ECS 150: Project #1 - Simple Shell

Prof. Joël Porquet-Lupine

UC Davis, Fall Quarter 2020

Changelog

> **NOTE:** *The specifications for this project are subject to change at anytime for additional clarification. Make sure to always refer to the **latest** version.*

- v1: First publication

General information

- Due before **11:59 PM, Thursday, October 22rd, 2020**.
- You will be working with a partner for this project.
- The reference work environment is the CSIF.

Objectives of the project

The objectives of this programming project are:

- Reviewing most of the concepts learned in previous programming courses: data structures, file manipulation, command line arguments, Makefile, etc.
- Discovering and making use of many of system calls that UNIX-like operating systems typically offer, especially syscalls belonging to the following categories: processes, files, and pipes.
- Understanding how a shell works behind the hood, and how processes are launched and configured.
- Writing high-quality C code by following established industry standards.

Program description

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. Well-known UNIX shells include for example `bash` (default shell on Ubuntu) and `zsh` (default shell on MacOS).

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 `date` to the input of executable program `tr` with arguments `2` and `1`), execute it, and wait for it to finish before displaying a completion message and prompting the user for a new command line.

```
sshell@ucd$ date | tr 2 1
Thu 09 Jan 1010 06:40:47 PM PST
```

```
+ completed 'date | tr 2 1' [0][0]
sshell@ucd$
```

Your shell will provide the following set of core features:

1. Execution of user-supplied commands with optional arguments
2. Selection of typical builtin commands
3. Redirection of the standard output of commands to files
4. Composition of commands via piping

In addition, your shell will provide the following set of extra features:

1. Output redirection appending
2. Simple `ls`-like builtin command

Constraints

Your code 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 adopt a sane and consistent coding style and be properly commented when necessary. One good option is to follow the relevant parts of the Linux kernel coding style.

Assessment

Your grade for this assignment will be broken down in two scores:

Auto-grading: ~60% of grade

Running an auto-grading script that tests your program and checks the output against various inputs

Manual review: ~40% of grade

The manual review is itself broken down into different rubrics:

- Submission : ~10%
- Report file: ~40%
- Makefile: ~10%
- Quality of implementation: ~30%
- Code style: ~10%

The `sshell` specifications

Commands and command line

When the shell is ready to accept input from the user, it must print '`sshell@ucd$` ' – without the quotes but with the trailing white space. At this point, the user can type a single command, or a pipeline of commands. Each command starts with the name of a program (e.g. `ls`, `ps`, `cat`, `echo`) and is optionally followed by arguments, separated by one or more white spaces, to be passed to the program (e.g. `ls -l`).

The shell may assume that:

- The maximum length of a command line never exceeds 512 characters.
- A program has a maximum of 16 arguments.
- The maximum length of individual tokens never exceeds 32 characters.

> **TIP:** Since it would be annoying for the user to always type the complete paths of the commands to execute (e.g. `/bin/ls`), programs should be searched according to the `$PATH` environment variable.

After the shell launches the command(s) corresponding to the command line, it waits until all the commands have finished. Only then, the shell displays a completion message on `stderr`, which contains the return values of the completed commands, and displays a new prompt for a new input command line to be supplied.


```
sshell@ucd$ echo Hello world
Hello world
+ completed 'echo Hello world' [0]
sshell@ucd$
```


In addition to programs and their arguments, the shell must understand certain specific meta-characters (e.g. `>`, `|`, etc.) as described in the next sections.

Builtin commands

When a user enters a command, the related program is usually an *external* executable file. For example, `ls` refers to the executable file `/bin/ls` while `fdisk` refers to `/sbin/fdisk` (this is abstracted by the `$PATH`, as mentioned above).

For some commands, it is preferable, or even necessary, that the shell itself implements the command instead of running an external program. As part of the core features, your shell must implement the commands `exit`, `cd` and `pwd`.

> **NOTE:** `pwd` can actually be implemented by an external program (and is often provided as such on most UNIX systems), but we decide for this project that it should be provided by the shell itself.

For simplicity, you may 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@ucd$ exit
Bye...
+ completed 'exit' [0]
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@ucd$ pwd
/home/jporquet/ecs150
+ completed 'pwd' [0]
sshell@ucd$ cd ..
+ completed 'cd ..' [0]
sshell@ucd$ pwd
/home/jporquet
+ completed 'pwd' [0]
sshell@ucd$
```


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@ucd$ echo Hello world>file
+ completed 'echo Hello world>file' [0]
sshell@ucd$ cat file
Hello world
```

```
+ completed 'cat file' [0]
sshell@ucd$
```

Piping

The pipe sign is indicated by using the meta-character `|` and allows multiple commands to be connected to each other within the same command line. 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. We assume that there can be up to three pipe signs on the same command line to connect multiple commands to each other.

Example:

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


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

```
sshell@ucd$ 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@ucd$
```

You can assume that builtin commands will never be called as part of a pipeline.

Error management

There are three types of errors that the shell needs to deal with:

1. Failure of library functions.
2. Errors during the parsing of the command line.
3. Errors during the launching of the command line.

