Mini-Shell Assignment: Building a Bash-Like Mini-Shell

In 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 can be found in the README file included in the repository.

This document provides an overview of assignment's requirements and some tips to write, build, and test your code. Carefully read the README file as it contains examples and further explanations. Tests files are in tests/ directory. Inner folders of tests/ directory include text files that store multiple lines of commands which will be passed to your mini-shell.

Grading your assignment will only be based on the number of tests your code passes. The maximum score is 90

Required Shell Functionalities

1. 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. 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.

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

2. Closing the shell

Inputting either quit or exit should close the mini-shell.

3. Running an application

The mini-shell should be able to run executables and be able to handle arbitrarily many numbers as arguments. Each application will run in a separate child process of the mini-shell created using fork. Look at the README file for examples.

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

4. Environment variables

Your shell will support using environment variables. The environment variables will be initially inherited from the bash process that started your mini-shell application. If an undefined variable is used, its value is the empty string: "". Look at the README file for examples

```
> NAME="John Doe"      # will assign the value “john Doe” to the NAME variable
> AGE=27               # will assign the value 27 to the AGE variable
> ./identify $NAME $LOCATION $AGE   # Will translate to ./identify "John Doe"
"" 27 because $LOCATION is not defined
```

Operators

Your mini-shell should support the following 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* is 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 order
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.

```
> 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
will be used as input for the reverse_input executable
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
stderr "ERROR: I always fail"
> echo "H" && echo "e" && echo "l" && ./throw_error && echo "l" && echo "o"
H
e
l
ERROR: I always fail
```

While 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
stderr "ERROR: I always fail"
> ./throw_error || ./throw_error || echo "Hello" || echo "world!" || echo
"Bye!"
ERROR: I always fail
ERROR: I always fail
Hello
```

Operator's 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

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. Redirection allows commands' file handles to be duplicated, opened, closed, made to refer to different files, and can change the files the command reads from and writes to. Redirection may also be used to modify file handles in the current shell execution environment. The following redirection operators may precede or appear anywhere within a simple command or may follow a command. Redirections are processed in the order they appear, from left to right. Here are some few types of redirections with its format.

Redirecting Input

Redirection of input causes the file whose name results from the expansion of *word* to be opened for reading on file descriptor *n,* or the standard input (file descriptor 0) if *n* is not specified.

```
> [n]<word
```

Redirecting Output

Redirection of output causes the file whose name results from the expansion of *word* to be opened for writing on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created; if it does exist it is truncated to zero size.

```
> [n]>[|]word
```

Redirecting Standard Output and Standard Error

This construct allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be redirected to the file whose name is the expansion of *word*.

```
> &>word
```

Appending Standard Output and Standard Error

This construct allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be appended to the file whose name is the expansion of *word*.

```
> &>>word
```

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

Implementation approach:

The parser

This assignment utilizes a Bison parser which can be found in *util/parser/.* An overview of Bison parser can be found in the background section of this document. This parser is used to parse the commands entered by the user, stored in a *command_t* structure. You should make the parser before running your code. Please read *util/parser/README.m* for further information.

The shell

You will have to implement missing parts marked as TODO items in *src/cmd.c* so that the mini-shell works correctly when the command tree is given as command_t type.

Background information

The Bison Parser

Bison is a powerful parser generator, which means it is a tool for creating parsers. It is commonly used to develop parsers for programming languages, interpreters, and compilers.

In order for Bison to parse a language, it must be described by a *context-free grammar*. This means that you specify one or more *syntactic groupings* and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an 'expression'. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

The Bison parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is valid, the end result is that the entire token sequence reduces to a single grouping whose symbol is the grammar's start symbol. If we use grammar for C, the entire input must be a 'sequence of definitions and declarations'. If not, the parser reports a syntax error.

When you run Bison, you give it a Bison grammar file as input. The most important output is a C source file that implements a parser for the language described by the grammar. This parser is called a *Bison parser*, and this file is called a *Bison parser implementation file*. Keep in mind that the Bison utility and the Bison parser are two distinct programs: the Bison utility is a program whose output is the Bison parser implementation file that becomes part of your program.

The job of the Bison parser is to group tokens into groupings according to the grammar rules, for example, to build identifiers and operators into expressions. As it does this, it runs the actions for the grammar rules it uses.

The tokens come from a function called the *lexical analyzer* that you must supply in some fashion (such as by writing it in C). The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what "inside" the tokens (though their semantic values may reflect this). Typically, the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this

Useful APIs:

pid_t:

Data type representing a Process ID (typically an integer). Usually returned by fork(), getpid(). And used by waitpid() and process management functions.

*waitpid(pid_t pid, int *status, int options)*

This function waits for a specific child process to change state (e.g., terminate).

- pid: Process ID to wait for (-1 = any child).
- status: Pointer to store exit status (use macros like WEXITSTATUS to decode).
- options: Flags (e.g., WNOHANG to return immediately if no child exited).
- Returns Child PID on success, -1 on error.

WEXITSTATUS(int status)

A macro to extract the child's exit code.

- Argument: status value set by waitpid().
- Returns: Exit code (8 bits, 0-255).

*open(const char *path, int oflag, ...)*

This function opens a file, returning a file descriptor.

- path: File path.
- oflag: Flags (can be combined with | operator):
  - O_RDONLY: Open for reading.
  - O_WRONLY: Open for writing.
  - O_CREAT: Create file if it doesn't exist (requires a third mode argument, e.g., 0644).
  - O_APPEND: Write at end of file.
  - O_TRUNC: Truncate file to length 0 if it exists.
- Returns file descriptor (≥0) on success, -1 on error.

*chdir(const char *path)*

Changes the current working directory. It returns: 0 on success, -1 on error.

fork()

Creates a new process (child). It returns 0 in the child, child's PID in the parent, -1 on error.

pipe(int fildes[2]):

This function creates a pipe for inter-process communication.

- Arguments: fildes[0] = read end, fildes[1] = write end.
- Returns: 0 on success, -1 on error.

dup2(int oldfd, int newfd)

This function duplicates oldfd (old file descriptor) to newfd (new file descriptor) (closes newfd first if open). It can be used to redirect input/output. It returns newfd on success, -1 on error.

close(int fd)

This function closes a file descriptor and returns 0 on success, -1 on error.

*execvp(const char *file, char *const argv[])*

Function to replace the current process with file (searches PATH).

- file: Executable name.
- argv: NULL-terminated argument array.
- Returns: Only on error (-1).

exit(int status)

Terminates the process with exit code status (0 = success).

*getcwd(char *buf, size_t size)*

A function to get the current working directory.

- buf (buffer to store path), size (buffer size).
- It returns buf on success, NULL on error.

*setenv(const char *name, const char *value, int overwrite)*

This function is used to Set environment variable name to value.

- overwrite ≠ 0 replaces existing variable.
- Returns: 0 on success, -1 on error.

*strcmp(const char *s1, const char *s2)*

This function compares two strings lexicographically. It returns 0 if they are equal and the character count difference in integers.

*fprintf(FILE *stream, const char *format, ...)*

Prints formatted output to a stream and returns the number of characters printed.

stderr:

Standard error stream (unbuffered by default).

EXIT_FAILURE:

Macro for exit code indicating failure.

File Descriptors:

- STDIN_FILENO: Standard input (0).
- STDOUT_FILENO: Standard output (1).
- STDERR_FILENO: Standard error (2).

Tips:

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

- To test and grade your solution, enter the tests/ directory and run grade.sh