Advanced Programming: Yet Another Shell (ysh)

Emilio E. G. Cantón Bermúdez

28/11/2019

Manual for YSH

Git

Many of us programmers do not wonder how many of the peculiarities within a shell that we use day to day pass without us thinking how it is implemented. The main problem to be solved with this project is to confront this gap between my knowledge of how to use a shell and how is implemented. The problem may be broken into the following pieces:

- Input parsing
- Running program
- Implement pipelines
- IO Redirection
- Environment variables

Context

The shell uses the following precompiler/global variables in order to work:

```
#define DELIM " \t\r\n"
#define PIPE_DELIM '|'
#define MAX_SIZE 4096
#define MAX_LENGTH 100
#define BUILTINS_LENGTH 4
#define HELP_STRING
"\
---------------------------------------------------\n\
Yet Another Shell by Emilio E. G. Cantón Bermúdez\n\
---------------------------------------------------\n\
Thanks for using YSH! As of today the following are\n\
implemented:\n\
-   Running commands\n\
-   Signal Handling\n\
-   Using environment variables\n\
-   Piping\n\n\
A few extra commands you should know:\n\
-   help: Show this message\n\
-   exit: Exit the shell\n\
"

typedef struct {
    int n_commands;
    char *commands[MAX_LENGTH][MAX_LENGTH];
    int pipes[MAX_LENGTH - 1][2];
} pipecmd;
```

```
int pid;
char *builtins[BUILTINS_LENGTH] = {"cd", "chdir", "exit", "help"};
```

Input parsing

The shell is able to process any general command on `PATH` with its arguments. In the following lines the functions are explained:

```
char *readline();
```

This functions reads a line from the input and saves it on a `char` pointer.

```
void tokenize(char *line);`
```

The main purpose of the function is to divide the command into *tokens* which are separated by DELIM. If a PIPE_DELIM is detected the function detects each separated command and adds it to the `[pipecmd](#Context) struct` along with the pipes descriptors of between each command. The `pipes` matrix stores each of these pipes, in which the specified input and output file descriptor is saved for the $i-th$ on the index 0 and 1 respectively. This means that each pipe contains one file descriptor for the previous command (read) and one for the next one (write). Therefore, the first and last command will always have one of the descriptors unset.

Redirection is implemented in here also. The function `strchr`is applied in order to find the chars `<` and `>`. The correct way to make a redirection is the following:

Input

`[0-2]<[FILENAME]` is expected in which the digit represents a file descriptor owned by the process and the filename the source of input.

Output

`[0-2]>[FILENAME]` is expected in which the digit represents a file descriptor owned by the process and the filename the destination for output.

Note: For the time being it is not possible to redirect both input and output on the same command

```
int execute(char **args, int redirects[2][2]);
```

The function takes a *simple* (one) command, forks a new process and executes the command. In between redirections are verifiec in order to apply them by using `dup2` and duplicating the correct file descriptors (if specified) into `STDIN_FILENO` and`STDOUT_FILENO` on the process file descriptor table (`/proc/[process id]/fd`). The execution is done using `execvp` which takes the filename of the command and the args as an NULL-ended array.

```
int execute_pipes(pipecmd *cmds);`
```

This function takes the `pipecmd struct` created by `[tokenize](#Input-parsing)` and executes each of the commands with its arguments en `pipecmd->commands`. The process forks in a `for` loop and duplicates the necessary descriptors on the corresponding pipe and closes the unused ones. Finally it executes the command, while the parent process waits for the process to finish. Before waiting, the parent closes the corresponding file descriptors of the current command. A part from letting the current commands related to this descriptors execute in a common way (if the descriptors are open the command keeps waiting for the descriptors to close) this ensures that none of the subsequent commands will copy these descriptors into their memory.

```
void start();
```

This is the main function which does the following:

- Get environment variables for user's context
  - User
  - Host
  - Current Working Directory (CWD)

Afterwards it takes the current process id in order to avoid breaking the main shell process when no command/child is running, prompts the user with the context, waits for a line to read and processes the line with `tokenize`

```
void handler(int signal);
```

This function handles the `SIGINT` signal in order to interrupt running programs.