A3 Processes and Pipes

Assignment 3: (10%): Processes and pipes

Due: Electronically, by 10:00 PM on **Monday** March 16, 2015

Introduction

In this assignment, you will be writing your own mini-shell. The shell should be able to launch commands in new processes, as well as support the built-in commands `cd` and `exit`. Your shell will also support standard input, output and error redirection, as well as the pipe (`|`) operator to chain commands. You will be provided with starter code to help you get started with the implementation.

Follow the instructions in the handout below carefully, so that you implement your code correctly and we receive all the right files.

Please read over this entire handout **at least twice** before you get started, so that you have a clear picture of what you have to do.

Your first step should be to check out your repository, as in the previous assignments, and make sure that you can commit to it.

You must do your assignment on one of the CDF lab machines or remotely through SSH at `cdf.toronto.edu`, so please **make sure your code works on CDF machines!** (It is highly likely that parts of this code will not work on Windows, and possibly not on OSX.)

Objective

In this assignment, your task is to implement a basic shell that allows the execution of command line tools and programs. Early in the course, we worked with the Unix shell, and more recently we have studied Unix processes and pipes. It is time to put all of this together and write your own mini-shell that can `almost` give bash a run for its money.

Your shell should allow running command line programs that have zero, one or multiple arguments. As we have seen in the lectures, commands can be *built-in* functions where no process is spawned, simple *non-builtin* commands where a process is spawned to execute them, and complex *non-builtin* commands where an operator such as a pipe (`|`) is used to chain commands. You should build this functionality into your shell one at a time using the steps recommended below:

- Step 1: read through all of the starter code, especially the comments. You will not be able to make progress until you understand the basic structures and the control flow.

- Step 2: built-in commands (i.e. `cd` and `exit`)
- Step 3: simple commands with no redirection
- Step 4: simple commands with redirection
- Step 5: complex commands (i.e. those that involve the pipe operator)

Starter code

The starter code will be committed to your repository in the a3 directory. You should not have to modify anything except `shell.c`, in the places indicated with "TODO" comments.

The starter code provided will guide you step by step in solving this assignment.
The `parser.c` and `parser.h` files deal with parsing commands and storing them in data structures (see below). The `shell.c` and `shell.h` files contain functions that process commands and execute them accordingly. You should read the code and understand it, then proceed to fill in all the `TODO` spots left for you to implement.

Built-in commands

The only built-in commands you need to support are **`exit`** and **`cd`**.

The `exit` command should exit the current shell session, and the `cd` command is used to change the current working directory.

The `cd` command can accept both relative and absolute (i.e. relative to root) paths. (See the comments in the code for hints on how to implement this.) For example:

- `cd csc209/assignments`
- `cd /home/bogdan/csc209/assignments/`
- `cd ../john/csc209/assignments/`
- `cd ../../home/jane/csc209/assignments/`

Note: you do **NOT** have to support anything involving expanding environment variables or special symbols such as `$HOME`, `~`, or `*`.

Simple and complex commands

Your next task is to implement simple commands and then complex commands. Before you can get started on these tasks, you need to understand the basic structures used to implement commands.

Your shell should allow both *simple commands*, and *complex commands* that can contain any number of simple commands chained using special operators. The only special operator you will implement for this assignment is the pipe (`|`) operator. (Other operators are described at the end of this document in the Optional section.)

Your shell should also support the following redirection operators:

- Redirection of standard input from a file: `<`
- Redirection of standard output to a file: `>`
- Redirection of standard error to a file: `2>`
- Redirection of both standard output and standard error to a file: `&>`

Note: you do **NOT** have to support redirection in append-mode (`>>` and `2>>`).

Basic Structures

Simple commands are stored in the following structure:

```
typedef struct simple_command_t {
    char *in, *out, *err;    /* Files for redirection, optional */
    char **tokens;           /* Program and its parameters */
    int builtin;             /* Builtin commands, e.g., cd */
} simple_command;
```

This is used for simple commands, involving a single program and no pipe chaining. A simple command can use redirection: `in`, `out` and `err` are strings containing the names of the files where the corresponding redirection should occur (`stdin` should be redirected to the file whose name is stored in `in`, `stdout` redirected to the file whose name is stored in`out`, etc.).

The `builtin` flag indicates whether the command is builtin or non-builtin. Valid values for the `builtin` flag are: 0 (non-builtin), 1 for `cd`, and 2 for `exit`. You should use the predefined macros `BUILTIN_CD` and `BUILTIN_EXIT` instead of hardcoding the values 1 and 2. This way your code can benefit from more clarity and extensibility.

The `tokens` member is an array of strings (array of pointers to characters) where each element of the array is a word in the command. The last element of the array is NULL.

For example, `ls -l` will be split into 2 tokens `ls` and `-l`, followed by a NULL pointer.

Hint: the NULL pointer simplifies your task when using certain `exec` calls.

Remember that complex commands are simple commands chained together with the pipe (`|`) operator. The pipe operator allows chaining of commands, by sending the output of the first command as input to the second command. You have already seen a few examples of this in the lectures.

Complex commands are defined in the following structure `command`:

```
typedef struct command_t {
     /* Two commands chained using a special operator. In turn, each one can
      * contain multiple commands itself; */
    struct command_t *cmd1, *cmd2;

    /* Simple command, no pipe */
    simple_command* scmd;
```

```
    /* In this assignment, consider only "|". */
    char oper[2];
} command;
```

This is a general form of a command. A command can be simple (no special operators), in which case `scmd` is non-NULL, or it can be complex in which case `scmd` will be NULL.

