ECS 150: Project #1 - Simple shell

Joël Porquet

UC Davis, Winter Quarter 2017, version 1.0

General information

Due before **11:59 PM, Friday, January 27th, 2017**.

You will be working with a partner for this project.

The reference work environment is the CSIF.

Specifications

Note that the specifications for this project are subject to change at anytime for additional clarification. Make sure to always refer to the latest version.

Introduction

The goal of this project is to understand important UNIX system calls by implementing a simple shell called **sshell**. A shell is a command-line interpreter: it accepts input from the user under the form of command lines and executes them.

In the following example, it is the shell that is in charge of printing the *shell prompt*, understanding the supplied command line (redirect the output of executable program `ls` with the argument `-l` to the input of executable program `cat`), execute it and wait for it to finish before prompting the user for a new command line.

```
jporquet@pc10:~/ecs150 % ls -l | cat
total 12K
-rw------- 1 jporquet users 11K 2017-04-04 11:27 ASSIGNMENT.md
jporquet@pc10:~/ecs150 %
```

Similar to well-known shells such as *bash* or *zsh*, your shell will be able to:

1. execute user-supplied commands with optional arguments
2. offer a selection of builtin commands
3. redirect the standard input or standard output of commands to files
4. pipe the output of commands to other commands
5. put commands in the background
6. handle a history of recent commands

A working example of the simple shell can be found on the CSIF, at `/home/jporquet/ecs150/sshell_ref`.

Constraints

The shell must be written in C, be compiled with GCC and only use the standard functions provided by the GNU C Library (aka `libc`). *All* the functions provided by the `libc` can be used, but your program cannot be linked to any other external libraries.

Your source code should follow the relevant parts of the Linux kernel coding style and be properly commented.

Phase 0: preliminary work

In this preliminary phase, copy the skeleton C file `/home/jporquet/ecs150/sshell.c` to your directory. Compile it into an executable `sshell` and run it.

```
jporquet@pc10:~/ % ./sshell
...
```

What does it do?

0.1 Understand the code

Open the C file `sshell.c` and read the code. As you can notice, we use the function `system()` to run the command `/usr/bin/date -u` (which displays the current time in UTC format) and print the *raw* status value that the command returned to the standard output (`stdin`).

The problem is that `system()` is too high-level to use for implementing a realistic shell. For example, it doesn't let you redirect the input or output, or run commands in the background.

Useful resources for this phase:

- http://man7.org/linux/man-pages/man3/system.3.html
- https://www.gnu.org/software/libc/manual/html_mono/libc.html#Running-a-Command

0.2 Makefile

Write a simple Makefile that generates an executable `sshell` from the file `sshell.c`, using GCC.

The compiler should be run with the `-Wall` and `-Werror` options.

There should also be a `clean` rule that removes any generated files and puts the directory back in its original state.

Useful resources for this phase:

- https://www.gnu.org/software/make/manual/make.html
- https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

Phase 1: running commands the hard way

Instead of using the function `system()`, modify the program in order to use the *fork+exec+wait* method.

In a nutshell, your shell should fork and create a child process; the child process should run the specified command with exec while the parent process waits until the child process has completed and the parent can collect its exit status.

The output of the program execution should be as follows:

```
jporquet@pc10:~/ % ./sshell
Tue Apr  4 21:12:18 UTC 2017
+ completed '/usr/bin/date -u' [0]
jporquet@pc10:~/ %
```

There are a couple of non-apparent differences between this output and the output of the provided skeleton code:

- The information message following the execution of the command is printed to `stderr` and not `stdout`.

  This can be verified by redirecting the error output to `/dev/null` and checking that the information message is not printed anymore:

  ```
  jporquet@pc10:~/ % ./sshell 2>/dev/null
  Tue Apr  4 21:12:18 UTC 2017
  jporquet@pc10:~/ %
  ```

- The printed status (i.e. `0` in the example above) is not the full *raw* status value anymore, it is the *exit* status only. Refer to the *Process Completion Status* section of the `libc` documentation to understand how to extract this value.

Useful resources for this phase:
https://www.gnu.org/software/libc/manual/html_mono/libc.html#Processes

Phase 2: read commands from the input

Until now, your program is only running a hard-coded command. In order to be interactive, the shell should instead read commands from the user and execute them.

In this phase, modify your shell in order to print the *shell prompt* `'sshell$ '` (without the quotes but with the trailing white space) and read a complete command line from the user. We assume that the maximum length of a command line never exceeds 512 characters.

Since it would be annoying for the user to always type the complete paths of the commands to execute (e.g. `/usr/bin/ls`), programs should be searched according to the `$PATH` environment variable. For that, you need to carefully choose which of the `exec` functions should be used.

For this phase, you can assume that the user can only enter the name of a program (e.g. `ls`, `ps`, `date`, etc.) without any argument.

