CS 5600 – Systems and Networks
Programming Assignment 2

Prof. Peter Desnoyers

Overview

For this assignment you'll write a simple shell, which handles simple commands, pipes, and input/output redirection.

It specifically will NOT support:

- job control - you can't start commands in the background with &, and if you type ^Z it will stop your shell, not the command you are running
- wildcard expansion - `*` is just another character:
  ```
  $ ls *.c
  ls: *.c: No such file or directory
  ```
- shell variables, except for `$?`
- control statements like `if`, 'for', etc.

Your shell doesn't need to support unlimited numbers of commands, arguments, or pipeline stages, although it should support at least 32 tokens on a line and 4 pipeline stages.
This lets you use fixed-sized arrays, which make for much simpler C code.

What it WILL support is:

- the `cd`, `pwd`, and `exit` built-in commands
- external programs
- redirection of external program input and output
- pipelines, where the output of one external program is the input of another

At various points in this description you are given instructions to refer to the "man page" for a system call or library function - please do so in a terminal window at that point. Note that much of the contents of a man page can be ignored - the most important parts for what we are doing are (1) the list of include files to use and (2) the arguments and return value. (the "RETURN VALUE" section is often near the end of a long man page)

Step 1: signals

If your shell is interactive, you'll want to disable the `^C` signal, so that you can quit out of a running program without terminating the shell:

```
signal(SIGINT, SIG_IGN); /* ignore SIGINT=^C */
```

Later when you use `fork` to create a subprocess you'll want to set it back to its default in that subprocess, so you can terminate a running command:

```
signal(SIGINT, SIG_DFL);
```

You should be able to run your shell now, and:

- it won't exit when you type ^C
- it will exit properly on end of file (i.e. when you type ^D, which indicates end-of-file on the Unix terminal)

Command status

Each command will have a *status* indicating whether it succeeded or failed.

For an executable run in a separate process, this will be the value passed to the `exit` system call, which we'll get from the `wait` system call in the parent.

For built-in commands it will be 0 for success, and 1 for failure.

Later in the assignment we'll make the status of the previous command available to other commands via the `$?` variable.

Step 2, Internal commands: cd, pwd, and `exit`

Note:

- the command line tokenizer is described below, in the section Command Line Tokenizer
- you can compare strings for equality using `strcmp` ("man 3 strcmp"), which returns zero if two strings are equal.

(question: why does `cd` have to be implemented as a built-in command rather than an executable run in a separate process? `exit`?)

For the `cd` command you will use the `chdir` command ("man 2 chdir") to change to the indicated directory. With no arguments you should use the value of the `HOME` environment variable, i.e. `getenv("HOME")`.

Note that `cd` can fail two ways:

- wrong number of arguments: print `"cd: wrong number of arguments\n"` to standard error - use `fprintf(stderr, ...`
- `chdir` fails: print `"cd: %s\n", strerror(errno)` to standard error

In both cases set status to 1, and set it to 0 otherwise.

`pwd` will use the `getcwd` system call ("man 2 getcwd") to get the current directory, passing it a buffer of `PATH_MAX` bytes, and print the result. You can assume `getcwd` always succeeds and set status to 0.

`exit` takes zero or 1 argument; with more than 1 it prints `"exit: too many arguments"` to stderr and sets status=1. With 0 arguments it calls `exit(0)`; with a single argument it calls `exit(atoi(arg))`, using `atoi`("man 3 atoi") to convert the argument from a string to an integer.

TEST IT: - run your shell, try:
- `pwd` - does it print out the right current directory? does it fail if you give it arguments?
- `cd`-ing to directories that exist, check with `pwd`
- `cd` to non-existent directory, check (a) error message, (b) still in same directory
- `exit` - does it work correctly with 0, 1, >1 argument? Try exiting with an arbitrary non-zero status and verify using the `$?` variable in your normal shell:

```
hw1$ ./shell56
$ exit 5
hw1$ echo $?
5
```

HINT: factor `cd`, `pwd`, and `exit` into individual functions that each take `argc` and `argv` as arguments. Maybe return `status` as the return value, but more on that later.

Now that you've implemented your first commands, make sure that it ignores empty command lines without complaining or crashing.

Step 3, external commands with no I/O redirection

If a command isn't an internal command, it's an external one - you'll fork a sub-process; in the child process you'll use `exec` to run the command, while the parent will use `wait` to wait until it's done.

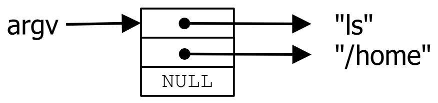
After `fork()` ("man 2 fork") you'll want to do the following:

- re-enable "^C" (see above)
- use the `execvp` library function ("man 3 execvp") to exec the indicated command

From the man page:

```
    int execvp(const char *file, char *const argv[]);
```

The first argument is the executable name, while the second is the `argv` array to be passed to the newly loaded program.
Instead of providing an argument count, the `argv` array is terminated with a NULL pointer.
Thus given the following argument to `execvp`:



`execvp` will load the executable `/usr/bin/ls` and pass it argc=2, argv={"ls", "/home"}.

