Program 4: A Simple Shell

Re-submit Assignment

Due Monday by 11:59pm **Points** 100 **Submitting** a file upload
File Types zip, tar, gz, and tar.gz **Available** until Apr 22 at 11:59pm

Program 4: A Simple Shell

Update 2019-04-15: The late penalties for this assignment have been reduced from the usual 10% per day to 5% per day. The last day to submit it is still Monday, April 22 (for a 25% penalty rather than 50%).

Programming assignments in CS 270 are **individual work**. You may discuss approaches with other students, but may not share code or pseudocode for the assignment. If do get ideas from somebody, or use snippets of code from elsewhere, you **must cite the source** in your documentation.

Background

A *shell* is an application program that runs other programs on behalf of the user (see chapter 8.4.6 of the textbook). A shell reads in *command lines* from the user or from a *script* file, executing the program specified on each line. Most commands run by a shell are external executables to be run in separate processes, but a few are *built-in commands*, which are implemented in the shell itself.

More advanced shells also provide programming capabilities: variables, conditional statements, loops, and so on, but you will not be going that far in this assignment: that is more of a matter for CS 441G (compilers) or CS 450G (programming languages).

Some popular shells:

- `sh`, the "Bourne shell", the traditional shell provided by Unix systems since the 1970s.
- `csh`, an interactive shell originally provided with the BSD variant of Unix.
- `bash`, the "Bourne again shell", the standard interactive shell on GNU/Linux, OS X, and many other modern Unix systems. As the name might imply, `bash` is a more advanced, but backwards-compatible, reimplementation of the Bourne shell.
- `zsh`, a mostly Bourne-compatible shell with many added features, even more than `bash`.
- `cmd.exe`, the standard shell on Windows, based on the earlier DOS shell `command.com`. Scripts in `cmd` are called *batch files*.
- `PowerShell`, a more advanced shell for Windows systems.

Your task in this assignment is to implement a simple shell that supports executing external command (programs), performing simple redirections, and evaluating a few built-in commands.

Provided files

The following files may be downloaded as a compressed tar archive: **pa4-provided.tar.gz**

- Makefile for building the project.
- shell.h: definition of the `command` structure, and prototypes for the provided functions `parse_command` and `free_command`.
- parser.c: implementation of `parse_command` and `free_command`.
- tests/: a subdirectory containing test inputs, expected outputs, and helper commands.
- run-tests: a sh script to run the tests and compare your shell's output with the expected output.

Specifications

Your shell should support being called in one of two ways:

- `./shell`: When called with no arguments, your program should read commands from `stdin` until it encounters an end-of-file (or a command that causes it to exit). When reading from stdin, your shell should print the string `"shell> "` (with no newline) to `stderr` as a prompt before every user input.
- `./shell scriptfile`: When called with an argument, your program should open that file and read commands from it until end-of-file (or a command that causes the shell to exit). When reading from a script file, your shell should *not* print a prompt. If there was an error opening or reading from the script (and not simply end-of-file), your program should print a message and exit with a nonzero status.

Whichever way it is called, your shell should do the following in a loop:

1. Read a line from `stdin` or the script. If you encountered an end of file, exit the shell with status 0. If you encountered another kind of error, report an error message to stderr and exit the shell with a nonzero status.
   (You may impose a maximum length on input lines, but the maximum must be at least 512 bytes, you should document any such limitation, and of course you should avoid buffer overflows.)
2. Call `parse_command` (in `parser.c` (http://www.cs.uky.edu/~neil/485/pa/4/parser.c)) to parse the line into a structure containing an array of arguments, an optional input redirection, and an optional output redirection.
3. Based on argument 0 (the command name), either execute one of the built-in commands, or fork and execute an external program.
4. Call `free_command` to free the memory returned by `parse_command`.

