Individual Coursework 5: Implementing a UNIX Shell

Due date: 5:05 PM GMT, 22nd March 2019

Value: 6% of marks for module

Introduction

The design of the UNIX[1] system call APIs for creating and controlling processes and plumbing together their inputs and outputs is one of the gems of modern software systems architecture. To help you become fluent in using this powerful set of mechanisms in C, in this assignment you will use `fork()`, `exec()`, and several other interesting system calls to build a *command shell,* or shell for short. A shell reads commands on its standard input and executes them. Modern UNIXes include multiple shells (and developers over the years have created still more shells with different syntaxes and features).

In your shell, you will implement simple commands, background commands, conditional commands (i.e., using the `&&` and `||` operators), I/O redirection, and pipes. You will also implement command interruption (i.e., with Ctrl-C). Your shell will implement a subset of the bash shell's syntax, as defined at:

`http://www.gnu.org/software/bash/manual/bashref.html`

and will be generally compatible with bash for the features they share. You may find a tutorial on how to use the UNIX shell useful preparation for CW5; one such tutorial is available at:

`http://www.linuxcommand.org/lc3_learning_the_shell.php`

This coursework must be done in a UNIX programming environment. We recommend that all 0019 students develop in the 0019 Linux virtual machine (VM) provided for use in this class, available at:

`http://www.cs.ucl.ac.uk/staff/B.Karp/0019/s2019/cw.html`

If you happen to have another UNIX machine at your disposal (including a Mac OS machine), you may find that you can develop your solution to CW5 in that environment, but the only development environment the 0019 staff can support is the 0019 VM. We've had difficulties with outdated software on the UCL CS lab machines and CSRW remote desktop, so do not recommend you use them for CW5.

> Chapter 8 of CS:APP/3e covers processes and the UNIX system calls that control them; this is vital background for this coursework. Section 8.4 in particular describes the system calls for creating and controlling processes, and Section 8.5 describes signals, which are important in handling software exceptions that arise while managing the processes your shell starts (e.g., when a process dies, when a user hits Ctrl-C on the shell's console to interrupt a process, etc.). Chapter 10 of CS:APP/3e covers UNIX I/O system calls, such as those needed to handle redirection of input and output for processes your shell starts; Sections 10.1–10.4 describe

[1] Throughout this handout, by "UNIX" we include all modern variants of UNIX, including the open-source Linux variant.

opening, reading from, and writing to files, and Section 10.9 describes redirection of I/O. Note that CS:APP/3e doesn't describe the `pipe()` system call. We will discuss `pipe()` in lecture (and of course the UNIX/Linux man pages for `pipe()` and all other system calls are an invaluable reference).

Tasks

The shell functionality you will implement in CW5 includes:

- execution of simple commands;
- execution of background commands;
- execution of command lists;
- execution of conditional statements;
- execution of command pipelines;
- reaping of zombie processes;
- execution of I/O redirection;
- support for interrupting commands from the console;
- support for the `cd` command (to change directory).

You will complete the above tasks in nine stages described in detail below. We provide tests with the initial code for CW5 for all the above functionality. Details on how grades are computed from tests passed and failed appear later in this handout.

You will also find considerable guidance on how to go about implementing the functionality for each stage of the coursework in this document. *Read this handout in its entirety carefully before you begin!*

As ever, it is important to get started early. You will need time to work your way through implementing all nine stages of the coursework, including time for debugging. You will almost certainly need the entire two weeks allotted to complete CW5.

Getting Started

We will use GitHub for CW5 in much the same manner as for CW2 through CW4. To obtain a copy of the initial code for CW5, please visit the following URL:

`https://classroom.github.com/a/kS9yG1to`

If you'd like a refresher on using git with your own local repository and syncing it to GitHub, please refer to the CW2 handout.

All your code for your solution to CW5 must go in the `sh0019.c` and `sh0019.h` files. The grading server will only consider code in these two files (and will use the baseline versions of all other files in the code we initially provide you).

