Assignment 1: Shell

For this assignment you will implement a Unix shell similar to `sh`, `bash` and `zsh`. A shell is an old, but still widely used, way of running programs on your system. Most shells, including the one you will implement, support features that allows for additional operations with these commands, such as feeding the output of one command to the input of another. Note that, except for explicitly mentioned built-ins like `cd`, a shell does **not** implement any of the commands it executes itself, it simply launches other binaries from your PATH.

For this assignment you can choose which exact features to implement in your shell. Not all features are required to obtain a 10 as final grade. A list of all possible features is listed below. Your shell may not be based on (or exploit) another shell. For example, `system(3)` is forbidden. You must submit your work as a tarball on Canvas. You can use `make tarball` to generate a tarball for submission automatically. This assignment is individual; you are not allowed to work in teams.

Getting started

1. Set up your environment; refer to the setup document for detailed instructions. In short, on Linux you should install `build-essential python3 python3-pexpect libreadline-dev flex`, on Windows use WSL2 (and then follow Linux instructions), and on macOS you will need to use Docker.
2. Unpack the provided source code archive; then run `make`.
3. Try the generated `mysh` binary and familiarize yourself with its interface. Try entering the following commands at its prompt:

   ```
   echo hello
   sleep 1; echo world
   ls | more
   ```

   The framework already provides a front-end for your shell that reads and parses user input into 'nodes'. By default `shell.c` calls `print_tree` to dump this data structure. To understand the default output, read the file `parser/ast.h`.
4. Read the manual pages for `exit(3)`, `chdir(2)`, `fork(2)`, `execvp(3)`, `waitpid(2)` (and the other `wait` functions), `pipe(2)`, `dup2(2)`, `sigaction(2)`. You will need (most of) these functions to implement your shell.

   Note that when you see something like `foo(n)`, this refers to page `foo` in section `n` of the man-pages. To read these, run the following command: `man n foo` (or use google). The section number is optional, and the `man` command will usually default to section 1 (shell commands), 2 (system calls) or 3 (library calls).
5. Try and modify the file `shell.c` and see the effects for yourself. This is where you should start implementing your shell. You should not have to modify the front-end (`front.c`) or the parser (`parser/` directory).

Features and grading

Your grade will be 0 if you did not submit your work on time, in an invalid format, or has errors during compilation.

If your submission is valid (on time, in valid format, and compiles), your grade starts from 0, and the following tests determine your grade (in no particular order):

- +1pt if you have submitted an archive in the right format with source code that builds without errors.
- +1pt if your shell runs simple commands properly.

- +1pt if your shell supports the `exit` built-in command.
- +1pt if your shell supports the `cd` built-in command.
- +1.5pt if your shell runs sequences of 3 commands or more properly.
- +1.5pt if your shell runs pipes of 2 simple commands properly.
- -1pt if `gcc -Wall -Wextra` reports warnings when compiling your code.
- -1pt if your shell fails to print an error message on the standard error when an API function fails.
- -1pt if your shell stops when the user enters Ctrl+C on the terminal, or if regular commands are not interrupted when the user enters Ctrl+C.

The following extra features will be tested to obtain higher grades, but only if you have obtained a minimum of 5 points on the list above already:

- +1pt if your shell runs pipes of more than 2 parts consisting of sequences or pipes of simple commands.
- +1pt if your shell supports redirections.
- +0.5pt if your shell supports detached commands.
- +0.5pt if your shell supports executing commands in a sub-shell.
- +0.5pt if your shell supports the `set` and `unset` built-ins for managing environment variables.
- +0.5pt if you parse the PS1 correct and support at least the host name, user name and current path.
- +2pt if your shell supports simple job control: Ctrl+Z to suspend a command group, `bg` to continue a job in the background, and `fg` to recall a job to the foreground.
- -1pt if your source files are not neatly indented or formatted.

The grade will be maximized at 10, so you do not need to implement all features to get a top grade.

> ***Note***
>
> Your shell will be evaluated largely automatically. This means features only get a positive grade if they work perfectly, and there will be no half grade for "effort". Most features listed above are separated into several sub-features which are worth points individually.

Compile and run

Run *make* to compile the sources. By default the provided Makefile assumes your solution is entirely contained in *shell.c*. Did you create additional sources or headers? Then you can modify the Makefile; add your files to *ADDITIONAL_SOURCES* (Makefile:2) or *ADDITIONAL_HEADERS* (Makefile:5). Now *make* will compile your files as well.

We provide a Python script to test the solution (i.e., *check.py*). You can use the Python script to run a specific test group.

For example:

```
python3 check.py compile
```

or

```
python3 check.py simple
```

Otherwise use it with Makefile (*make check* to run all tests). The script runs specific (random) commands and analyzes the output on the shell to infer if your implementation is correct. You are allowed to inspect the Python script. However, do not hardcode the output to pass the test. We manually inspect the code and we check for attempts to bypass our tests.

Linux is the only supported OS. Your solution may or may not work on other OSes. We advice Docker if you want to test locally on a virtualized environment.

