CSC209 Summer 2014

Software Tools and Systems Programming

Assignment 3 (10%): Processes and pipes

Due Date: 11:59pm on Thursday, July 17, 2014

Objective

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 built-in commands, such as "cd" or "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 a starter code, which should help you get started in the implementation.

Important Notes:

- You must submit your solution by the due date on MarkUs.
- All programs are to be written using *C* and compiled by *gcc*.
- Your programs should be tested on *CDF* before being submitted (in fact, it's STRONGLY ENCOURAGED that you write your assignment on a Linux system to begin with). Failure to test your programs on *CDF* will result in a mark of 0 being assigned.
- CODE THAT DOES NOT COMPILE WILL GET 0 MARKS!
- To get full marks, your code must be well-documented.
- Include your information (student ID, full name, login) inside **all** C programs as a comment at the top of the file.

What To Submit

When you have completed the assignment, submit your source code on MarkUs. Please do not submit executables or anything other than source files and header files. Please include a file "README.txt" containing your name, student number, CDF login and the number of grace days used. The Readme can also include other explanations (for example, regarding implementation decisions), if you wish.

In this assignment, your task is to implement a basic shell that allows the execution of command line tools and programs. Since you have learnt about shell programming, and how to work with 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.

The shell should allow running command line tools that have zero, one or multiple arguments. As we've seen in the lectures, commands can be *non-builtin* (a process is spawned to execute them), and *built-in* (no process is spawned). Your shell should allow for both types of commands.

To simplify your task, from the built-in commands, you only have to support **"cd"** and **"exit"**.
The exit command will exit the current shell session, and the cd command will be used to change the current working directory. The cd command takes both relative and absolute (relative to root) paths. For example:

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

You do NOT have to support anything involving expanding environment variables, or special symbols, e.g.,

```
$ cd $HOME/csc209/assignments/
$ cd ~/csc209/assignments/
$ cd ~bogdan/csc209/assignments/
```

A shell generally supports a set of special operators, for various purposes like background execution or chaining commands:

- The '&' 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, if 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, if the first command fails (the exit status of the first command is greater than zero), e.g.,
  `$ [ -d newfolder ] || mkdir newfolder`
  (If the folder "newfolder" does not exist, create it)
- The '|' (pipe) operator allows chaining of commands, by sending the output of the first command as an input to a second command. You've already seen a few examples of this one in the lectures and labs.

Out of these special operators, <u>for the scope of this assignment, you will only have to implement the pipe '|' operator.</u>

Your shell should allow both *simple commands,* which do not contain such operators, AND *complex commands* that can contain any number of simple commands, chained using pipe operators.

Optionally, you are welcome to try implementing more operators, although there are certain tricky aspects you will have to be careful about, in order to correctly implement their semantics, as well as their priority order (from most to least priority: '|', '&&' and '||', '&', ';').

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 output and error: '&>'

You do **NOT** have to support redirection in append-mode ('>>' and '2>>'), but you are welcome to extend your shell with these options as well if you wish.

Your shell does **NOT** have to support the following things (although you are welcome to extend your shell to make it more awesome, if you wish):

- 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

Starter code

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) that allow you to handle them easier. 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.

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 (or other special operators). 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. The 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 the values 1 and 2. This way your code can benefit from more clarity.

The "tokens" are basically the words in the command, followed by a NULL token. For example, "ls -l" will be split into 2 tokens "ls" and "-l", followed by a NULL pointer. The command "ls –l | wc -l" will contain 5 tokens (parsing out all whitespaces, like blankspace and tab), followed by a NULL token.
Hint: the NULL token simplifies your task when using execlp and execvp calls).

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

```
typedef struct command_t {
        struct command_t *cmd1, *cmd2;  /* Two commands chained using a special operator;
                                                In turn, each one can contain multiple commands itself; */
        simple_command* scmd; /* Simple command, no pipe */
        char oper[2];   /* In this assignment, consider only "|". Optionally, implement others: ";", "&&", etc. */
} command;
```

This is a general form of a complex command. A complex 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 can 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 dedicated for storing the special operator chaining cmd1 and cmd2.

Try to first go through the code, and to understand the provided starter code. Next, try to follow the detailed instructions in the code, and fill in the code in the spots we left for you to complete, indicated by "**TODO**" comments.

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.

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

```
$ pwd
$ ls -l
$ ls -l > file.out
$ cat file.out
$ ls -l | wc –l
$ ls -l | grep file1
$ ls -l > filelist.out | wc -l
$ cat filelist.out
$ ls -l | wc -l > wc.out
$ cat wc.out
$ ls –l > ls.out | wc –l > wc.out
```

```
$ cat ls.out
$ cat wc.out
$ rm –f ls.out wc.out
$ ls -l | grep bogdan | wc –l
$ ps aux | grep bogdan | grep bash | grep –v grep | wc –l
$ ls –l | grepp bogdan    (should say something like: “grepp: No such file or directory”)
$ lsa –l | grep bogdan     (similarly, regarding mistyped “lsa” command)
```

Also, make sure to run several commands involving ‘cd’, as shown in the handout (only the ones you should implement).

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.

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

After the second “exit”, you exit the parent shell as well, so you are back to your regular terminal prompt.