shell# sleep 5 & sleep 5
    ```

    should launch one instance of `sleep 5` in the background, then a second in the foreground.
  - Typing <ctrl-c> should not kill the shell or any commands running in the background.
  - When any background command terminates, it should be reported.

    ```
    shell# sleep 55 &         ; invoke sleep in background
    shell# cat < hello.c      ; give other commands
    shell# ...                ; other commands
    shell# ...                ; other commands
    [2] "sleep" terminated    ; sleep command is done
    ```

  - All backgrounded processes should be maintained in a process table by `rbsh`, so that the program name is displayed when it terminates, and for use in the `jobs` and `kill` commands.

- `jobs`: Displays all the active background programs that have been started from this shell, along with their *id*s. (This id need not be that same as the actual process id. For example, it could be the index into your process table).

  ```
  shell# jobs
  PID     Name
  [0]    mycat < myfile.c > newfile.c
  [1]    sleep 20
  [4]    grep mysh < doc | wc
  ```

- `kill`: Without any arguments, prints a usage statement:

  ```
  shell# kill
  Usage: kill <pid> [<pid> ...]
  ```

  Otherwise it kills the processes with the specified ids and displays the processes killed.

```
shell# kill 4
[4] "grep" Killed
shell# kill 3 7
[3] "cat" Killed
[7] "wc" Killed
```

- Implement a builtin `cd` command.
- Errors should be reported meaningfully.
- Additional functionality of your group's choosing should also be implemented (see also the Submission and Evaluation section below).

  Ideas for extra functionality include:

  - command history: `history` command, `!command`, `!!`, `^` modification of previous command
  - control structures in the shell (`for`, `while`, `if`)
  - aliases
  - user-specified prompts (as in `bash`, `tcsh`)
  - More advanced redirection and pipes: `<<`, `>&`, `|&`
  - <ctrl-z> trapping and corresponding job control (`fg`, `bg` commands)
  - Parenthesis-delimited subshells

  See `builtin(1)` for more ideas.

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.
- There are some string utility functions that were useful in the reference solution that 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)`, `signal(3)`, `kill(2)`, `open(2)`, `dup2(2)`, `close(2)`, `pipe(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`, `help`, `jobs`, `kill`) stand alone on a command line (no pipes, I/O redirection).
- A shell may run for a long time. 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!
- A verbose/debugging mode is essential for, well, debugging.
- Take good advantage of git and GitHub to facilitate collaboration.
- The following might be a good order to tackle the required functionality.
  - `exit` and `help` commands
  - run a command (with no argument passing, no redirection, no pipes)
  - run a command with argument passing
  - run a command with input and output redirection
  - run a command in the background
  - `jobs` command
  - `kill` command
  - <ctrl-c> trapping
  - trapping termination of background processes
  - pipes
  - `;`- or `&`-separated commands
  - `cd` command
  - extra functionality
- On noreaster you can find my version of `rbsh` under `/home/cs330/shell/rbsh`.

Submission

Commit and push!

Grading

The functionality required for the first deadline is denoted by a "Part 1".

This assignment will be graded out of 100 points.

Feature	Value	Score
Documentation	12	
`exit` and `help` commands (Part 1)	2	
Running a command with no arguments (Part 1)	15	
Running a command with arguments (Part 1)	10	
Builtin `cd` command (Part 1)	2	
Running a command with input/output redirection	10	
Launching a command in the background	10	
Launching a sequence of jobs separated by semicolons or ampersands	8	
Builtin `jobs` command	4	
Builtin `kill` command	3	
Correct trapping of Ctrl-C keystroke	4	
Reporting termination of backgrounded jobs	4	
Pipeline of 2 processes	6	
Arbitrarily long pipeline of processes	4	
Extra functionality	6	
Bonus functionality	0	
Total	100	

Here are some possibilities for the 6 points for extra functionality. Functionality up to 10 additional points beyond the required 6 will be considered for bonus credit.

- 2-point enhancements: aliases, more advanced redirection and pipes: `<<`, `>&`, `|&`, user-specified prompts.
- 4-point enhancements: command history, control structures in the shell (`for`, `while`, `if`), modification of previous command with `^`.
- 6-point enhancements: Ctrl-Z trapping and corresponding job control (`fg`, `bg` commands).
- Please ask about other possible enhancements. Point values will be based on level of difficulty.