Each time you push updated code to your GitHub repository for CW5, our automatic grading server will pull a copy of your code, run our automated tests on your code, and place a grade report in a file `report.md` in your GitHub repository. Your mark on CW5 will be that produced by the automated tests run by our automatic grading server on the latest commit (update) you make to your GitHub repository before the CW5 deadline. More on these tests below.

> Please note that your code's behavior on the automated tests when run in the 0019 Linux VM will determine your mark on CW5, without exception.[2] If you choose to develop in a different environment, you must ensure that your code passes the automated tests on our automatic grading server. (We push grade reports to your GitHub repo each time you update your code to make it easy for you to check this.) The CS 0019 staff cannot "support" development environments other than the 0019 Linux VM; we cannot diagnose problems you encounter should your code pass the tests in some other environment, but fail them in the 0019 Linux VM.

Parsing the Shell's Grammar

There are two fairly distinct parts to implementing a shell: parsing the command input and then executing the appropriate commands once they've been parsed. Our emphasis in this class is on the UNIX system call interface and process control, not parsing. So we've provided you much of the code to parse command input (though not quite all).

The `parse_shell_token()` function we provide returns the next "token" from the command line, and it differentiates between normal words like "`echo`" and special control operators like "`;`" or "`>`". But in the code we hand out to you initially, `eval_line()` treats every token like a normal command word, so "`echo foo ; echo bar | wc`" would simply print "`foo ; echo bar | wc`"! Real shells allow users to build up interesting commands from collections of simpler commands, connected by control operators like `&&` and `|`. Part of your task is to complete the parsing phase. You can complete it all at once, but you don't need to; see below for staging hints.

sh0019 command lines follow this grammar. Each command is a "`commandline`" defined as follows:

```
commandline ::= list
              |  list ";"
              |  list "&"

list      ::=  conditional
              |   list ";" conditional
              |   list "&" conditional

conditional ::=  pipeline
              |   conditional "&&" pipeline
              |   conditional "||" pipeline

pipeline ::=  command
              |   pipeline "|" command

command  ::=  [word or redirection]...
```

[2] The only exception is if the 0019 staff determine that your code doesn't implement the functionality required in the CW5 handout.

```
redirection  ::=  redirectionop filename
redirectionop  ::=  "<"  |  ">"  |  "2>"
```

This grammar says, for example, that the command "`echo foo && echo bar & echo baz`" is parsed and executed as follows:

```
{   { echo foo } && { echo bar }   } & { echo baz }
```

That is, the `&&` is "inside" the background command, so "`echo foo && echo bar`" runs in the background and "`echo baz`" runs in the foreground.

A robust shell will detect errors in its input and handle them gracefully, but all the inputs we give your shell in the CW5 tests follow the grammar above.

Executing Commands

The bulk of CW5 is actually implementing the shell—i.e., causing the user's commands to execute after they've been parsed.

If you're confused about a shell feature and tutorials and man pages don't help, experiment! Tinkering with tools to understand their behavior is a hallmark of the Systems approach. The `bash` shell (which is the default on the 0019 VM and on Mac OS) is compatible with your shell. You may find the following commands particularly useful for testing; find out what they do by reading their man pages and feel free to be creative in how you combine them:

- `echo` (print arguments to standard output)
- `true` (exit with status 0)
- `false` (exit with status 1)
- `sleep N` (wait for "N" seconds then exit)
- `sort` (sort the standard input's lines)
- `wc` (count the standard input's lines)
- `cat` (print one or more files specified as arguments to standard output)

Run your shell by typing `./sh0019` and entering commands at the prompt. Exit your shell by typing Ctrl-D at the prompt or by going to another window and entering the command `killall sh0019`.

The Nine Stages of the Shell

We describe below the nine implementation stages you must complete in CW5: what you need to implement in each, and hints on how to do so.

