Minishell

Objectives

- Learn how shells create new child processes and connect the I/O to the terminal.
- Gain a better understanding of the `fork()` function wrapper.
- Learn to correctly execute commands written by the user and treat errors.

Statement

Introduction

A shell is a command-line interpreter that provides a text-based user interface for operating systems. Bash is both an interactive command language and a scripting language. It is used to interact with the file system, applications, operating system and more.

For this assignment you will build a Bash-like shell with minimal functionalities like traversing the file system, running applications, redirecting their output or piping the output from one application into the input of another. The details of the functionalities that must be implemented will be further explained.

Shell Functionalities

Changing the Current Directory

The shell will support a built-in command for navigating the file system, called `cd`. To implement this feature you will need to store the current directory path because the user can provide either relative or absolute paths as arguments to the `cd` command.

The built-in `pwd` command will show the current directory path.

Check the following examples below to understand these functionalities.

```
> pwd
/home/student
> cd operating-systems/assignments/minishell
> pwd
/home/student/operating-systems/assignments/minishell
> cd inexistent
no such file or directory
> cd /usr/lib
> pwd
/usr/lib
```

NOTE: Using the `cd` command without any arguments or with more than one argument doesn't affect the current directory path. Make sure this edge case is handled in a way that prevents crashes.

Closing the Shell

Inputting either `quit` or `exit` should close the minishell.

Running an Application

Suppose you have an executable named `sum` in the current directory. It takes arbitrarily many numbers as arguments and prints their sum to `stdout`. The following example shows how the minishell implemented by you should behave.

```
> ./sum 2 4 1
7
```

If the executable is located at the `/home/student/sum` absolute path, the following example should also be valid.

```
> /home/student/sum 2 4 1
7
```

Each application will run in a separate child process of the minishell created using `fork`.

Environment Variables

Your shell will support using environment variables. The environment variables will be initially inherited from the `bash` process that started your minishell application.

If an undefined variable is used, its value is the empty string: `""`.

NOTE: The following examples contain comments which don't need to be supported by the minishell. They are present here only to give a better understanding of the minishell's functionalities.

```
> NAME="John Doe"                   # Will assign the value "John Doe" to the N
> AGE=27                            # Will assign the value 27 to the AGE varia
> ./identify $NAME $LOCATION $AGE   # Will translate to ./identify "John Doe" "
```

A variable can be assigned to another variable.

```
> OLD_NAME=$NAME    # Will assign the value of the NAME variable to OLD_NAME
```

Operators

Sequential Operator

By using the `;` operator, you can chain multiple commands that will run sequentially, one after another. In the command `expr1; expr2` it is guaranteed that `expr1` will finish before `expr2` to be evaluated.

```
> echo "Hello"; echo "world!"; echo "Bye!"
Hello
world!
Bye!
```

Parallel Operator

By using the `&` operator you can chain multiple commands that will run in parallel. When running the command `expr1 & expr2`, both expressions are evaluated at the same time (by different processes). The order in which the two commands finish is not guaranteed.

```
> echo "Hello" & echo "world!" & echo "Bye!"  # The words may be printed in any
world!
Bye!
Hello
```

Pipe Operator

With the `|` operator you can chain multiple commands so that the standard output of the first command is redirected to the standard input of the second command.

Hint: Look into anonymous pipes and file descriptor inheritance while using `fork`.

```
> echo "Bye"                        # command outputs "Bye"
Bye
> ./reverse_input
Hello                               # command reads input "Hello"
olleH                               # outputs the reversed string "olleH"
> echo "World" | ./reverse_input  # the output generated by the echo command wil
dlroW
```

Chain Operators for Conditional Execution

The `&&` operator allows chaining commands that are executed sequentially, from left to right. The chain of execution stops at the first command **that exits with an error (return code not 0)**.

```
# throw_error always exits with a return code different than 0 and outputs to st
> echo "H" && echo "e" && echo "l" && ./throw_error && echo "l" && echo "o"
H
e
l
ERROR: I always fail
```

The `||` operator allows chaining commands that are executed sequentially, from left to right. The chain of execution stops at the first command **that exits successfully (return code is 0)**.

```
# throw_error always exits with a return code different than 0 and outputs to st
> ./throw_error || ./throw_error || echo "Hello" || echo "world!" || echo "Bye!"
ERROR: I always fail
ERROR: I always fail
Hello
```

Operator Priority

The priority of the available operators is the following. The lower the number, the **higher** the priority:

1. Pipe operator (`|`)
2. Conditional execution operators (`&&` or `||`)
3. Parallel operator (`&`)
4. Sequential operator (`;`)

I/O Redirection

The shell must support the following redirection options:

- `< filename` - redirects `filename` to standard input
- `> filename` - redirects standard output to `filename`
- `2> filename` - redirects standard error to `filename`
- `&> filename` - redirects standard output and standard error to `filename`
- `>> filename` - redirects standard output to `filename` in append mode
- `2>> filename` - redirects standard error to `filename` in append mode

Hint: Look into open, dup2 and close.

