

## CSC 209 Assignment 3, Fall 2011

Due by the end of Friday November 18, 2011; no late assignments without written explanation.

This assignment involves implementing basic command execution such as is performed by any unix shell. The parsing of the command line is supplied; your task is to implement the fork/exec/wait, i/o redirection, and more, as follows.

### The supplied parsing code

In /u/ajr/209/a3 on CDF there is skeleton source code for a feeble little unix shell which I call “fsh”. The code there does some simple parsing of a command line, resulting in the following basic structure (some further fields are described later, and some further fields will not be used in this assignment):

```
struct parsed_line {
    char *inputfile, *outputfile; /* NULL for no redirection */
    struct pipeline *pl; /* the command(s) */
    struct parsed_line *next;
};
struct pipeline {
    char **argv; /* array ending with NULL */
    struct pipeline *next; /* NULL if this doesn't pipe into anything */
};
```

“pl” is a linked list of pipeline elements, where a single command without piping is considered to be a pipeline of length one.

“argv” is an array of strings, by being a pointer to the zeroth element of an array of pointers-to-char, terminated by a null pointer. This ends up being a convenient format for this data because it is exactly what you have to pass to the kernel entry point “execve()”.

struct parsed\_line is a linked list of such items so that we can represent command-lines such as “a;b” and “a&&b”, as described below.

The function “parse()” in parse.c returns one of these struct parsed\_line items. Note how some of it is used in the starter fsh.c to output a partial description of what the command is.

Your task is to replace the dummy execute() function in fsh.c with one which executes the command instead of printing information about it. You will also modify builtin.c for one of the points below.

### Suggested sequence of implementation

1. First, compile and run “testparse” and type some commands to it. Compile it and everything else by typing “make”. In testparse, type zero or more argv lists separated by vertical bars, possibly with an input or output redirection for the entire command. Type some commands separated by semicolons. E.g.

```
ls
ls >file
cat file
who | grep ajr
cat <file | grep something | cat >file2
ls >file; cat file
```

etc.

2. Only look at parse.c as much as you're interested; mostly, look at testparse.c and how it *uses* the parse function and data structures (although we will not be using either of the “isdoubling” facilities in this assignment). Then look at fsh.c to see the context in which you will be writing code to execute these commands. You will replace fsh.c's execute() entirely, but no changes to main() are required (or desirable).

3. Make execute() (in fsh.c) execute a simple command by using execve() (see the man page). Make sure that p->pl is not a null pointer, but otherwise you can ignore the structure of the data structure which is passed in and just use p->pl->argv, which is suitable for passing as the second parameter to execve(). The third parameter to execve() will

(continued)

be the global variable “environ”. You can declare it with “extern char \*\*environ;”.

At first, pass `p->pl->argv[0]` as the first parameter to `execve()`. This will mean you can't type “cat file” but must instead type “/bin/cat file”. We'll fix that in the next step.

Assign the exit status of the command to the global variable “laststatus”. This will be used in three ways, one of which is already implemented—`main()` returns `laststatus`, so when you reach end-of-file on the input (e.g. you press ^D), the exit status of the shell is the exit status of the last command. You can also run shell scripts with “./fsh <file”, although this is messy in several ways which we won't fix for the assignment.

4. Construct a string consisting of various appropriate directory names in turn concatenated with `p->pl->argv[0]` (you will find `efilenamecons()` in `error.c` useful for this), and call `stat()` on these strings as the quickest way to find which file exists and hence which string should be passed as the first parameter to `execve()`. The directory names to use are `/bin`, `/usr/bin`, and “.” (for the current directory), in that order. So for the command “cat”, we would look for `/bin/cat`, `/usr/bin/cat`, and finally `./cat` (except that we would have stopped with `/bin/cat`, which exists).

You will call `stat()` to determine file existence. The return value of `stat` is negative for error (you should proceed to the next potential directory name) or zero for success. So you won't look at any of the data in the struct `stat` (although you would need to in a real shell, to check whether the file is executable), but you still have to pass a pointer to a valid struct `stat` to call `stat` properly.

If the command is not found in any of the list of directories, print the usual error message: “%s: Command not found\n”.

After a failed `execve()`, call `perror()`. The parameter to `perror()` should be the first parameter to `execve()` (*including* the prepended directory name).

Note that the above dance with concatenating strings only applies if `argv[0]` does not contain a slash. Test with `strchr(argv[0], '/')`. For example, the user can still 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`. To summarize this paragraph in other words, if a slash appears anywhere in the `argv[0]` string, it is a complete file pathname (absolute or relative), not to have the search directories prepended.

5. Implement i/o redirection. You have to open the appropriate files *after* the `fork()`, in the child only. Test your implementation with commands such as “`ls >file`” and “`cat <file`”. Note that i/o redirection applies to an entire linked list of “struct pipeline”; if you type a command such as “`cmd1 <file1 | cmd2 | cmd3 >file2`”, “file1” and “file2” should only be opened once each. Also note that the redirection must occur *after* the `fork()`; in testing “`ls >file`”, make sure that you see the next ‘\$’ prompt on (the former) stdout, rather than its going into the file “file” along with the ‘ls’ output.