Run *make docker-update* to obtain the official docker image. You can now run *make docker-check* to run all the tests inside docker. Please refer to the setup guide to install Docker and get into the container. (read also *Tips in developing using docker*)

Example commands

```
## simple commands:
ls
sleep 5   # must not show the prompt too early
```

```
## simple commands, with built-ins:
mkdir t
cd t
/bin/pwd  # must show the new path
exit 42   # terminate with code
```

```
## sequences:
echo hello; echo world # must print in this order
exit 0; echo fail  # must not print "fail"
```

```
## pipes:
ls | grep t
ls | more    # must not show prompt too early
ls | sleep 5 # must not print anything, then wait
sleep 5 | ls # must show listing then wait
ls /usr/lib | grep net | cut -d. -f1 | sort -u
```

```
## redirects:
>dl1 ls /bin; <dl1 wc -l
>dl2 ls /usr/bin; >>dl1 cat dl2 # append
<dl2 wc -l; <dl1 wc -l # show the sum
>dl3 2>&1 find /var/. # errors redirected
```

```
## detached commands:
sleep 5 &  # print prompt early
{ sleep 1; echo hello }& echo world; sleep 3 # invert output
```

```
## sub-shell:
( exit 0 ) # top shell does *not* terminate
cd /tmp; /bin/pwd; ( cd /bin ); /bin/pwd # "/tmp" twice
```

```
## environment variables
set hello=world; env | grep hello # prints "hello=world"
```

```
(set top=down); env | grep top # does not print "top=down"

# custom PATH handling
mkdir /tmp/hai; touch /tmp/hai/waa; chmod +x /tmp/hai/waa
set PATH=/tmp/hai; waa # OK
unset PATH; waa # execvp() reports failure
```

Syntax of built-ins

Built-in: `cd <path>`

Change the current directory to become the directory specify in the argument. Your shell does not need to support the syntax `cd` without arguments like Bash does.

Built-in: `exit <code>`

Terminate the current shell process using the specified numeric code. Your shell does not need to support the syntax `exit` without arguments like Bash does.

Built-in (advanced): `set <var>=<value>`

Set the specified environment variable. Your shell does not need to support the syntax `set` without arguments like Bash does.

Built-in (advanced): `unset <var>` **(optional)**

Unset the specified environment variable.

Error handling

Your shell might encounter two types of error:

- When an API function called by the shell fails, for example `execvp(2)` fails to find an executable program. For these errors, your shell must print a useful error message on its standard error (otherwise you can lose 1pt on your grade). You may/should use the helper function `perror(3)` for this purpose.
- When a command launched by the shell exits with a non-zero status code, or a built-in command encounters an error. For these errors, your shell *may* print a useful indicative message, but this will not be tested.

In any case, your program should not leak resources like leaving file descriptors open or forgetting to wait on child processes.

Prompting

The text a shell shows before user input is called the prompt, which is normally determined by the `PS1` environment variable. This variable can contain special characters that will be replaced, for example, by the current username or the current working directory. For more information see PROMPTING in the `bash(1)` manual page.

For your shell, supporting `PS1` is not required, but doing so correctly is worth additional points. If you do, it should support the current user (`\u`), current hostname (`\h`) and current working directory (`\w`).

If no `PS1` value is set (or this feature is not implemented), you should display a default prompt that should always end with a `$` character. The `PS1` values set by the tester will always end with a `$`.

You can use `getuid` and `gethostname` to get the current user and hostname.

Some tips about the shell

1. It is not necessary that your shell implements advanced features using '*', '?', or '~'.
2. If you do not know how to start, it is best to first start with simple commands, i.e., the 'command' node type.

```
if (node->type == NODE_COMMAND) {
  char *program = node->command.program;
  char **argv = node->command.argv;
  /* Here comes a good combination of fork and exec */
  ...
}
```

3. A shell usually supports redirections on all places of a simple command; `ls > foo` and `>foo ls` are normally equivalent. However, this shell only supports `>foo ls`.
4. Within a 'pipe' construction, all parts should be forked, even if they only contain built-in commands. This keeps the implementation easier.

```
exit 42 # closes the shell
exit 42 | sleep 1  # exit in sub-shell, main shell remains

cd /tmp # changes the directory
cd /tmp | sleep 1  # change directory in sub-shell
                   # main shell does not
```

Tips in developing using docker

If you are developing using docker, and want to test out your shell with your own tests (and not the automated ones) you can use the following command within your working folder:

```
docker run -it --rm --read-only -v "`pwd`:/code" -w /code \
vusec/vu-os-shell-check /bin/bash
```

After running this command, you can use the linux commands, such as `make` to build your mysh binary and run it manually, or alternatively `make check` to run the automated tests.

Some tips about the environment

- Use `make check` (or `make docker-check`) to test your features locally.
- You should also submit your work on canvas for evaluation (via Codegrade). Do so often to make sure your code behaves as expected on our setup, as there may be differences between your local environment and ours. You can submit your work for evaluation as often as you like. This environment is the same as the one used for final grading. **Make sure your submission works on our environment, you will not receive points if it does not.** The last submission you make before the deadline will be used for final grading.
- If you are experiencing issues due to differences between your local environment and that on the grading server, you can use Docker to replicate the environment locally. For instructions, refer to the setup document.
- Please report any bugs you may encounter in the automated checking script, such as the awarded points being too high or low. It is strictly prohibited to attempt to cheat the script or attack the infrastructure.

- You are free to choose a C coding style for your code, as long as you are consistent. An example coding style is that of the Linux kernel [1].
- You may add additional source files to organize your code. Add these files to `ADDITIONAL_SOURCES` or `ADDITIONAL_HEADERS` so the environment will correctly use these.

---

1 https://www.kernel.org/doc/html/v4.10/process/coding-style.html