

My Shell (aka Command Line Interpreter)

Learning Objectives

Upon completion of this assignment, you should understand how to:

1. apply basic C/UNIX process management system calls
2. manipulate and redirect a program's standard input, output and error streams
3. do basic inter-process communication operations

The mechanisms you will practice using include:

- Process management: `fork()`, `wait()`, `exec()`
- File I/O: `open()`, `close()`, `dup2()`
- Inter-process communication: `pipe()`

Program Specification

NAME

`mysh` – run a command line interpreter

SYNOPSIS

`mysh [prompt]`

DESCRIPTION

`mysh` invokes a command line interpreter that supports command execution and input/output redirection. `mysh` takes a single optional command line argument that specifies the prompt string. If no arguments are given, the prompt defaults to "`mysh:` ". If "`-`" is given, do not print a prompt.

The basic `mysh` operation is:

1. print prompt, "`mysh:` "
2. read a single input line from standard input
3. execute foreground or background command(s) as specified
4. wait for command(s) executed in the foreground to complete
5. go to 1

OPERATORS

On an input command line, commands, operators and operands are always separated by whitespace. `mysh` supports the following command line operators:

`&`

Place commands into the background: after invoking the specified commands, `mysh` immediately prints its prompt and waits for another command line.

'&' may only be specified as the final command line argument.

`<`

Redirect the current command's standard input stream from the file named immediately after the '`<`' operator.

- >
Redirect the current command's standard output stream to the file named immediately after the '>' operator. `mysh` never "clobbers" an existing output file.
- >>
Redirect the current command's standard output stream to the file named immediately after the '>>' operator. Append to the file if it already exists.
- |
Redirect the current command's standard output stream to the standard output stream of the succeeding command. There may be *any* number of pipe-connected processes.

EXITING

`mysh` exits with 0 when the user inputs "exit" or CNTL-D. These are the only circumstances under which `mysh` exits. When command lines are malformed or fail to execute, `mysh` prints an appropriate error message then prompts the user for another command line.

ERRORS

Upon error, `mysh` prints one of the following statements to the standard error stream:

```
"Error: Usage: %s [prompt]\n", program_name
"Error: \"&\" must be last token on command line\n"
"Error: Ambiguous input redirection.\n"
"Error: Missing filename for input redirection.\n"
"Error: Ambiguous output redirection.\n"
"Error: Missing filename for output redirection.\n"
"Error: Invalid null command.\n"
"Error: open(\"%s\"): %s\n", file_name, strerror(errno)
```

Implementation Details

`mysh` searches the PATH environment variable for executables; i.e., `mysh` uses `execvp()`.

When opening files with `O_CREAT`, use "`S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH`" for the mode, i.e. the third argument to `open()`.

Requirements and Constraints

1. Command lines are restricted to 1024 characters or less.
2. You may use `wait()` and `wait3()`, but not `waitpid()`.
3. When executed in the foreground, `mysh` waits for all processes in a pipe to complete.
4. Kill all stray processes left around after quitting `mysh`
5. Always check to make sure `execvp()` did not unexpectedly fail.

6. Transient zombies may exist, but `mysh` should periodically clean up zombie processes.
7. Close unneeded file descriptors. When a child process calls `execvp()`, there should only be three open file descriptors – 0, 1, and 2. Likewise when the main `mysh` process is fetching a command line. `mysh` should not use a file descriptor greater than 5.
8. Comprehensively handle all error conditions, including operator misuse (e.g. trying to redirect output to a pipe and a file at the same time) and bad commands.
9. If a command line is malformed, no part of it should be executed. (You do not need to check whether commands are valid or have execute permissions.)

Tips

1. Use `fgets()` to read input lines.
2. Check `fgets()` and `wait()` for premature returns due to system interruption: if `fgets()` or `wait()` fails and `errno == EINTR`, try the call again!
3. You may use the provided `tokens.c` and `tokens.h` files to “tokenize” your command lines.
4. Remember in `execvp()`’s argument vector, the first element must be the command itself and the last element must be `NULL`.
5. You will need to track the pids of all foreground command processes in order to make sure they have all terminated before fetching a new command line.
6. Consider implementing and testing in the following order:
 - a. Parsing command line into individual commands. FWIW:
 - i. I created a `CmdSet` struct and a `Cmd` struct.
 - ii. For each command line, my `CmdSet` struct contains an array of `Cmd` structs, one for each command, and whether they should be executed in the foreground or background
 - iii. My `Cmd` Struct contains the commands argument vector, input filename (if redirected) and output filename (if redirected)
 - b. Executing commands in the foreground
 - c. Executing commands in the background
 - d. Redirecting input/output in the foreground
 - e. Redirecting input/output in the background
 - f. Implementing a single pipe
 - g. Implementing two or more pipes
7. I separated the command line processing phase from the command line execution phase and checked for all command line issues, including bad I/O redirection, during command line processing, i.e. before executing any part of the command line.
8. Consider the case when there is a foreground process running, and a background process terminates. Make sure `mysh` doesn’t try to fetch a new command line prematurely.