Gabriel Caiaffa - 11838669

Isabella Basso do Amaral - 11810773

Mini Shell

This is a minimal shell implementation for the Minix system.

> Note: it should also work on other Unix systems, but has only been tested on Minix and Linux.

Structure

It consists of a simple a parser on `minhaMiniShell.c` that can call binaries.

Four commands are made available, namely:

- `nem_eu_nem_de_ninguem FILE`

  This command will change permissions of the specified FILE to 0000.

- `soh_eumesmo FILE`

  This command will change permissions of the specified FILE to 0700.

- `rodaeolhe PATH_TO_CMD [ARGS ...]`

  This command will run the specified PATH_TO_CMD with [ARGS …] on the foreground, then print:

  `=> programa 'PATH_TO_CMD' retornou com código RETVAL`

  where `RETVAL` is the return code of the program.

- `sohroda PATH_TO_CMD [ARGS ...]`

  This command will run the specified PATH_TO_CMD with [ARGS …] on the background.

Each command has its own binary.

Building and installing

To build simply run `make`, and to install run `make install`. Binaries will be installed to `/usr/local/bin`.

Implementation

Error checking

Error checking is performed with `errno` as needed, using `exit` to terminate the process in case of unrecoverable errors or signals – where due, i.e. `rodaeolhe` was specifically asked to return the child's exit status thus it does not comply to this.

Shell

The `minhaMiniShell` binary acts as a simple parser, with an initialization step and its main loop. As every variable is located on the stack there's no need for an after step.

Initialization The shell first creates local variables for holding user input, then we assign a simple `(*handle_sig)(int)` function as a signal handler, that should just exit the parser once it gets either `SIGINT` or `SIGQUIT`. Signal handlers are dealt with using the `signal` syscall.

Parser loop We also begin by defining local variables to hold command specific state (e.g. arguments and their count).

Then we use the `write` syscall to insert the prompt on the standard output (file description 1) and use the `read` syscall to get user input from the standard input (file descriptor 0). As the input is not formatted, we use the `string` library's `strtok_r` function to parse the command and its arguments.

> Note: This shell implementation fails to hold string arguments together, and can only handle simple space delimiters.

Finally, we `fork` the process: - The parent uses `waitpid` in order to wait for its children to finish. - The child uses `execvp` in order to execute the command provided by the user.

Commands

File permissions management Both `nem_eu_nem_de_ninguem` and `soh_eumesmo` will use the `chmod` syscall in order to change permissions. Before calling `chmod` both commands will use the `stat` syscall in order to check whether the file exists.

Process execution Both `rodaeolhe` and `sohroda` use the `fork` and `execve` syscalls in order to execute the commands passed in. As we use arguments on both commands, we can simply get them using the `char *argv[]` then `execve` directly.

On `sohroda` we must first explicit that we wish to ignore `SIGINT` and `SIGQUIT` signals, as we wish for the program to run on the background, thus it must not intercept those signals. As in the shell, we use the `signal` syscall for that, specifying `SIG_IGN` as the handler.

On `rodaeolhe` we don't bother handling signals as the shell already does that for us, but in order to print the exit message:

```
=> programa 'PATH_TO_CMD' retornou com código RETVAL
```

we must use the `waitpid` syscall on the parent process in order to wait for the children to execute. After we wait, we can simply `write` the output piece by piece.

Before we can print the child's exit code we must convert it to a string, thus we've implemented a simplified `itoa` (which we call `_utoa`) that converts an unsigned integer to a string. We know that `waitpid` takes in an `int *statloc` that holds both the child's exit status and its termination status, thus, we use the `WEXITSTATUS` macro to get the exit status.

The `_utoa` function works by, first, successively getting the remainder of the number divided by the base (10) and shifting this value into its ASCII equivalent (by summing it to '0'). The loop ensures we get the next digit every iteration by also dividing the number by the base. Finally, we must reverse the number as we read it backwards.

We print the exit message to the standard output for `rodaeolhe` using the `write` syscall again.