CS 214 Spring 2024
Project III: My shell

David Menendez

Due: April 8, at 11:59 PM (ET)

For this project, you and your partner will design and implement a simple command-line shell, similar to `bash` or `zsh`. Your program, `mysh`, will provide *interactive* and *batch* modes, both of which will read and interpret a sequence of commands.

This project will provide experience of

- Posix (unbuffered) stream IO
- Reading and changing the working directory
- Spawning child processes and obtaining their exit status
- Use of `dup2()` and `pipe()`
- Reading the contents of a directory

1 Overview

Your program `mysh` takes up to one argument. When given one argument, it will read commands from the specified file. When given no argument, it will read commands from standard input.

Using `isatty()`, `mysh` will determine whether to run in *interactive mode* or *batch mode*. The difference between these is that interactive mode (when the input comes from a terminal) will print welcome and good-bye messages and a prompt before reading the next command, while batch mode (when not reading from a terminal) prints nothing.

For full credit, your program must have one input loop and command parsing algorithm that works for both modes.

Your code must use `read()` to obtain input, and should be careful to (1) obtain a full command before executing it, and (2) not call `read()` after receiving a newline character until it has executed the received command. The second point is important to ensure that `mysh` is interactive and does not block waiting for additional input when it has already received a complete command.

`mysh` terminates when it receives the `exit` command, or when its input stream ends.

Welcome and good-bye When running in interactive mode, `mysh` should print a message such as "Welcome to my shell!" before printing the first command prompt. Additionally, after receiving the exit file or reaching the end of the input stream, it should print a message such as "Exiting my shell." (Feel free to choose whatever wording you like for these messages.)

Prompt format When running in interactive mode, `mysh` will print a prompt to indicate that it is ready to read input. The prompt is normally the string "mysh> " (note the trailing space).

Usage Batch mode:

```
$ cat myscript.sh
echo hello
$ ./mysh myscript.sh
hello
$
```

Batch mode with no specified file:

```
$ cat myscript.sh | ./mysh
hello
$
```

Interactive mode:

```
$ ./mysh
Welcome to my shell!
mysh> cd subdir
mysh> echo hello
hello
mysh> cd subsubdir
mysh> pwd
/current/path/subdir/subsubdir
mysh> cd directory_that_does_not_exist
cd: No such file or directory
mysh> cd ../..
mysh> exit
mysh: exiting
$
```

2 Command format

`mysh` reads one command per line, which describes a *job*. A typical job will involve one or more child processes executing specified programs with specified arguments and possibly overridden standard input or output files.

When processing a command, `mysh` must determine the number of child processes to start and, for each one,

1. The path to the executable file
2. The list of argument strings
3. Which files to use for standard input and output

To accomplish this, `mysh` will process the command line in stages.

The command itself is made of *tokens*. For our purposes, a token is a sequence of non-whitespace characters, except that `>`, `<`, and `|` are always a token on their own. Thus, a string `foo bar<baz` consists of four tokens: "foo", "bar", "<", and "baz".

The token stream determines the argument list and whether the command includes redirection. We can build the argument list as we step through the token stream.

Wildcards If a token includes the wildcard character `*`, it describes a pattern of file path names. For simplicity, you may assume that patterns contain only one asterisk, which is always in the final path segment (e.g., `foo*txt` or `../bar/*.c`). The wildcard only applies to file names, not paths, so any names it produces will be in the same directory.

Open the directory that would contain these files and search for names matching the pattern. Add each name to the argument list.

If no names match the pattern, add the original token to the argument list.

Redirection The special tokens `<` and `>` introduce input and output redirection, respectively. The first argument following the redirection flag is not added to the argument list, but instead kept as a redirection file name.

Note that redirection can occur anywhere in a command. Tokens following the redirection file name are treated as regular arguments.

Pipelines The special token `|` separates two programs in a *pipeline*, each of which will have a separate argument list and file redirection targets. Additionally, `mysh` will use `pipe()` to arrange that standard output for the first process will be written to standard input for the second process.

For simplicity, you may limit `mysh` to pipelines involving two processes.

Conditionals If the first token is `then` or `else`, it is not added to the argument list. Instead, `mysh` will check the exit status of the previously executed command. A command beginning with `then` is only executed if the previous command succeeded, and a command beginning with `else` is only executed if the previous command failed. If a conditional command is not executed, the previous exit status remains unchanged.

Conditionals apply to the entire command, even if it includes a pipeline. Use of `then` or `else` after a `|` is invalid.

Note that we cannot use these to create a true if-then-else block.

Examples This command,

```
then foo < bar baz
```

includes a conditional, and describes the argument list ("foo", "baz"), and redirects standard input to "bar".

This command,

```
foo bar < baz | quux *.txt > spam
```

executed in a directory containing files "bacon.txt" and "eggs.txt" will start a pipeline with two processes.

The first has argument list ("foo", "bar"), with input redirected to "baz".

The second has argument list ("quux", "bacon.txt", "eggs.txt"), with output redirected to "spam".

In this sequence of commands,

```
foo
else bar
else baz
```

"baz" will only be executed if both "foo" and "bar" report failure.

In this sequence of commands,

```
foo
then bar
else baz
```

"baz" will execute if "foo" fails or if "foo" succeeds and "baz" fails. This is intentional.

In this sequence of commands,

```
foo
else exit whoops
bar
```

if "foo" fails, `mysh` will print "whoops" and terminate.

2.1 Program name

The first argument determines which program is executed. This argument may be the name of a program, a path to a program file, or a "built-in" command.

Pathnames If the first argument contains a slash character, `/`, `mysh` will assume it is a path to an executable file.