Parsing commands

You do not have to parse commands yourself, but can rely on the provided `parse_command` function. This function takes a string argument (the command line to be parsed) and returns a pointer to a `struct command` containing three members:

- `args` is an array of pointers to C strings, terminated by a `NULL` pointer after the last argument. `args[0]` is the command name, and the subsequent members are the command's arguments. This array is suitable for passing as the second argument to `execvp()`.
- `in_redir` is the name of the file to redirect standard input from, or `NULL` if stdin should not be redirected.

- `out_redir` is the name of the file to redirect standard output to, or `NULL` if stdout should not be redirected.

Note that, if the command line was blank, `parse_command` returns a `command` struct with no arguments or command name (`args[0] == NULL`). Your program should handle such empty commands (by ignoring them).

You *must* call `free_command` on the pointer returned by `parse_command`, after executing the command and before reading the next one.

Built-in commands

Your shell should support the following built-in commands. You may ignore any redirections specified with these commands. If a built-in command encounters an error, it should *not* terminate the entire shell, but rather report that error and return to the main loop.

`cd dir`

The `cd` command, when given one or more arguments, should use the `chdir` system call to change the shell's current working directory as specified by the first argument `dir`.

`cd`

If `cd` is called with no arguments, it should change to the user's home directory. You can find the home directory with `getenv("HOME")`, but beware: that function might return `NULL` if the user has removed that environment variable. If that is the case, report an error message (but do not terminate the program).

`setenv variable value`

The `setenv` command sets an environment variable. These variables control various aspects of the standard library and the programs you execute: the user's home directory, the path in which to search for executables, etc.

Your implementation should call the `setenv()` library function, passing a nonzero value as the `overwrite` argument. You should report any error reported by `setenv`. If the user did not supply any arguments, you should report an error and *not call `setenv`*.

`setenv variable`

When called with only one argument, `setenv` should unset the named environment variable by calling the `unsetenv()` library function.

`exit`

The `exit` command, with any number of arguments, causes the shell to exit with exit status 0 (indicating success).

(missing)

If the input did not contain a command name (for example, if the user entered only spaces), your shell should ignore the input.

External commands and redirection

Any other value of the 0th argument indicates an external command. The syntax of an external command is:

```
    command arg1 arg2 ... [< infile] [> outfile]
```

To execute an external command, you will need to:

1. Call `fork` to create a child process.
2. In the child process, open the input and/or output files, and move them to file descriptor 0 (stdin) or 1 (stdout) using `dup2()`. If there was an error, exit the child process.
   - For an input redirection, you should open the file with the flag `O_RDONLY`.
   - For an output redirection, you should open the file with the flag `O_WRONLY`, along with appropriate flags to create new files or truncate existing files. See `man 2 open` for more details.
   - It's generally a good idea to close the original file descriptor after `dup2()` succeeds, to avoid taking up file descriptor slots even after executing the external program. This will not affect the file descriptor (0 or 1) that you duplicated *to*.
3. In the child process, call `execvp()` to execute the command. This library function is like `execve()`, but if the program is a bare name without slashes, it searches for the program in the path (the PATH environment variable).
4. In the parent process, call `waitpid()` to wait for the child process to terminate.
5. In the parent process, print a message to `stderr` based on the child process's exit status (see `man waitpid` for more information):
   - If the child exited normally with status 0, do not print a message.
   - If the child exited normally with nonzero status, print for example `"Command returned 3"`
   - If the child was killed by a signal, print a message such as `"Command killed: Interrupt"`. You can use the `strsignal()` function to convert a signal number returned by the `WTERMSIG` macro into a C string such as "Interrupt".

There is one more trick to executing external commands. If you follow the above algorithm, and the user tries to kill the command with Ctrl-C, that will *also* kill your shell. You should not allow that to happen. Inside the parent process, when you are about to wait for the child, you can use the `signal()` function to ignore the `SIGINT` (keyboard interrupt) signal. After the child process has finished, use `signal()` again to reset the handler to the default, so that Ctrl-C at the shell prompt *does* still kill your shell.