Failure of library functions

If a library function fails, for example if `malloc()` is unable to allocate memory or if `fork()` is unable to spawn a child, then the shell is allowed to terminate its execution right away. You may optionally use `perror()` to report the cause of the failure.

Parsing errors

If an incorrect command line is supplied by the user, the shell should only display an error message on `stderr`, discard the invalid input and wait for a new input, but it should **not** die.

If a command line contains more than one parsing error, the leftmost one should be detected first and reported.

Here are all the potential parsing errors for the core features:

- `"Error: too many process arguments"`

```
sshell@ucd$ ls 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Error: too many process arguments
sshell@ucd$
```

- `"Error: missing command"`

```
sshell@ucd$ > file
Error: missing command
sshell@ucd$ | grep hi
Error: missing command
sshell@ucd$ ls |
Error: missing command
sshell@ucd$
```


- `"Error: no output file"`

```
sshell@ucd$ echo >
Error: no output file
sshell@ucd$
```

- `"Error: cannot open output file"`

```
sshell@ucd$ echo hack > /etc/passwd
Error: cannot open output file
sshell@ucd$
```

- `"Error: mislocated output redirection"`

```
sshell@ucd$ echo Hello world > file | cat file
Error: mislocated output redirection
sshell@ucd$
```

Launching errors

Launching errors are usually not detected during parsing, but when the shell actually tries to execute the parsed command line. Like parsing errors, launching errors should not cause the shell to die. The command launch should gracefully fail and the shell should ask for a new input.

Here are all the potential launching errors for the core features:

- `"Error: cannot cd into directory"`

```
sshell@ucd$ cd doesnotexist
Error: cannot cd into directory
+ completed 'cd doesnotexist' [1]
sshell@ucd$
```

- `"Error: command not found"`

```
sshell@ucd$ windows98
Error: command not found
+ completed 'windows98' [1]
sshell@ucd$
```

Extra features

This quarter, you have two additional features that the shell must implement.

Output redirection appending

When using the simple output redirection, the specified output file is truncated (i.e., reset to an empty size before being written to), as illustrated in the following example.

```
sshell@ucd$ echo toto > output
+ completed 'echo toto > output' [0]
sshell@ucd$ cat output
toto
+ completed 'cat output' [0]
sshell@ucd$ echo titi > output
+ completed 'echo titi > output' [0]
sshell@ucd$ cat output
titi
+ completed 'cat output' [0]
sshell@ucd$
```


The new sequence of meta-characters `>>` indicates that the specified output file should be *appended* to, and not truncated.

```
sshell@ucd$ echo toto >> output
+ completed 'echo toto >> output' [0]
sshell@ucd$ cat output
```

```
toto
+ completed 'cat output' [0]
sshell@ucd$ echo titi >> output
+ completed 'echo titi >> output' [0]
sshell@ucd$ cat output
toto
titi
+ completed 'cat output' [0]
sshell@ucd$ echo tata > output
+ completed 'echo tata > output' [0]
sshell@ucd$ cat output
tata
+ completed 'cat output' [0]
sshell@ucd$
```


This sequence of meta-characters shares the same set of parsing and launching errors as the regular output redirection.

`ls`-like builtin command

The new builtin command `sls` lists the current directory's contents, and prints each entry along with its size, one per line.

```
sshell@ucd$ sls
project_1.md (24578 bytes)
Makefile (97 bytes)
project_1.html (282441 bytes)
```

```
+ completed 'sls' [0]
sshell@ucd$
```

For simplicity, you may assume that this builtin commands will never be called with incorrect arguments. However, it can have a potential launching error:

- `"Error: cannot open directory"`

```
sshell@ucd$ ls -ld test
d--x--x--x 2 joel joel 4096 Sep 22 10:43 test
+ completed 'ls -ld test' [0]
sshell@ucd$ cd test
+ completed 'cd test' [0]
sshell@ucd$ sls
Error: cannot open directory
+ completed 'sls' [1]
sshell@ucd$
```


Reference program and testing

A reference program can be found on the CSIF, at `/home/cs150jp/public/p1/sshell_ref`. You need to copy it to your own directory before using it.

> **IMPORTANT:** Since we will use auto-grading scripts in order to test your program, make sure that your shell implementation generates the **exact same output** as the reference program.

Testing can be automatically performed by comparing the output of your shell implementation with the output of the reference shell, for the same input.

```
jporquet@pc10:~/ $ echo -e "echo Hello\nexit\n" | ./sshell >& your_output
jporquet@pc10:~/ $ cat your_output
sshell@ucd$ echo Hello
Hello
+ completed 'echo Hello' [0]
sshell@ucd$ exit
Bye...
+ completed 'exit' [0]
jporquet@pc10:~/ $ echo -e "echo Hello\nexit\n" | ./sshell_ref >& ref_output
jporquet@pc10:~/ $ diff your_output ref_output
jporquet@pc10:~/ $
```


> **TIP:** Using a simple bash script, you can easily turn every single example contained in this document into a test case.