Bare names If the first argument does not contain a slash, and is not one of the built-in names, `mysh` will search for the program. For this assignment, we will only search the directories `/usr/local/bin`, `/usr/bin`, and `/bin`, in that order.

You may use `access()` to determine whether a file with the appropriate name exists in these directories without needing to traverse the directory itself.

Note that most commands are executed in this way and are not built in to the shell. Do not attempt to implement common Unix commands (cp, mv, cat, etc.) yourself!

Built-in commands The commands `cd`, `pwd`, `which`, and `exit` are implemented by `mysh` itself. (Programs by these names may exist in the three directories above, but you should ignore those.)

`cd` is used to change the working directory. It expects one argument, which is a path to a directory. `mysh` should use `chdir()` to change its own directory. `cd` should print an error message and fail if it is given the wrong number of arguments, or if `chdir()` fails.

`pwd` prints the current working directory to standard output. This can be obtained using `getcwd()`.

`which` takes a single argument, which is the name of a program. It prints the path that `mysh` would use if asked to start that program. (That is, the result of the search used for bare names.) `which` prints nothing and fails if it is given the wrong number of arguments, or the name of a built-in, or if the program is not found.

`exit` indicates that `mysh` should cease reading commands and terminate. Additionally, `exit` should print any arguments it receives, separated by spaces.

While these built-in commands are implemented by the shell, and do not usually involve creating a subprocess, `mysh` should otherwise treat them as regular commands and allow them to participate in redirection and pipes. (All three of these commands will simply ignore standard input, but `pwd`, `which`, and `exit` have output that could be sent to a file or piped to another program.)

Note also that `mysh` terminates *after* executing a job involving `exit`. A command such as `foo | exit` will terminate `mysh` once foo is complete.

2.2 Wildcards

A token containing an asterisk (`*`) is a *wildcard* (or *glob*), representing a set of files whose names match a pattern. We allow a single asterisk in a file name or in the last section of a path name. Any file in the specified directory whose name begins with the characters before the asterisk and ends with the characters after the asterisk is considered to match.

In other words, any name where we can replace a sequence of zero or more characters with an asterisk to obtain the pattern is considered a match.

Thus, `foo*bar` matches file names in the working directory that begin with "foo" and end with "bar".

Similarly, `baz/foo*bar` matches file names in the subdirectory "baz" that begin with "foo" and end with "bar".

When a command includes a wildcard token, it will be replaced in the argument list by the list of names matching the pattern.

If no names match the pattern, `mysh` should pass the token to the command unchanged.

Hidden files Patterns that begin with an asterisk, such as `*.txt`, will not match names that begin with a period.

2.3 Redirection

The tokens $<$ and $>$ are used to specify files for a program to use as standard input and standard output, respectively. The token immediately following the $<$ or $>$ is understood as a path to the file, and is not included in the argument list for the program.

Normally, a child process will use the same standard input and output as its parent. When using file redirection, `mysh` should open the specified file in the appropriate mode and use `dup2()` in the child process to redefine file 0 or 1 before calling `execv()`.

When redirecting output, the file should be created if it does not exist or truncated if it does exist. Use mode 0640 (`S_IRUSR|S_IWUSR|S_IRGRP`) when creating.

If `mysh` is unable to open the file in the requested mode, it should report an error and set the last exit status to 1.

2.4 Pipes

A pipe connects standard input from one program to the standard output from another, allowing data to "flow" from one program to the next. `mysh` allows for a single pipe connecting two processes.

Before starting the child processes, use `pipe()` to create a pipe, and then use `dup2()` to set standard output of the first process to the write end of the pipe and standard input of the second process to the read end of the pipe.

If the pipe cannot be created, `mysh` should print an error message and set the last exit status to 1. Otherwise, the exit status of the command is the exit status of the last sub-command.

2.5 Additional notes

When a wildcard does not match any files, it should be included in the argument list as-is.

For simplicity, you may assume that the token immediately following a $<$ or $>$ does not contain a wildcard.

Note that more than one token may have a wildcard. Be sure to handle them in a uniform manner.

It is recommended that you use `execv()` to execute programs, as it can easily be given a variable number of arguments. Note that the arraylist code from earlier can be used to efficiently create an array by repeatedly appending items to its end, without needing to know in advance how long the array will be. This is ideal for creating an argument list in the presence of wildcard tokens.

Do not use `execvp()` or `execlp()` to start programs. They will search a different set of directories than the ones mandated for `mysh`.

A single command can redirect both standard input and standard output. These are not required to be given in any particular order. The following commands are equivalent:

```
foo quux < bar > baz
foo < bar quux > baz
foo > baz < bar quux
```

Pipes and redirection can both override the standard input or output of a subprocess. In the event that both are specified, the redirection takes precedence. For example, this command line

```
foo | bar < baz
```

will send the output of foo to a pipe, but bar will have its standard input connected to the file baz, not the pipe. (The data sent to the pipe will be lost, as presumably intended.)

3 Submission

A system will be provided for you to declare your partnership prior to submission, details forthcoming. Determine in advance which partner will be responsible for submitting the completed assignment. Communication and coordination is key to a successful project!

Submit a Tar archive containing:

- Your source code file(s), including testing code
- Your make file

- Your README
- Any test inputs used by your testing process

Your README should be a plain text document containing the names and NetIDs of both partners.

Your test plan should be part of the README, and detail your testing strategy and test cases. Describe the scenarios you considered it important to check and how you performed those checks. Note that having a good test suite is an excellent way to detect errors that have been introduced or reintroduced into your code.