> We suggest you implement the features of your shell in the order of the numbered stages below.

Stage 1: Simple Commands

Implement support for simple commands like "`echo foo`" by changing `start_command()` and `run_list()`. You'll need to use the following system calls: `fork()`, `execvp()`, and `waitpid()`. Consult the man pages for details on how to call each of them and what they do. Also read the function definitions in `sh0019.h`.

You may also need to exit a forked copy of the shell (for example, if `execvp()` fails). To exit a child process, call the `_exit()` function. For instance, call `_exit(1)` or, equivalently, `_exit(EXIT_FAILURE)` to exit with status 1.

> *Your CW5 code should* ***never*** *call the normal* `exit()` *function.* `exit()` performs cleanup actions, including changing the configuration of open stdio files, that shouldn't happen in child processes (they should only happen in the parent shell). If you call `exit()` instead of `_exit()` from child processes, you may see weird behavior where, for example, the parent shell re-executes parts of its command input.

Stage 2: Background Commands

Next, implement support to run processes in the background, such as `sleep 1 &`. A background process allows the shell to continue accepting new commands without waiting for the prior command to complete. Implementing background commands will require changes to `eval_line()` (to detect control operators) and `run_list()`, as well as, most likely, to `struct command`.

Stage 3: Command Lists

Implement support for *command lists,* which are chains of commands linked by `;` and `&`. The semicolon runs two commands in sequence—the first command must complete before the second begins. The ampersand runs two commands in parallel by running the first command in the background. These operators have equal precedence and associate to the left.

This stage will require changes to `run_list()` and `struct command` at a minimum.

> Hint: How much do you need to change `struct command` to handle the full shell grammar above? All that's really required is a simple linked list of `struct command`s. This is possible because the shell's execution strategy for commands works sequentially, from left to right (with an exception for pipelines, as you'll see in a later stage). You may be tempted to create what's called an *expression tree* with separate `struct command`, `struct pipeline`, `struct conditional`, and `struct command_list` types. If that really helps you reason about your design, go for it, but it may be more significantly more effort than it's worth.

Stage 4: Conditionals

Implement support for *conditionals,* which are chains of commands linked by `&&` and/or `||`. These operators run two commands, but the second command is run conditionally, based on the status of the first command. For example:

```
$ true ; echo print      # The second command always runs, because ';' is an
                         # unconditional control operator.
print
$ false ; echo print
```

```
print
$ true && echo print      # With &&, though, the 2nd command runs ONLY if
                          # the first command exits with status 0.
print
$ false && echo print
                          # (prints nothing)
$ true || echo print      # With ||, the 2nd command runs ONLY if the first
                          # command DOES NOT exit with status 0.
                          # (prints nothing)
$ false || echo print
print
```

The `&&` and `||` operators have higher precedence than `;` and `&`, so a command list can contain many conditionals. `&&` and `||` have the same precedence and they associate to the left. The exit status of a conditional is taken from the last command executed in that conditional. For example, `true || false` has status 0 (the exit status of `true`) and `true && false` has exit status 1 (the exit status of `false`).

Explore how conditionals work in the background. For instance, try this command:

```
$ sleep 10 && echo foo & echo bar
```

To support conditionals, you'll probably find you need to make changes to `run_list()`, `eval_line()`, and `struct command`. You'll also use the `WIFEXITED` and `WEXITSTATUS` macros defined in `man waitpid`.

Stage 5: Pipelines

Implement support for *pipelines,* which are chains of commands linked by `|`. The pipe operator `|` runs two commands in parallel, connecting the standard output of the left command to the standard input of the right command.

The `|` operator has higher precedence than `&&` and `||`, so a conditional can contain several pipelines. Unlike conditionals and lists, the commands in the pipeline run *in parallel.* The shell starts all the pipeline's commands, but only waits for the *last* command in the pipeline to finish. The exit status of the pipeline is taken from that last command.