Constraints and technical restrictions

1. Your shell may be written in either C or C++. You may use any version of the C or C++ standard, but you must edit the Makefile to use the correct version flag in `CFLAGS`:
   - `-std=c89`: old-school ANSI C (`gcc` default).
   - `-std=c99`: the most commonly-used version of modern C.
   - `-std=c11`: the most recent C standard.
   - `-std=c++98`: old-school C++ (`g++` default).
   - `-std=c++11`: modern C++.

  - `-std=c++14`: even more modern C++.
  - `-std=c++17`: the most up-to-date version of C++.
2. Your program should compile with the `-Wall` compiler flag (enable all warnings), without producing any compiler or linker warnings or errors.
3. It must be possible to compile and run your program on the class virtual machines. The sequence of commands `make clean; make` should build your program without any manual intervention.
4. You must check the result of every system call and library function that might return an error. Check the "Return value" and/or "Errors" sections of the appropriate manpages to determine whether errors are possible, and how to detect them.
5. Unless it terminates with an error message, your program should free all memory that it allocates. In particular, every call to `parse_command` should be matched by a call to `free_command` on the same pointer.
6. Your program should be free of buffer overflows and other undefined behavior.
7. When executing an external command, your shell must use `fork()`, `execvp()`, and `waitpid()`. You should *not* use `system()`, `posix_spawn()`, or similar functionss.
8. Redirections should be perfomed in the child process. The child process should use the `open()` system call directly (not `fopen` or `ifstream`), followed by `dup2()` to put the file on the appropriate file descriptor (0 for stdin, 1 for stdout).
9. Otherwise, you may use any C or C++ standard library functions you want. You may not link in other libraries.
10. Output redirection `>` should create a new file if necessary, or truncate an existing file. Input redirection `<` should not.
11. Your shell should produce no output other than the prompt, error messages, and reports on the exit status of external commands. All of those outputs should be sent to `stderr`, not `stdout`.

Error handling

If any system call or library function fails, your program should report the reason for a failure in an error message to `stderr`, and either (1) handle and correct the error, (2) terminate the program (or the child process), or (3) (in built-in commands only) end the command without terminating the program.

Your program explictly does *not* have to check for or handle errors in output functions (`fprintf`, `cerr <<`, etc.).

If your program does terminate because of an error, it should indicate this by calling `exit` with a nonzero status code.

Your error messages should generally include two parts, separated by a colon. First, a message indicating the context of the error in the program: that is, what it was trying to do. Then, a message indicating the nature or cause of the error, usually as indicated by the `errno` (http://en.cppreference.com/w/c/error/errno) variable. For example:

```
Error executing badcmd: No such file or directory
Could not open /private for output: Permission denied
```

For consistency with the messages produced by other programs, you should obtain the second part of the message by using one (or both) of the following C library functions:

- `perror()` (http://en.cppreference.com/w/c/io/perror) to print the error message associated with `errno`, along with a prefix you supply, to stderr.
- `strerror()` (http://en.cppreference.com/w/c/string/byte/strerror) to obtain the error message as a (C) string, for use with `fprintf`, `cerr`, etc.

Among the library functions and system calls you are likely to use, at least the following require error checks and messages:

- `malloc` and `calloc`.
- `fopen` and `fclose` of the script file, if provided.
- Reading commands from the script file. End-of-file is not an error, but there might be actual read errors (for example if the user closes the terminal that the shell is reading from). You can use the C library functions `ferror()` and `feof()` to tell the difference.
- `fork`ing a child process.
- `open` of redirected files. Note that terminating the program because of an error in the child process will (correctly) not kill the parent process.
- `dup2` of the redirected file handle.
- `execvp` to execute a program from the child process. In fact, an error is the only reason an `exec` function would ever return.
- `waitpid` in the parent process.
- `chdir` in your `cd` builtin command (do not terminate the shell!)
- `setenv` and `unsetenv` in your `setenv` builtin command (do not terminate the shell!)