Suggested work phases

> **NOTE:** The following phases are merely a suggestion to help you progress step by step, but you are under no obligation to follow them!
>
> Some additional information is provided so it is still worth reading through.

Phase 0: preliminary work

A skeleton C file is provided in `/home/cs150jp/public/p1/sshell.c` to help you start this project. Copy it to your directory. Compile it into an executable named `sshell` and run it.

```
jporquet@pc10:~/ $ ./sshell
sshell@ucd$ echo Hello
Hello
Return status value for 'echo Hello': 0
sshell@ucd$ exit
Bye...
jporquet@pc10:~/ $
```


It's already a very simple shell!

0.1 Understand the code

Open the C file and read the code. As you can notice, we use the rudimentary function `system()` to run commands. The problem is that `system()` is too high-level to use for implementing a realistic shell. For example, it doesn't support output redirection or piping.

Useful resources for this phase:

- `man system`
- GNU Libc – 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 -Wextra` (enable all warnings, and some more) and `-Werror` (treat all warnings as errors) flags.
- 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:

- GNU Make – Manual
- GNU GCC – Warning options

Phase 1: running simple commands the hard way

Instead of using the function `system()`, modify the program in order to use the *fork+exec+wait* method, as seen in lecture. For this phase, start by focusing on simple commands with no arguments.

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 is able to collect its exit status.

In order to automatically search programs in the `$PATH`, you simply need to carefully choose which of the `exec` functions should be used (see first link below).

```
sshell@ucd$ date
Fri 10 Jan 2020 12:31:13 AM PST
+ completed 'date' [0]
sshell@ucd$
```

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

- The completion message following the execution of the command is printed to `stderr` and not `stdout`.
- 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 (see second link below).

Useful resources for this phase:

- GNU Libc – Executing a file
- GNU Libc – Processes

Phase 2: Arguments

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

For this phase, you will need to really 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): GNU Libc – String and array utilities.

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

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

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

At this point, and if you have not already, it probably is the right time to think of how you could represent commands using proper data structures. 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;
}
```

The result of parsing the command line should be the instance of a data structure which contains all the information necessary to launch the specified command (so that the original command line does not have to be parsed again).

Phase 3: builtin commands

Implement the rest of the builtin commands, namely `pwd` and `cd`.

Useful resources for this phase:

- GNU Libc – Working directory

Phase 4: Output redirection

Implement output redirection. There are two steps to this implementation

1. Parsing of the output redirection (meta-character and output file) from the command line. Note that the output redirection is only an instruction to the shell, it should not be transmitted to the program as arguments.
2. Manipulation of the `stdout` file descriptor prior to running the specified program, so that the program prints into the file and not to the terminal.

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

If the output file already exists, it should be truncated. See options to `open()` to figure out the correct flag.

Phase 5: Pipeline commands

Implement piping. 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).

Note that in a pipeline of commands, only the last command may have its output redirected.

Useful resources for this phase (sections 15.1 and 15.2): GNU Libc – Pipes and FIFOs.

Phase 6: Extra features

Once you have completed the set of core features, add the two extra features to your shell.

The output redirection appending feature should only require very slight tweaking of your output redirection logic, in order to append to the output file rather than truncate it.

For the implementation of builtin command `sls`, you need to figure out how to open a directory and scan its contents. See this useful resource: GNU Libc – Accessing directories.

Submission

Content

> **IMPORTANT:** Your submission should be empty of any clutter files (such as executable files, core dumps, backup files, `.DS_Store` files, and so on).

Your submission should contain the following files:

- `sshell.c`, and any other C files containing your code.

  > **NOTE:** It is not necessarily recommended to split your code into multiple files for a project of that size.

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


```
$ cat AUTHORS.csv
00010001,jdupont@ucdavis.edu
00010002,mdurand@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 200 lines and the maximum width for each line should be 80 characters (check your editor's settings to configure it automatically –please spare yourself and do not do the formatting manually).
  - 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 -Wextra -Werror`.

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

Gradescope

Gradescope will be opened for submission on Tuesday, October 20th at 0:00. At that time, you will be able to submit your project as a Git repository.

> **IMPORTANT:** There should be only one final submission per group, submitted by one of the two partners.
>
> The other partner should be added to the submission as “group member”.

Academic integrity

Novelty

You are expected to write this project **from scratch**.

Therefore, you cannot use any existing source code available on the Internet, or even reuse your own code if you took this class before.

Authorship

You are also expected to write this project **yourself**.

Asking anyone someone else to write your code (e.g., a friend, or a "tutor" on a website such as Chegg.com) is not acceptable and will result in severe sanctions.

Sources

You must specify in your report any sources that you have viewed to help you complete this assignment. All of the submissions will be compared with MOSS to determine if students have excessively collaborated, or have used the work of past students.

Violation

Any failure to respect the class rules, both as explained above and in the syllabus, or the UC Davis Code of Conduct will automatically result in the matter being transferred to Student Judicial Affairs.

UC DAVIS
COMPUTER SCIENCE