Example of output:

```
sshell$ date
Tue Apr  4 14:09:08 PDT 2017
+ completed 'date' [0]
sshell$
```

Useful resources for this phase:

- https://www.gnu.org/software/libc/manual/html_mono/libc.html#Line-Input
- https://www.gnu.org/software/libc/manual/html_mono/libc.html#Executing-a-File

Error management

In case of user errors (e.g. invalid input, command not found, etc.), the shell should display a meaningful error message on `stderr` and wait for the next input, but it should **not** die.

Example of a command not found:

```
sshell$ toto
Error: command not command
sshell$
```

The only reason for which the shell is allowed to die (with an exit value of `1`) is if a system call actually fails. For example, `malloc()` fails to allocate memory or `fork()` fails to spawn a child.

Phase 3: arguments

In this phase, you must add to your shell the ability to handle command lines containing programs and their arguments.

A command is then defined as the name of a program, followed by optional arguments, each separated by white spaces (at least one, but can also be more than one). In order to simplify your implementation, you can assume that a command will never have more than 16 arguments (name of the program included).

For this phase, you will need to start *parsing* the command line in order to interpret what needs to be run. Refer to the `libc` documentation to learn more about strings in C (and particularly sections 5.1, 5.3, 5.4, 5.7 and 5.10): https://www.gnu.org/software/libc/manual/html_mono/libc.html#String-and-Array-Utilities

Example of commands which include arguments (with more or less white spaces separating arguments):

```
sshell$ date -u
Tue Apr  4 22:07:03 UTC 2017
+ completed 'date -u' [0]
sshell$ date                      -u
Tue Apr  4 22:46:41 UTC 2017
+ completed 'date                      -u' [0]
```

```
sshell$
```

At this point, and if you have not already, it probably is the right time to think of how you could represent commands using a data structure. After all, a `struct` object in C is nothing different than a C++/Java class without methods. But such an object can still contain fields that contain the object's properties, and C++-like methods can be implemented as simple functions that receive objects as parameters.

Example:

```
/* C++ class */
class myclass {
    int a;

    mymethod(int b) {
        a = b;
    }
};

/* Equivalent in C */
struct myobj {
    int a;
};

myfunc(struct myobj *obj, int b) {
    obj->a = b;
}
```

Hint: the result of parsing the command line should be the instance of a data structure which contains all the information necessary to run the specified command.

Phase 4: builtin commands

Usually, when a user enters a command, the related program is an *external* executable file. For example, `ls` refers to the executable file `/usr/bin/ls` while `ip` refers to `/usr/sbin/ip`.

For some commands, it is preferable that the shell itself implements the command instead of running an external program. In this phase, your shell must implement the commands `exit`, `cd` and `pwd`.

For simplicity, you can assume that these builtin commands will never be called with incorrect arguments (i.e. no arguments for `exit` and `pwd` and exactly one argument for `cd`).

`exit`

Receiving the builtin command `exit` should cause the shell to exit properly (i.e. with exit status `0`). Before exiting, the shell must print the message '`Bye...`' on `stderr`.

Example:

```
jporquet@pc10:~/ % ./sshell
sshell$ exit
Bye...
jporquet@pc10:~/ % echo $?
0
```

`cd` and `pwd`

The user can change the *current working directory* (i.e. the directory the shell is currently "in") with `cd` or display it with `pwd`.

Example:

```
sshell$ pwd
/home/jporquet/ecs150
+ completed 'pwd' [0]
sshell$ cd ..
+ completed 'cd ..' [0]
sshell$ pwd
/home/jporquet
+ completed 'pwd' [0]
sshell$ cd toto
Error: no such directory
+ completed 'cd toto' [1]
sshell$
```

Useful resources for this phase:
https://www.gnu.org/software/libc/manual/html_mono/libc.html#Working-Directory

Phase 5: Input redirection

The standard input redirection is indicated by using the meta-character < followed by a file name. Such redirection implies that the command located right before < is to read its input from the specified file instead of the shell's standard input (that is from the keyboard if the shell is run in a terminal).

Example:

```
sshell$ cat file
titi
toto
+ completed 'cat file' [0]
sshell$ grep toto<file
toto
+ completed 'grep toto<file' [0]
sshell$ grep toto < tata
Error: cannot open input file
sshell$ cat <
Error: no input file
sshell$
```

Note that the input redirection symbol can or not be surrounded by white spaces.

Phase 6: Output redirection

The standard output redirection is indicated by using the meta-character > followed by a file name. Such redirection implies that the command located right before > is to write its output to the specified file instead of the shell's standard output (that is on the screen if the shell is run in a terminal).

Example:

```
sshell$ echo Hello world!>file
+ completed 'echo Hello world!>file' [0]
sshell$ cat file
Hello world!
+ completed 'cat file' [0]
sshell$ echo hack > /etc/passwd
Error: cannot open output file
sshell$ echo >
Error: no output file
sshell$
```

Note that the output redirection symbol can or not be surrounded by white spaces.

Phase 7: Pipeline commands

So far, a command line could only be composed of one command (name of program and optional arguments). In this phase, we introduce the notion of pipeline of commands.