Most of these functions indicate errors by returning either -1 (for functions that return an integer) or `NULL` (for functions that return a pointer), but there are a few exceptions, particularly in the `stdio` functions: check the manual pages to be sure.

Remember to send error messages to `stderr` (`cerr` in C++), *not* `stdout`.

Test cases

A script and collection of test cases are provided to help test your program. To run the tests, first make sure you are in the directory that contains your executable named `shell` and the `tests` subdirectory, then do:

```
./run-tests
```

You should see a list of 25 tests, with "SUCCESS" or "FAILURE" appearing after each. If the test failed, the next line is one of three messages that explains the error:

Files YOUROUT and EXPECTED differ, please review.

The script did not produce the expected output to stdout. The message lists the file containing the script's actual output in your shell (for example, `tests/output.12`), and the file containing the expected output (`tests/expected/test.12.redir-out-failure.in`).

Note that the output here is the stdout of the commands that your shell executes. If your shell itself prints any output to stdout (as opposed to stderr), that will result in a test failure. Also, the output must match *exactly*: even a difference in whitespace or line endings will cause a failure.

Received unexpected error output (see YOURERR)

Your shell was not expected to print any messages to stderr for this script, but it did. The listed file contains the data your shell printed to stderr.

If you print a prompt even when given a script file as input, or if you left debugging output in your shell, that will cause this failure.

Missing expected error output (see EXPECTEDERR)

Your shell was expected to print an error message to stderr for this script, but it did not. The listed file contains the expected error messages.

Note that `run-tests` does not check the exact contents of the error messages, only whether there were error messages. It is still your responsibility to make sure the error messages are reasonable.

Each of the twenty-five tests has three associated files. For example, test 12:

- `tests/test.12.redir-out-failure.in` contains the script being tested, which is provided to your shell as a command-line argument.
- `tests/expected/test.12.redir-out-failure.out` contains the expected stdout output of your shell for this test.
- `tests/expected/test.12.redir-out-failure.err` contains the expected stderr output of your shell for this test. As noted above, you do not have to exactly match the error output.

If a test fails, its output and errors are saved to `tests/output.NN` and `tests/error.NN`, where `NN` is the number of the test. If the test succeeds, these two files are deleted.

Documentation

Your submission should include a `README` file. This should be a **plain text** file with at least the following sections:

- **Author**: Your name and email address, and the date the program was finished.
- **Contents**: A list of all the files in your submission, with a brief one-line description of each. For example, "main.c: implementation of `main` and helper functions", or "README: this file".
- **Running**: Brief instructions explaining how to compile and run your program.
- **Implementation notes**: Describe and justify any design decisions you made when implementing your shell. For example: which functions are responsible for executing commands? How does the shell determine whether a command is a built-in? Did you implement any helper functions for error-handling or other purposes?

- **Limitations**: Describe any limitations of your shell. For example, the provided parser supports a maximum of 30 arguments, so longer command lines cannot be used.

  If you have any failing test cases, memory errors, etc., also describe those problems here. Where do the problems occur, what steps did you take to try to solve them, and what further steps would you take if you had more time?

  Likewise, if you noticed in your own testing any situations that your shell does not handle well, even if they are not covered by the test cases, describe them here.

- **References**: If you discussed the design of your program with anyone, list them here and describe their contribution. For example:

  > The design of the `frobulate()` function benefitted from discussions with my tutor J. Random Hacker, who suggested doing the bit-twiddling before the loop rather than inside the loop.

  Likewise, if you used code snippets from sites such as Stack Overflow, describe what you used, **explain how it works in your own words**, and provide the name of the author and the URL where they posted the code. For example:

  > The data-copying loop in `perform_magic()` is based on code written by C. Guru at: `http://answermyprogrammingquestions.com/0xc0debeef/`. The loop casts the character pointer to a integer pointer and uses that to copy four bytes of data at a time. If the number of bytes was not divisible by four, the code then copies the remaining 1-3 bytes individually.

