Operating Systems Lab 3: Shell

Shell - Input and Output

For the second iteration of the shell assignment, you will work on implementing input and output features for your shell: input/output redirection for commands and pipelines, to be specific. For example, you will be able to run `./a.out | ./b.out < in > out`, which will send the contents of file `in` to `stdin` of `./a.out`, after which its `stdout` will be sent to `stdin` of `./b.out`, and in turn its `stdout` will be stored in file `out`.

As you can see in the grammar (below), our shell contains the notion of a 'pipeline', where we combine multiple (can be more than 2) commands using the `|` operator which will connect the `stdout` and `stdin` of these commands in a 'pipeline' fashion. In contrast to normal bash, only the first input and final output can be redirected to files. So our command `./a.out | ./b.out < in > out` would be equivalent to the bash command `./a.out < in | ./b.out > out`.

For a pipeline, you should start all commands simultaneously - you should not wait for the first to exit before starting the second. Each command should be linked to the next through a Linux native pipe (`pipe()`), not by storing the output in a buffer manually. The exit code of a pipeline is the exit code of the last command in the pipeline, and a pipeline is only finished when all its commands are finished. If no input or output redirection is specified for a pipeline, the respective input/output should be connected to the terminal, just like for a single command in the first iteration.

An additional feature we ask you to implement is the built-in command `cd` that will take the name of a directory and switch to it. When no name is given, you should print `Error: cd requires folder to navigate to!`; when the given folder is invalid, print `Error: cd directory not found!`. Of course, `cd` should have appropriate exit codes: `0` when the directory was changed successfully, and `2` when there was an error.[1]

In your implementation, you will probably need the functions `open`, `close`, `dup`, `dup2` (or `dup3`), and `pipe`. When writing to a file, you should truncate the existing file (as is default behaviour in `bash`, so don't append existing content). Your implementation should also make sure that the input and output file are not the same, otherwise print the error `Error: input and output files cannot be equal!` (you don't have to consider edge cases where the two file names are different but are actually linked to the same file). Of course, not providing a filename after entering `>` or `<` is considered a syntax error.

The same submission conditions and instructions apply as for the first iteration. Again, your implementation will be checked by Themis, including error handling and memory management tests.

Hints

- Pay special attention to properly `wait()` (or `waitpid()`) for each child process to finish, especially in pipelines, to ensure proper command execution order and to prevent creating orphan or zombie processes. Themis will check for this!

[1] Exit code `2` specifically denotes errors with built-in functions, rather than other binaries: `https://tldp.org/LDP/abs/html/exitcodes.html`

- Waiting for multiple processes to complete can either be done by calling `waitpid` for each child process ID you obtain, or by calling `waitpid(-1, ...)` or `wait` n times, where n is the number of processes you spawned. We recommend you start keeping track of child process IDs, as you will need those in the last iteration.
- Built-in commands other than `status` should have an exit code as well: `0` upon success and `2` upon any failure. As mentioned before, syntax errors should not affect the most recent exit code.
- Themis will test not only the features from this iteration, but also those of all previous iterations! Therefore, make sure these continue to work so you are *adding* rather than *replacing* functionality.
- You should **not** use `setpgid()` or similar methods, as this might break Themis when you do not terminate child processes correctly! Such an issue will then cause processes to be orphaned, which will keep Themis waiting indefinitely.
- Make sure to disable input and output buffering using `setbuf(stdin, NULL);` and `setbuf(stdout, NULL);`, to prevent out-of-order prints in the Themis output (and consequently failing testcases).
- If you use flex, make sure to set the option `%option always-interactive`.
- Themis will only check `stdout`, so make sure to print any errors there and **NOT** on `stderr`!
- Themis will check for memory leaks using `valgrind` in all testcases! Memory leaks will be indicated by a 'Runtime error' with exit code **`111`**. Don't worry about memory leaks between a `fork` and `exec`: `valgrind` won't report those as the entire process will be replaced by the `exec` call. Also exiting your child process after a failed `exec` won't be reported to Themis.

Grammar

The full grammar describing the syntax to accept in your shell is defined below. It already includes features to be implemented in future labs, for completeness. In general, your intuition and experience with shells will be sufficient to know what to allow/forbid in input and note that we will not test 'exotic' inputs in Themis - the objective of this lab is not to build a perfect parser, but a usable shell.

In the grammar, an 'executable' can be the path to a file, or can refer to a file in the user's `$PATH`, while 'builtin' refers to a built-in command. The 'options' part represents any (possibly empty!) set of parameters/strings that should be put into `argv` of the program or command.

From the grammar, it should be clear that a built-in command cannot join any of the I/O redirection and piping that you are asked to implement in this lab. This might simplify your implementation. Additionally, note that any character appearing in the grammar is a 'reserved character' that can only otherwise occur in strings (`""`). Strings are only parsed using double quotes, single quotes can be treated as normal characters.

⟨*command*⟩ ::= ⟨*executable*⟩ ⟨*options*⟩

⟨*pipeline*⟩ ::= ⟨*command*⟩ `|` ⟨*pipeline*⟩
| ⟨*command*⟩

⟨*redirections*⟩ ::= `<` ⟨*filename*⟩ `>` ⟨*filename*⟩
| `>` ⟨*filename*⟩ `<` ⟨*filename*⟩
| `>` ⟨*filename*⟩
| `<` ⟨*filename*⟩
| ⟨*empty*⟩

⟨*chain*⟩ ::= ⟨*pipeline*⟩ ⟨*redirections*⟩
| ⟨*builtin*⟩ ⟨*options*⟩

⟨*inputline*⟩ ::= ⟨*chain*⟩ `&` ⟨*inputline*⟩
| ⟨*chain*⟩ `&&` ⟨*inputline*⟩
| ⟨*chain*⟩ `||` ⟨*inputline*⟩
| ⟨*chain*⟩ `;` ⟨*inputline*⟩
| ⟨*chain*⟩
| ⟨*empty*⟩

Bonuses

Again, there is the possibility to implement some bonuses to obtain up to 2 bonus points (yielding a maximal grade of 12). For the second iteration, these are some of our suggestions:

- Allow a 'script' (i.e. sequence of commands, pipelines, redirections, etc.) to be entered in your shell all at once by providing a script filename as the first argument: `./shell scriptfile`.
- Add the possibility to redirect `stderr` in addition to `stdout` by implementing the `n>` operation that pipes file stream `n` to the given file.
- Allow a pipeline output stream to be redirected to multiple files, or to have multiple input files for a pipeline.
- Implement a simple command history (when pressing up/down arrows).
- Implement a directory stack, so implementing the commands `pushd` and `popd` to easily navigate 'back'.

Note that despite some of these options being the same as in previous labs, you cannot get bonus points for a bonus more than once. Of course, you can get as creative as you want and do more, we will award points proportional to the features you implemented with a cap of 2 points. If you decide to implement some of these bonuses, make sure to make them togglable using a compilation flag (`#if EXT_PROMPT`, compile with `-DEXT_PROMPT`) and submit to Themis in the separate 'bonuses' entry. Make sure to include a `README` documenting your work - we can only award points for bonuses discussed there.