The pipe sign is indicated by the meta-character **|** and allows multiple commands to be connected to each other. When the shell encounters a pipe sign, it indicates that the output of the command located before the pipe sign must be connected to the input of the command located after the pipe sign. There can be multiple pipe signs on the same command line to connect multiple commands to each other.

Example:

```
sshell$ echo Hello world! | grep Hello|wc -l
1
+ completed 'echo Hello world! | grep Hello|wc -l' [0][0][0]
sshell$
```

Note that there is no limit on the number of commands part of a pipeline as long as it fits into the command line (i.e. 512 characters).

The information message must display the exit value of each command composing the pipeline separately. This means that commands can have different exit values as shown in the example below (the first command succeeds while the second command fails with exit value `2).

```
sshell$ echo hello | ls file_that_doesnt_exists
ls: cannot access 'file_that_doesnt_exists': No such file or directory
+ completed 'echo hello | ls file_that_doesnt_exists' [0][2]
sshell$
```

In a pipeline of commands, only the first command can have its input redirected and only the last command can have its input redirected.

```
sshell$ cat file | grep toto < file
Error: mislocated input redirection
sshell$ echo Hello world! > file | cat file
Error: mislocated output redirection
sshell$
```

Hint: for this phase, you will probably need to think of a data structure that can be used to represent a job (i.e. a pipeline of one or more commands).

Useful resources for this phase (sections 15.1 and 15.2):
https://www.gnu.org/software/libc/manual/html_mono/libc.html#Pipes-and-FIFOs

Phase 8: Background commands

Up until now, when the user enters a command line and the shell executes it, the shell waits until the specified job is completed before it displays the prompt again and is able to accept a new command line to be supplied.

The ampersand character & indicates that the specified job should be executed in the background. In that case, the shell should not wait for the job's completion but immediately display the prompt and allow for a new command line to be entered. The background sign may only appear as the last token of a command line.

When a background job finally completes, the shell should display the exit status of all the job's commands right before a new prompt is being displayed.

Example:

```
sshell$ sleep 1&
sshell$ sleep 5
+ completed 'sleep 1&' [0]
+ completed 'sleep 5' [0]
sshell$ echo > file & | grep toto
Error: mislocated background sign
sshell$
```

Trying to exit while there are still running jobs in the background should be considered a user error.

Example:

```
sshell$ sleep 5&
sshell$ exit
Error: active jobs still running
+ completed 'exit' [1]
sshell$ <Return>
... more than 5 seconds later ...
sshell$ <Return>
+ completed 'sleep 5&' [0]
sshell$ exit
Bye...
```

Useful resources for this phase: https://www.gnu.org/software/libc/manual/html_mono/libc.html#Process-Completion

Deliverable

Content

Your submission should contain, besides your source code, the following files:

- `AUTHORS`: first name, last name, student ID and email of each partner, one entry per line formatted in CSV (fields are separated with commas). For example:

  ```
  $ cat AUTHORS
  Homer,Simpson,00010001,hsimpson@ucdavis.edu
  Robert David,Martin,00010002,rdmartin@ucdavis.edu
  ```

- `REPORT.md`: a description of your submission. Your report must respect the following rules:

  - It must be formatted in markdown language as described in this Markdown-Cheatsheet.

  - It should contain no more than 300 lines and the maximum width for each line should be 80 characters (check your editor's settings to configure that).

  - It should explain your high-level design choices, details about the relevant parts of your implementation, how you tested your project, the sources that you may have used to complete this project, and any other information that can help understanding your code.

  - Keep in mind that the goal of this report is not to paraphrase the assignment, but to explain how you implemented it.

- `Makefile`: a Makefile that compiles your source code without any errors or warnings (on the CSIF computers), and builds an executable named `sshell`.

The compiler should be run with the options `-Wall -Werror`.

There should also be a `clean` rule that removes generated files and puts the directory back in its original state.

Your submission should be empty of any clutter files such as executable files, core dumps, etc.

Git

Your submission must be under the shape of a Git bundle. In your git repository, type in the following command (your work must be in the branch `master`):

```
$ git bundle create sshell.bundle master
```

It should create the file `sshell.bundle` that you will submit via `handin`.

You can make sure that your bundle has properly been packaged by extracting it in another directory and verifying the log:

```
$ cd /path/to/tmp/dir
$ git clone /path/to/sshell.bundle -b master sshell
$ cd sshell
$ ls -l
...
$ git log
...
```

Handin

Your Git bundle, as created above, is to be submitted with `handin` from one of the CSIF computers:

```
$ handin cs150 p1 sshell.bundle
Submitting sshell.bundle... ok
$
```

You can verify that the bundle has been properly submitted:

```
$ handin cs150 p1
The following input files have been received:
...
$
```

Academic integrity

You are expected to write this project from scratch, thus avoiding to use any existing source code available on the Internet. You must specify in your `REPORT.md` file any sources of code that you or your partner have viewed to help you complete this project. All class projects will be submitted to MOSS to determine if pairs of students have excessively collaborated with other pairs, or have used the work of past students.

Excessive collaboration, or failure to list external code sources will result in the matter being transferred to Student Judicial Affairs.