CSC 209 Assignment 3, Summer 2019: Processes

Due by the end of Friday July 26, 2019; no late assignments without written explanation.

Part 1: A trivial shell

In this part of the assignment you will examine the basic operations needed to execute commands in a shell with execve(). In /u/csc209h/summer/pub/a3/mysh you will find code which does the basic shell loop and parsing of the command-line; your task is to add the execution of the parsed command. Note how mysh.c reads an input line in a loop, and parses it using parse() from parse.c.

Copy these files into a new directory (including Makefile), copy in parsepath.c either from your assignment two solutions or from mine, type "make", and try the resulting program.

You will modify execute.c (only), removing the current demonstration insides of execute() and making it execute the passed-in already-parsed command. See parse.h for the data structure, but it's quite simple: two possible file names for redirection (NULL if the specified channel is not to be redirected), and an argv-like array of strings which are the individual words in the command.

Note that the parsing is quite simple—whitespace characters are the only special characters other than the i/o redirection symbols. For example, "cat file" works as expected, but "echo x; echo y" will not (the semicolon will just be considered to be part of the arguments to 'echo').

The first thing your execute() function has to do is to find the command. If argv[0] does not contain a slash, look for it in the list of directories supplied as the already-parsed PATH variable (also parameters to execute()). (Note: Do not opendir() the directories; just put together the path name and stat() it to see whether it exists. A real shell would then use the returned "struct stat" to check whether it's executable, but we won't do that for this assignment.) If the command is not found in any of these directories, print the usual error message: "%s: Command not found\n".

However, if argv[0] does contain a slash, then it is a path name (absolute or relative) and should be attempted to be executed directly, without searching the directories in the PATH. For example, the user can type /bin/cat, and this doesn't mean /bin/bin/cat, or /usr/bin/bin/cat, or even ./bin/cat—it just means /bin/cat as typed. Also, "./cat" means to run cat in the current directory, even though "/bin/./cat" would be a valid name for /bin/cat. So just like in a real shell, the user can use "./" to evade the search algorithm. For another example which you can test: you could set a PATH variable (before invoking mysh) such as `"/bin:/usr:"`, and then you could try (for example) "bin/sort" as a command, which should execute bin/sort instead of executing /usr/bin/sort.

The argv value passed to your execute() function differs from the standard argv parameter to main() in C, in that the argument list is terminated with a NULL pointer; thus we do not need an argc parameter. This is the format which execve() expects; that is to say, your second argument to execve() will simply be p–>argv. The third parameter to execve() will be the global variable "environ". You can declare it with "extern char **environ;".

When you wait() for the command, assign its exit status to the global variable "laststatus". Since main() returns laststatus, when you reach end-of-file on the input (e.g. you press control-D), the exit status of the shell is the exit status of the last command.

You may NOT use the 'p' family of exec() functions: execvp(), execlp(), or execvep(). These functions implement the PATH searching which is part of this assignment. Of course in practice we would use those functions when applicable, but *someone* has to implement the path searching; and in this assignment, that someone is you.

Part 2: A binary search tree without malloc(), via fork()

In the second part of the assignment you will write a binary search tree, except that each connection between nodes will be represented by pipes, and each node is a separate process. The processes are arranged in a binary search tree, connected with pipes. Each connection needs bidirectional communication, in which a search request can be sent from parent to child and the result can be sent from

(over)

child to parent; thus each link (graph edge) will correspond to *two* pipes (since a pipe is unidirectional).

The search keys are integers, and the values are arbitrary character strings of size up to 9 bytes (i.e. stored as a string in an array of size 10). The initial process is the root node. The key and value are specified on the command-line (the key is argv[1] and the value is argv[2]).

The initial process reads lines from stdin in a loop. The line can be simply an integer; this means to search for that key. Otherwise, the line can be an integer, a space, and the value; this means to set that key to that value. In either case the (possibly new) key and value are output.

Each process loops until EOF, getting a key or a key+value either from stdin (in the case of the root node process) or from the pipe from its parent (in the case of all other processes). Then, if the key matches its own key, it can return the key+value information up the pipe to its parent (or, if it's the root node, output it to stdout). Otherwise, it passes the request down to either its left child or right child as appropriate. When a process is first created, it doesn't have a left child or right child; when it goes to pass a request down to a child, it calls fork() if necessary (setting up the new child, which then does its own loop).

Do not attempt to height-balance your binary search tree.

Upon EOF from the pipe from the parent, close the pipes to any child node processes, wait() for all child node processes, then exit. Your child nodes, if any, will thus get EOF from their pipes to their parents (you), and do the same; and so on all the way down the tree. Similarly, when the root node process gets EOF from stdin, it will also do this (close pipes to children, wait, exit). This will cause an orderly shutdown of the entire tree when you signal EOF on stdin (e.g. by pressing control-D). (Note that this scheme means that the root node will not exit until all other nodes have exited, which is good because the shell you ran the program from is waiting for the initial process, which is the root node.)

There is "starter code" in /u/csc209h/summer/pub/a3/tree.c.starter which contains functions to parse the key and value data into a simple struct (declared in tree.c.starter) containing an integer plus a ten-char array, and you can easily pass this up or down the tree with single write() and read() calls. To represent a missing value, we use the empty string (i.e. p–>value[0] is '\0').

You will want to copy this tree.c.starter file to your own directory and call it "tree.c". You will submit only this file. You do not need to modify the main() in tree.c.starter, but you can do so if you wish; similarly for the utility functions in tree.c.starter.

Other notes

Your code must be in standard C. It must compile on the teach.cs machines with "gcc –Wall" with no errors or warning messages, and may not use linux-specific or GNU-specific features.

Submit in the usual way, with commands such as

```
submit -c csc209h -a a3 execute.c tree.c
```

and the other 'submit' commands are also as before.

Please see the assignment Q&A web page at

```
http://www.teach.cs.toronto.edu/~ajr/209/a3/qna.html
```

for other reminders, and answers to common questions.