CS4411 Project 1. Implement a shell

Due Jan. 26 11:50 PM

Purpose

The purpose of this project is to familiar you with Unix system calls.

Project summary

The skeleton code is the starting point for this project. You need to implement the shell to execute commands in the following ways:

1) support running simple commands (e.g. /bin/cat foo.txt bar.txt)
2) support input redirection from a file
   (e.g. commands like /usr/bin/gcc -E - < somefile.txt)
3) support output redirection to a file
   (e.g. commands like /usr/bin/gcc -E file.cc > somefile.txt)
4) support pipelines of multiple commands
   (e.g. commands like /bin/cat foo.txt | /bin/grep bar | /bin/grep baz or /bin/cat foo | /bin/grep baz > ouptut.txt)
5) outputs the exit status of each command
6) prints out error messages to stderr

Specification

1 Shell language

- Shell commands are lines which consist of a sequence of whitespace-separated tokens. Whitespace characters are space (' ' in C or C++), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), vertical tab ('\v').
- Each line has a maximum length of 100 characters.
- Each token is either an operator, if it is < or > or |`, or a word otherwise
- Each line of inputs is a pipeline, which consists of a series of a commands (possibly just one) separated by | tokens.
- Each well-formed command consists of a series of (in any order):
    a. up to one input redirection operation, which consists of a < token followed by a word token

b. up to one output redirection operation, which consists of a > token followed by a word token, and

c. one or more words (not part of any redirection operation), which are used to form the command for an exec system call

- Any command which has a < or > operator not followed by a word token is malformed. Any command which has no word which is not part of a redirection operation is malformed. Any command which has more than one input redirection operation or more than one output redirection operation is malformed.

As a special case, you may optionally accept and execute no programs for a line containing no tokens, or you may treat it as a malformed command.

2 Running commands

To run a pipeline, the shell runs each command in the pipeline, then waits for all the commands to terminate. If any command in the pipeline is malformed, you may instead print an error and not execute any command in the pipeline.

To run a command, the shell:

- checks the command is malformed according to the specification above
- first checks if it is the built-in command listed below, and if so does something special.
- forks off a subprocess for the command
- if it is not the first command in the pipeline, connect its stdin (file descriptor 0) to the stdout of the previous command in the pipeline. The output should go to a pipe created by pipe( ) (see man pipe) before the subprocess was forked. (You may ***not***, for example, create a normal temporary file, even if this sometimes works.)
- if it is not the last command in the pipeline, connect its stdout (file descriptor 1) to the stdin of the next command in the pipeline
- if there is an output redirection operation, reopen stdout (file descriptor 1) to that file. The file should be created if it does not exist, and truncated if it does.
- if there is an input redirection operation, reopen stdin (file descriptor 0) from that file
- uses the first word (ignoring words in redirection operations) as the path of the executable run. The shell should *not* search the PATH environment variable.

3 Output exit statues

After running each command in the pipeline and before prompting for the next command, the shell should wait for all commands in the pipeline and output their exit statuses. To output their exit statuses, for each command **in the order the commands appeared in the pipeline**, the shell should print out *to stdout* information about how the command terminated on its own line:

- if the command terminates due to a signal (e.g. control-C), then print out a line starting with the name of the program, followed by any text of your choice that describes what happened;
- otherwise, then output the name of the program, optionally followed by its arguments, followed by a space, then exit status: then another space, then the numerical exit status. For example, if running test/foo argument results in the foo executable calling exit(99) (where exit is the C exit function), then you may output test/foo exit status: 99 or test/foo argument exit status: 99.

4 Handling errors

Your shell should print out all error messages to stderr (file descriptor 2).

You must use the following error messages:

If an executable does not exist, print an error message containing “Command not found” or “No such file or directory” (case insensitive). You may include other text in the error message (perhaps the name of the executable the user was trying to run) or print out additional messages in this case. Note that “No such file or directory” is what perror or strerror will output for an errno value of ENOENT, as will happen when execv is passed an executable path that does not exist. (See also the manual pages you can get by running man perror or man strerror.)
If a command is malformed according to the language above, print an error message containing “Invalid command” (case insensitive)
You must also print out an error message (but you may choose what text to output) if:

exec fails for another reason (e.g. executable found but not executable)
fork or pipe fail for any reason. Your program must not crash in this case.
opening a input or output redirected file fails for any reason.
If one command in a pipeline results in an error, you must print out at least one error message, but it is okay if other commands in the pipeline are executed even though some error messages are printed out (e.g. it’s okay if something_that_does_not_exist | something_real prints an error about something_that_does_not_exist after starting something_real). It is also acceptable for you to execute none of the commands in the pipeline in this case.