6. Implement pipelines of length two. That is, if `p->pl->next` is non-null, do a `pipe()` call in the child process, then fork again, then rearrange file descriptors as appropriate in the two youngest processes, and `exec`. Make sure that simple commands still work! Now is also a good time to make sure that you just get another prompt if you just press return (which yields a “pipeline” of length zero). Pipelines of length greater than two are trickier and you don't have to do them for this assignment, but they will be implemented in the posted solution.

7. The “exit” command has to be “built in” to the shell rather than run as a separate process—exit means for *this shell* to exit, not some other process. Near the beginning of `execute()`, checking for possible null pointers along the way, call `strcmp(p->pl->argv[0], "exit")` and if it is equal, instead of doing all the fork stuff, just call the function “`builtin_exit`” (already written for you in `builtin.c`, and declared in `builtin.h`) which takes one parameter of type `char **` which is the `argv`. No `fork()` call occurs. This is a reasonable general strategy for implementing a large number of builtin commands.

8. Implement a “cd” builtin command. (This also has to be built-in, so that it changes the current directory of the shell process, rather than of a new child process which is about to exit anyway.) To implement this, note how `builtin_exit()` does its argument parsing, and add a `builtin_cd()` function in `builtin.c` (it is already declared in `builtin.h`), and call it appropriately from `execute()` in `fsh.c`.

There are three cases for argument parsing: Either there is more than one argument (`argc>2`), which is a usage error just like with “exit”; or there is one argument, in which case you should attempt (with all relevant error

(continued)

checking) to `cd` to `argv[1]`; or there is no argument, in which case you should attempt to `cd` to the user's home directory as identified by the environment variable "HOME". Call `getenv("HOME")` to retrieve this variable, and do check for the (unusual) case that `getenv()` returns `NULL` to indicate that this variable is not present in the environment.

Make sure that `cd` and `exit` interact properly with "laststatus".

9. Implement the connectives ' ; ', '&&', and '| | '. The struct `parsed_line` contains a field "conntype" which is either `CONN_SEQ` for semicolon, `CONN_AND` for '&&', or `CONN_OR` for '| | '.

First, add the loop to iterate through all members of the "struct `parsed_line`" linked list, instead of just executing the first member.

Then you will find that if you type a command like "a && b", it will run 'b' whether 'a' succeeds or not. Use the `laststatus` variable in an appropriate 'if' to decide whether to execute this command: If the `conntype` is `CONN_SEQ`, we execute this command without regard to `laststatus`; if the `conntype` is `CONN_AND`, we execute this command only if `laststatus` is zero (because this command is preceded by '&&'); and if the `conntype` is `CONN_OR`, we execute this command only if `laststatus` is non-zero (because this command is preceded by '| | ').

The `conntype` value for the first command in the line is always `CONN_SEQ`; this simplifies your loop (it means that the first command is not a special case). For example, if the user types

```
cmd1 && cmd2 && cmd3 ; cmd4
```

then the value of the "conntype" field will be `CONN_SEQ` for `cmd1`, `CONN_AND` for `cmd2` and for `cmd3`, and `CONN_SEQ` for `cmd4`.

## Other notes

You will want to begin by making a subdirectory to hold the .c files. You'll want to copy in the starter files from /u/ajr/209/a3, i.e. "`cp /u/ajr/209/a3/* .`". Type "make" to use the supplied Makefile to build your program.

Your C programs must be in standard C. They must compile on the CDF machines with "`gcc -Wall`" with no errors or warning messages, and may not use linux-specific or GNU-specific features.

You will submit your revised `fsh.c` and `builtin.c` files. They will be compiled with the original versions of all of the other files for automated testing. If you have edited the other files (e.g. to put in debugging prints), I strongly recommend copying over all modified files anew from /u/ajr/209/a3 and doing "make clean" and then "make" to produce a `fsh` for your final testing.

Once you are satisfied with your code, you can submit it for grading with the command

```
submit -c csc209h -a a3 fsh.c builtin.c
```

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

Please see the assignment Q&A web page at

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

for other reminders, and answers to common questions.

## Remember:

This assignment is due at the end of Friday, November 18, by midnight. Late assignments are not ordinarily accepted, and *always* require a written explanation. If you are not finished your assignment by the submission deadline, you should just submit what you have, for partial marks.

And *please* be careful not to commit an academic offence in your work on this (or any) assignment, even if you're under pressure. Just submit what you can do yourself; do not look at other students' assignments, and do not show your assignment (complete or partial) to other students. Even a zero out of 10% is far better than cheating and suffering an academic penalty. Students also receive academic offence penalties for giving their assignment to other students, since they are helping that other student to commit an academic offence. Your friend might promise in all sincerity not to hand in your work as their own, but if they can't do the assignment themselves, a copy of your solution is not going to help them enough and when the deadline approaches, they might hand in your work. Don't tempt them.