To support pipelines, you'll need to use some further system calls, namely `pipe()`, `dup2()`, and `close()`, and you'll need to make changes to `start_command()`, `run_list()`, and `struct command`.

Stage 6: Reaping Zombie Processes

Your shell should eventually reap all its zombie processes using `waitpid()`.

> Hint: You must reap all zombies *eventually,* but you need not to reap them *immediately.* We don't recommend using signal handlers to reap zombies, since a signal handler can interfere with the `waitpid()` calls used to wait for foreground processes to complete. A well-placed `waitpid()` loop will suffice to reap zombies. Where should it go?

Stage 7: I/O Redirection

Implement support for *I/O redirection,* where some of a command's file descriptors are read from (for input file descriptors) and/or written to (for output file descriptors) disk files. You must handle three kinds of redirection:

- `< filename`: The command's standard input is taken from `filename`.
- `> filename`: The command's standard output is sent to `filename`.
- `2> filename`: The command's standard error is sent to `filename`.

For instance, `echo foo > x` writes `foo` into the file named `x`.

The `parse_shell_token()` function returns redirection operators as type `TOKEN_REDIRECTION`. You'll need to change `eval_line()` to detect redirections and store them in `struct command`. Each redirection operator must be followed by a filename (a `TOKEN_NORMAL` token). You'll also change `run_command()` to set up the redirections, using system calls `open()`, `dup2()`, and `close()`.

The shell sets up a command's redirections before executing the command. If a redirection fails (because the file can't be opened), the shell doesn't actually run the command. Instead, the child process that would normally have run the command prints an error message to standard error and exits with status 1. Your shell should behave this way, too. For example:

```
$ echo > /tmp/directorydoesnotexist/foo
/tmp/directorydoesnotexist/foo: No such file or directory
$ echo > /tmp/directorydoesnotexist/foo && echo print
/tmp/directorydoesnotexist/foo: No such file or directory
$ echo > /tmp/directorydoesnotexist/foo || echo print
/tmp/directorydoesnotexist/foo: No such file or directory
print
```

How to figure out the right error message to display on standard error? Try `man strerror`.

> Hint: Your calls to `open()` will have different arguments depending on what type of redirection is used. How to figure out what those arguments should be? You can use the man page or you can simply use the `strace` command to check the regular shell's behavior. For example, try this:
>
> ```
> $ strace -o strace.txt -f sh -c "echo foo > output.txt"
> ```
>
> The strace output is placed in file `strace.txt`. Examine that file's contents. Which flags were provided to `open()` for `output.txt`? You can repeat this experiment with different redirection types.

Stage 8: Interruption

Implement support for *interruption:* pressing Ctrl-C in the shell should kill the currently running command line, if there is one.

Handling Ctrl-C is an initial step into *job control,* which encompasses UNIX's functionality to help users interact with sets of related processes. Job control can be a complicated affair involving process groups, controlling terminals, and signals. Luckily, Ctrl-C is not too hard to handle on its own. You will need to take the following steps:

- All processes in each pipeline must have the same *process group* (see below).
- Your shell should use the `claim_foreground()` function that we provide to inform the OS about the currently active foreground pipeline.

- If the user presses Ctrl-C while the shell is executing a foreground pipeline, every process in that pipeline must receive the `SIGINT` signal. This will kill them.

What are process groups? Job control is designed to create a common-sense mapping between operating system processes and command-line commands. This gets interesting because processes spawn new helper processes. If a user kills a command with Ctrl-C, the helper processes should also die. UNIX's solution uses *process groups,* where a process group is a set of processes. The Ctrl-C key kills all members of the current foreground *process group,* and not just the current foreground process.

Each process is a member of exactly one process group. This group is initially inherited from the process's parent, but the `setpgid()` system call can change it:

- `setpgid(pid, pgid)` sets process `pid`'s process group to `pgid`. Process groups use the same ID space as process IDs, so you'll often see code like `setpgid(pid, pid)`.
- `setpgid(0, 0)` means the same thing as `setpgid(getpid(), getpid())`. This divorces the current process from its old process group and puts it into the process group named for itself.