  Programming assignments are expected to be your own work, so any borrowed code should be **very small** relative to the total size of your program. You may not borrow code from or share code with other UK students at all.

What to submit

Submit a zip or tar.gz file containing a directory with the following files:

- The source code for your implementation of the shell. This may be as many or as few source or header files as you like, but don't put everything in `parser.c`.
- A Makefile that builds an executable named `shell`. This may be the provided Makefile, a modified version of it, or a brand new Makefile.
- A **plain text** README file as described in the "Documentation" section above.

To make a .tar.gz archive of the directory program4, you can use a command like:

```
tar czvf cs270-program4.tar.gz program4/
```

To make a .zip archive:

```
zip -r cs270-program4.zip program4/
```

Submit your .tar.gz or .zip file to this assignment.

Grading

This assignment will be scored out of 100 points:

- 50 points: 2 points per test case passed.
- 15 points: program compiles with no warnings under `-Wall`.
- 15 points: there is appropriate error handling code for every system call and library function that might fail. Error messages are appropriate, containing both the context and the cause of the error.
- 10 points: documentation is complete, readable, and error-free.
- 10 points: code shows **good style** **(http://www.cs.uky.edu/~raphael/checklist.html)** (variable names, formatting, function parameters, etc).

In addition, up to 15 bonus points are available, for implementing background jobs and a few related built-in commands: See below.

Hints and FAQs

1. If your shell's prompt is not appearing, make sure you call `fflush(stderr)` after printing it to flush the output buffer (`cerr << flush;` in C++). Otherwise, the output might not appear until a newline is printed.
2. When calling `open` with `O_CREAT` to create a file, you must also supply a third argument for the file mode (permissions). Use the value `0666` (octal, hence the leading zero) so that the file will have the correct default permissions (see `man umask` to see why this doesn't result in files that are writable by everybody). For example:

   ```
   if ((fd = open(filename, O_RDWR | O_CREAT, 0666)) < 0) ...
   ```

   If you dislike magic numbers, you may instead write `0666` as `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH`.

Bonus: background commands

For up to 15 bonus points, add support to your shell for *background commands*, indicated by an ampersand (`&`) at the end. When your shell executes a background command, it should *not* wait for the child process to terminate. Instead, it should add that process's command and PID to a list (which you might store in a global array for example), and print the *job number* (the index into that list) for future reference.

To assist with using background commands, you should also implement the following built-in commands:

`jobs`

List the job number, PID, and command name of each running background command. For example:

```
shell> jobs
[1] 12371: sleep
[2] 12373: cp
```

`fg num`

Bring the command with job number `num` back to the *foreground*, by waiting for it with `waitpid`. If `num` is not a valid job number, or that process has already terminated, report an error message.

Like the parent process of an external command, this command should ignore `SIGINT` while it is waiting, and should print a message if the process's exit status is nonzero.

`fg`

If no argument is provided, `fg` operates on the most-recently-started job that is still running. Note that this might not be the last background command executed, because that one may have already terminated.

To implement the bonus, you will have to modify `struct command` to add a flag indicating whether this is a foreground or background command. You will probably also need to modify `parse_command` to set the flag. Note that the `&` should *not* be passed to the command as an argument!

Your program should not leave zombie processes around. That means that you will need to use the `signal()` function to establish a signal handler for `SIGCHLD`. In your signal handler you can call `wait` to get the process ID and exit status of the process that exited, remove the job from the jobs table, then print a message to stderr with the exit status or terminating signal.

```
[1] 12371: sleep - Command returned 0.
[2] 12373: cp - Command killed: Hangup
```

Finally, note that the command name in the `struct command` will be freed by the `free_command` function. So your table of background jobs will need to maintain a copy of this string, which you can make with the `strdup` function. The string will need to be freed after the child is waited for.