If it is a complex command, `cmd1` and `cmd2` are the two chained commands. In turn, `cmd1` and `cmd2` might be complex commands themselves. In the starter code, you are given a hint on how to deal with them by using recursion. The `oper` string is used to store the special `|` operator for chaining `cmd1` and `cmd2`.

In this assignment, you are NOT allowed to use the `popen` or `system` functions.

For file I/O, redirection, and processes, you are working at a low level using file descriptors. Consequently, you should only use POSIX functions, like `open` and `close`. Functions such as `fopen` or `fclose` are not to be used in this assignment.

Sample shell commands you should try (just a few examples)

Here are some commands. Notice that the prompt includes the current working directory. The first command, `shell` is typed at Bash to start our shell. The rest of the commands are run from our shell (not Bash).

- `$ ./shell`
- `/home/bogdan/csc209/a3/> pwd`
- `/home/bogdan/csc209/a3/> ls -l`
- `/home/bogdan/csc209/a3/> ls -l > file.out`
- `/home/bogdan/csc209/a3/> cat file.out`
- `/home/bogdan/csc209/a3/> ls -l | wc -l`
- `/home/bogdan/csc209/a3/> ls -l | grep file1`
- `/home/bogdan/csc209/a3/> ls -l > filelist.out | wc -l`
- `/home/bogdan/csc209/a3/> cat filelist.out`
- `/home/bogdan/csc209/a3/> ls -l | wc -l > wc.out`
- `/home/bogdan/csc209/a3/> cat wc.out`
- `/home/bogdan/csc209/a3/> ls -l > ls.out | wc -l > wc.out`
- `/home/bogdan/csc209/a3/> cat ls.out`
- `/home/bogdan/csc209/a3/> cat wc.out`
- `/home/bogdan/csc209/a3/> rm -f ls.out wc.out`
- `/home/bogdan/csc209/a3/> ls -l | grep bogdan | wc -l`
- `/home/bogdan/csc209/a3/> ps aux | grep bogdan | grep bash | grep -v grep | wc -l`

- `/home/bogdan/csc209/a3/> ls -l | grepp bogdan` (should say something like: "grepp: No such file or directory")
- `/home/bogdan/csc209/a3/> lsa -l | grep bogdan` (similarly, regarding mistyped "lsa")

Also, make sure to run several commands involving `cd`, as shown earlier in the handout.

Once you are done testing these and more, one cool thing you could try is to run your shell program within your own shell. This way you can create a new shell session as a child process of your own shell. Again, the first `shell` is being typed at Bash to start-up our shell, and then the rest is typed at our shell (not Bash).

- `$ ./shell`
- `/home/bogdan/csc209/a3> ./shell`
- `/home/bogdan/csc209/a3> echo Woo-hoo, new shell within the shell`
- `Woo-hoo, new shell within the shell`
- `/home/bogdan/csc209/a3> exit`
- `/home/bogdan/csc209/a3> echo I am back to the parent shell`
- `I am back to the parent shell`
- `/home/bogdan/csc209/a3> exit`
- `$`

After the second `exit`, you exit the parent shell as well, so you are back to your regular Bash shell.

Submission instructions

CODE THAT DOES NOT COMPILE WILL GET 0 MARKS! It's better that you submit code with partial functionality than code that does not compile at all.

To get full marks, your code must be well-documented. Please include comments wherever appropriate, and indent your code properly.

Please do not submit executables or anything other than source files and header files. You may choose to submit a `README.txt` file if you want to add additional information that may be useful to the marker such as implementation decisions that you made or an indication of optional additions you implemented. If you were not able to complete the assignment, the `README.txt` file should include an explanation of the parts that you believe to be working or not working.

Please make sure that you have committed all your source files, the `makefile` and your `README.txt` (if you have created one).

Submission checklist:

You can make your life a lot simpler by ensuring that your submission is complete. When you first sit down to do some work on this assignment, check out the repository. Notice which files are already in the repository. Every time you create a new file, run `svn add` immediately to ensure that future commit operations will commit the new file as well.

Use version control as it was meant to be used. Commit your work frequently -- at least once every 2-3 hours that you are working on the assignment. This will prevent any last minute svn problems, and provide a record of your work.

Here is a list of things that you should do **on CDF** at least an hour before the assignment deadline:

- Create a temporary directory in your account (not one that is a subdirectory of your working directory for your repository).
- Check out your repository into this empty directory to be sure that the correct files have been committed.
- Run `make` to be sure that the `shell` program compiles without warnings or error.
- Run `shell` and try a few commands to make sure that it works as you expected.

Optional Features

None of the following are required for credit. These are optional features that you might have fun implementing when you finish the required functionality described above.

Variables and Substitutions

- Expanding environment variables (preceded by $), such as in paths to commands or files.
- Defining new environment variables for the current shell session.
- Wildcard substitution, e.g., `ls *.txt`
- Double quotes, single quotes, and back quotes

More Operators

- `&` operator sends a process to the background, as we've seen in lectures. However, it can be used to run 2 commands in parallel as well. For example, the following commands will be run in parallel:
  ```
  ls & whoami
  ```
- The `;` operator allows running multiple commands in one run, sequentially, e.g.,
  ```
  ls ; whoami ; pwd
  ```
- The `&&` operator is a logical AND operator that lets you execute a second command iff the first command runs successfully (the exit status of the first command is zero), e.g.,

```
$ mkdir newfolder && cp file.txt newfolder
```
(If the `mkdir` is successful, copy a file into it)

- The `||` operator is a logical OR operator that lets you execute a second command iff the first command fails (the exit status of the first command is greater than zero).