To kill all processes in group `pgid`, use the system call `kill(-pgid, signal_number)`.

(Note that one process can change another process's process group. Process isolation restricts this functionality somewhat, but it's safe for the shell to change its children's process groups.)

For interrupt handling, each process in a foreground command pipeline must be part of the same process group. This will require that you call `setpgid()` in `start_command()`. In fact, you should call it *twice,* at two different locations in `start_command`, to avoid race conditions (why?).

Once the above has been done, your shell should call `claim_foreground()` before waiting for a command. This function makes the terminal dispatch Ctrl-C to the process group you choose. Call `claim_foreground(pgid)` before waiting for the foreground pipeline, and call `claim_foreground(0)` once the foreground pipeline is complete. This function manipulates the terminal so that commands like `man kill` will work inside your shell.

When a user types Ctrl-C into a terminal, the UNIX system *automatically* sends the `SIGINT` signal to all members of that terminal's foreground process group. This will cause any currently executing commands to exit. (Their `waitpid()` statuses will have `WIFSIGNALED(status) != 0` and `WTERMSIG(status) == SIGINT`.)

Finally, if *the shell itself* gets a `SIGINT` signal, it should cancel the current command line and print a new prompt. Implementing this feature will require adding a signal handler.

> Hint: We *strongly* recommend that signal handlers do *almost nothing*. A signal handler might be invoked at any moment, including in the middle of a function or library call; memory might be in an arbitrary intermediate state. Since these states are dangerous, UNIX restricts signal handler actions. Most standard library calls are disallowed, including `printf()`. (A signal handler that calls `printf()` might be observed to work most of the time—but one time in a million the handler would exhibit unpredictably incorrect behavior.) The complete list of library calls allowed in signal handlers can be found in `man 7 signal`. For this coursework, you can accomplish everything you need with a one-line signal handler that writes a global variable of type `volatile sig_atomic_t` (which is a synonym for `volatile int` on today's Linux on x86-64).

Note that Ctrl-C and command lines interact in different ways on different operating systems. For instance, try typing Ctrl-C while the shell is executing `sleep 10 ; echo Sleep failed`

(or `sleep 10 || echo Sleep failed`). The result may vary with the OS: Mac OS prints Sleep failed, but Linux does not! Your shell implementation for 0019 CW5 may exhibit either behavior. But note that if you press Ctrl-C during `sleep 10 && echo Sleep succeeded`, the message does not print on *any* OS, and you must not print the message either.

Stage 9: the `cd` Command

Implement support for the `cd` *directory* command. The `cd` command is different than other commands your shell executes; why?

Running the Tests and Submitting

The automated tests for CW5 are very much intended to help you debug, and to guide your implementation (in the way the ones in CW2 did): you can see which specific inputs are causing your shell to behave incorrectly, and change your code accordingly.

There are 83 automated tests in total, divided into groups as follows:

- `SIMPLE`: simple command execution
- `BG`: background command execution
- `LIST`: command list execution
- `COND`: conditional command execution
- `PIPE`: command execution with pipes
- `ZOMBIE`: reaping of zombie processes
- `REDIR`: I/O redirection execution
- `INT`: SIGINT (Ctrl-C) processing
- `CD`: `cd` command execution
- `ADVPIPE`: advanced pipe test case
- `ADVBGCOND`: advanced tests of background conditionals

Use the command `make check` to run the automated tests in your development environment. You may also run `make check-`*X* to run specific test *X* (e.g., where *X* could be `simple2`), or (for example) `make check-simple` to run all the `SIMPLE` tests. (You may find it useful to design your own tests as you debug, as well.)

Each individual test is of equal weight in determining your mark: your mark will be $P/83 \times 100$, where P is the number of tests your code passes.