If multiple errors could occur while executing a command, then you may print out messages for any or all of the possible errors. For example, when running the command foo < input.txt > output.txt, if opening both input.txt and output.txt would fail, then you may output an error message about opening input.txt failing, about opening output.txt failing, or both.

Hints

0. Read system call manual

This project involves the following system calls:
Fork, execv, wait, getpid, pipe, exit, dup.
In order to understand the parameters for the above system calls, you need to look up the manual for them. One convenient way is to use man command in the Linux. Since man can show the same query name for the different cases, you can run “man -k syscall_name ” to find the info for the system call, where “syscall_name” is your query.

1. Testing tool

We have supplied a shell_test.py program, which you can run by running make test. This will often produce a lot of output, so you might try redirecting its output to a file like with make test >test-output.txt.

2. A possible order of operations

1) Implement and test parsing commands into whitespace-separated tokens. Collect the tokens into an array or vector to make future steps easier.

2) Implement and test running commands without pipelines or redirection. In this case the list of tokens you have will be exactly the arguments to pass to the command.

3 Parsing

In C++, you can read a full line of input with std::getline. In C, you can read a full line of input with ``fgets`.

In C++, one way to divide a line of input into tokens is by using std::istringstream like

```
std::istringstream s(the_string);
while (s >> token) {
    processToken(token);
}
```

In C, one way to divide a line of input into tokens is by using strsep like

```
char *p = the_string;
char *token;
while ((token = strsep(&p, " \t\v\f\r\n")) != NULL) {
    ...
```

```
}
```
Note that strsep changes the string passed to it.

My reference implementation creates a class to represent each command in a pipeline (|-seperated list of things to run), and a class to represent the pipeline as a whole. I first collect each command line into a vector of tokens, then iterate through that vector to create and update command objects.

Our specification does not require redirection operations to appear in any particular place in commands. This means, for example, that

foo bar < input.txt
and

foo < input.txt bar
and

< input.txt foo bar
are all equivalent.

4 Running commands

Pseudocode for running comands is as follows:

```
for each command in the line {
   pid = fork();
   if (pid == 0) {
      do redirection stuff
      execv ( command, args );
      oops, print out a message about exec failing and exit
   } else {
      store pid somewhere
   }
}
for each command in the line {
   waitpid(stored pid, &status);
   check return code placed in status;
}
```

To implement redirection, probably the most useful function is dup2, which can replace stdin (file descriptor 0) or stdout (file descriptor 1) with another file you have opened. When redirecting to a file, you will most commonly use open() to open the file, call dup2 to replace stdin or stdout with a copy of the newly opened file descriptor, then close() the original file

descriptor. This occurs typically would be done just before the call to execve as in the pseudocode above.

To implement pipelines, the typical technique is to call pipe to obtain a connected pair of file descriptors, use dup2 to assign these to stdout or stdin, and close the original file descriptor just before calling execve.

To convert from a const char** or array of const char*s to the type expected by execv, you can use a cast like (char**) pointer_to_first_element.

5 Printing Error Messages

In C++, one way to print to stderr is using cerr, which works like cout:

```
#include <iostream>
...
std::cerr << "Some message.\n";
```

In C or C++, one way to print to stderr is using fprintf:

```
#include <stdio.h>
...
fprintf(stderr, "Some message.\n");
```

6 Common problems

1) My shell hangs

If pipelines hang, then a likely cause is neglecting to close the ends of the pipes the parent process. (reads on the pipe will not indicate end-of-file until all of the write ends of the pipe are closed.)

2) My shell stops working after I run too many commands

A likely cause is running out of file descriptors by failing to close all file descriptors.

7 Submission

1 Issue “make submit” to generate the compressed project files and rename it to be shell-lastname.tgz.
2 Upload the shell-lastname.tgz to Canvas.

Credit

This assignment is based on Charles Reiss's assignment.