Support Code

The support code consists of three directories:

- `src/` is the skeleton mini-shell implementation. You will have to implement missing parts marked as `TODO` items.
- `util/` stores a parser to be used as support code for implementing the assignment. For more information, you can check the `util/parser/README.md` file. You can use this parser or write your own.
- `tests/` are tests used to validate (and grade) the assignment.

Building mini-shell

To build mini-shell, run `make` in the `src/` directory:

```
student@so:~/.../assignment-mini-shell$ cd src/

student@so:~/.../assignment-mini-shell/src$ make
```

Testing and Grading

The testing is automated. Tests are located in the `tests/` directory.

```
student@so:~/.../assignment-mini-shell/tests$ ls -f
Makefile  grade.sh  run_all.sh  _test/
```

To test and grade your assignment solution, enter the `tests/` directory and run `grade.sh`. Note that this requires linters being available. The easiest is to use a Docker-based setup with everything installed, as shown in the section "Running the Linters". When using `grade.sh` you will get grades for correctness (maximum `90` points) and for coding style (maximum `10` points). A successful run will provide you an output ending with:

```
### GRADE


Checker:                                                 90/ 90
Style:                                                   10/ 10
Total:                                                  100/100


### STYLE SUMMARY
```

Running the Checker

To run the checker and everything else required, use the `make check` command in the `tests/` directory:

```
student@so:~/.../assignment-mini-shell/tests$ make check
make[1]: Entering directory '...'
rm -f =~
[...]
16) Testing sleep command.....................................failed  [ 0/100]
17) Testing fscanf function...................................failed  [ 0/100]
18) Testing unknown command...................................failed  [ 0/100]

                                                              Total:   0/100
```

For starters, tests will fail.

Each test is worth a number of points. The total number of points is `90`. The maximum grade is obtained by dividing the number of points to `10`, for a maximum grade of `9.00`.

A successful test run will show the output:

```
student@so:~/.../assignment-mini-shell/tests$ make check
make[1]: Entering directory '...'
rm -f =~
[...]
01) Testing commands without arguments.......................passed  [03/100]
02) Testing commands with arguments..........................passed  [02/100]
03) Testing simple redirect operators........................passed  [05/100]
04) Testing append redirect operators........................passed  [05/100]
05) Testing current directory................................passed  [05/100]
06) Testing conditional operators............................passed  [05/100]
07) Testing sequential commands..............................passed  [03/100]
08) Testing environment variables............................passed  [05/100]
09) Testing simple pipe......................................passed  [05/100]
10) Testing multiple pipes...................................passed  [10/100]
11) Testing variables and redirect...........................passed  [05/100]
12) Testing overwritten variables............................passed  [02/100]
13) Testing all operators....................................passed  [03/100]
14) Testing parallel operator................................passed  [10/100]
15) Testing big file.........................................passed  [05/100]
16) Testing sleep command....................................passed  [07/100]
17) Testing fscanf function..................................passed  [07/100]
18) Testing unknown command..................................passed  [04/100]

                                                              Total:   90/100
```

The actual tests are located in the `inputs/` directory.

```
student@so:~/.../assignment-mini-shell/tests/$ ls -f _test/inputs
test_01.txt  test_03.txt  test_05.txt  test_07.txt  test_09.txt  test_11.txt  te
test_02.txt  test_04.txt  test_06.txt  test_08.txt  test_10.txt  test_12.txt  te
```

Running the Linters

To run the linters, use the make lint command in the `tests/` directory:

```
student@so:~/.../assignment-mini-shell/tests/$ make lint
[...]
cd .. && checkpatch.pl -f checker/*.sh tests/*.sh
[...]
cd .. && cpplint --recursive src/ tests/ checker/
[...]
cd .. && shellcheck checker/*.sh tests/*.sh
```

Note that the linters have to be installed on your system: `checkpatch.pl`, `cpplint`, `shellcheck` with certain configuration options.

Debugging

To inspect the differences between the output of the mini-shell and the reference binary set `DO_CLEANUP=no` in `tests/_test/run_test.sh`. To see the results of the tests, you can check `tests/_test/outputs/` directory.

Memory leaks

To inspect the unreleased resources (memory leaks, file descriptors) set `USE_VALGRIND=yes` and `DO_CLEANUP=no` in `tests/_test/run_test.sh`. You can modify both the path to the Valgrind log file and the command parameters. To see the results of the tests, you can check `tests/_test/outputs/` directory.

Previous
« Parallel Graph
Next
Asynchronous Web Server »

Copyright © 2024 SO Team