Once again, we urge you to get started early.

Submitting via GitHub

We will deem the timestamp of your CW5 submission to be the timestamp on GitHub's server of the last commit you make before the submission deadline of 5:05 PM GMT on 22nd March 2019. Your mark will be the mark you receive on the automated tests for that version of your code. Coursework lateness penalties are described in detail on the 0019 class web site.

If you wish to submit after the deadline, *you must take the following steps for your coursework to be marked:*

1. When you wish to receive a mark for a version of your code that you push to GitHub after the submission deadline, you must begin your commit log message for that commit with the exact string `LATESUBMIT`. *Our grading system will not mark your late submission unless you comply with this requirement.* We follow this policy so that if a student "accidentally" pushes a further commit after the deadline, they aren't penalized for a late submission.

2. If you wish to claim late days, per the 0019 late days policy (details on the 0019 class web site), you must post a private message on Piazza to the Instructors *no later than one hour after pushing the version of your code you wish to submit late to GitHub* with a subject line `CW5 LATE DAYS` and a body stating the number of late days you would like to claim. *Please follow these instructions exactly: note the all capitals, and please use this exact subject line string.*

3. You may make only one late submission (i.e., one GitHub commit with the initial string `LATESUBMIT` and one Piazza posting with the subject line `CW5 LATE DAYS`). If you make more than one late submission, we will only mark the first one.

Academic Honesty

This coursework is an *individual coursework*. Every line of code you submit must have been written by you alone, and must not be a reproduction of the work of others—whether from the work of students in this (or any other) class from this year or prior years, from the Internet, or elsewhere (where "elsewhere" includes code written by anyone anywhere).

Students are permitted to discuss with one another the definition of a problem posed in the coursework and the general outline of an approach to a solution, but not the details of or code for a solution. Students are strictly prohibited from showing their solutions to any problem (in code or prose) to a student from this year or in future years. In accordance with academic practice, students must cite all sources used; thus, if you discuss a problem with another student, you must state in your solution that you did so, and what the discussion entailed.

Any use of *any* online question-and-answer forum (other than the CS 0019 Piazza web site) to obtain assistance on this coursework is strictly prohibited, constitutes academic dishonesty, and will be dealt with in the same way as copying of code. The same goes for use of any online material specifically directed toward solving this coursework.

You are free to read other reference materials found on the Internet (and any other reference materials), ***apart from*** *source code that implements a UNIX or Linux shell.* You may of course use the code we have given you. *Again, all other code you submit must be written by you alone.*

Copying of code from student to student or from the Internet is a serious infraction; it typically results in the awarding of zero marks to all students involved, and is viewed by the UCL administration as cheating under the regulations concerning plagiarism. Penalties imposed can extend all the way to exclusion from all further examinations at UCL. The course staff use extremely accurate plagiarism detection software to compare code submitted by all students (as well as code found on the Internet) and identify instances of copying of code; this software sees through attempted obfuscations such as renaming of variables and reformatting, and compares the actual parse trees of the code. Rest assured that it is far more work to modify someone else's code to evade the plagiarism detector than to write code for the assignment yourself!

Read the Piazza Web Site

You will find it useful to monitor the 0019 Piazza web site during the period between now and the due date for the coursework. Any announcements (*e.g.*, helpful tips on how to work around unexpected problems encountered by others) will be posted there. And you may ask questions

there. *Please remember that if you wish to ask a question that reveals the design of your solution, you must mark your post on Piazza as private, so that only the instructors may see it.* Questions about the interpretation of the coursework text, or general questions about C that do not relate to your solution, however, may be asked publicly—and we encourage you to do so, so that the whole class benefits from the discussion.

References

CS:APP/3e Chapters 8 and 10, particularly §8.4-8.5, 10.1–10.4, and 10.9

Acknowledgement

This coursework is derived from one created by Eddie Kohler.