(question - how does `execvp` know where to find the executable `ls`?)

The command line parser I've given you makes sure that the `argv[]` array is terminated with a NULL pointer, so you can just pass it to execvp:

```
execvp(argv[0], argv);
```

If `execvp` fails, you should print a message to standard error - `"%s: %s\n", argv[0], strerror(errno)` - and then exit with `EXIT_FAILURE`. (question: why do you have to exit here, rather than returning?)

In the parent process you'll need to wait for the child pracess to finish, using `waitpid`, and get its exit status (i.e. the argument passed to `exit()`)
It's ok to copy and paste the following code without fully understanding it:

```
    int status;
    do {
        waitpid(pids[i], &status, WUNTRACED);
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
    int exit_status = WEXITSTATUS(status);
```

Test this:

- successful commands, e.g. `ls`, `ls /tmp`, etc.
- unsuccessful ones, e.g. `this-is-not-a-command`
- ^C handling - run `sleep 5` and verify you can kill it with ^C and return to your shell.

FACTORING: - I suggest that you factor out the code which forks and execs, and put it in a separate function from where you call `waitpid`.

Debugging:

You may find the GDB command `set follow-fork-mode child` useful. documentation

Also the `strace -f` command can be very useful, although verbose - e.g. here's a selection of the 140 lines it prints out for my shell. (note that `fork` in Linux is actually implemented using a system call named `clone`)

```
hw1$ echo ls | strace -f ./shell56
execve("./shell56", ["./shell56"], 0xffffed0a7ba8 /* 24 vars */) = 0
brk(NULL)                               = 0xaaaadecaa000
  ...
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|
SIGCHLDstrace: Process 119667 attached
, child_tidptr=0xffff9934bf50) = 119667
[pid 119667] set_robust_list(0xffff9934bf60, 24 <unfinished ...>
   ...
[pid 119667] execve("/usr/local/sbin/ls", ["ls"], 0xffffe3f4c0a8 /* 24
vars */ <unfinished ...>
        ...
```

Step 4: the $? special shell variable

The basic shell has a number of built-in variables, listed under "Special Parameters" in the man page (`man sh`); we implement only one of these:

? - Expands to the exit status of the most recent pipeline.

To implement this you can just use `sprintf` to print the exit status into a buffer (e.g. `char qbuf[16]`), and then go through your array of tokens, find any which compare equal to `$?`, and replace them with a pointer to that buffer.

Test it:

```
hw1$ ./shell56
$ false
$ echo $?
1
$ ./shell56
$ exit 5
$ echo $?
5
$ exit
hw1$
```

Step 5: Pipes

Limitations: it's OK if you don't handle more than 4 pipeline stages.

To implement pipes you'll need to use the `pipe` and `dup2` system calls. (you might find `dup` useful as well)

Read the "description" section of the man page: "man 2 dup". ("man 2 dup2" gives you the same page)

`dup2(int oldfd, int newfd)` closes `newfd` if it is already open, and makes a *copy* of `oldfd`numbered `newfd`.
In particular you can do something like this:

```
int fd = open("file", O_RDONLY);
dup2(fd, 0);
close(fd);
```

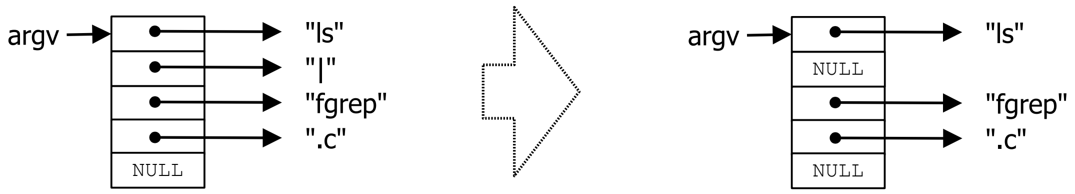
Now standard input will read from "file" instead of whatever it used to be, typically the terminal.
Note that by closing `fd` after `dup2`, we have the same number of open file descriptors when these lines finish as when we started.

The `pipe` call creates two file descriptors, one for reading and one for writing.
Given a pipeline `ls | grep .c` you'll want to use the write fd as standard output (fd 1) for the first command, and the read fd as standard input (fd 0) for the second one.

There are going to be a lot of file descriptors that need closing, as `fork` duplicates everything in the parent process, including open file descriptors, you want *(everything* closed (across parent and children) except standard input and/or output in the appropriate child processes.
Thus for the `ls | grep` pipeline we'll have:

- parent: `pipe` -> `read_fd`, `write_fd`
- child1: `dup2(write_fd, 1)`, `close(write_fd)`, `close(read_fd)`
- child2: `dup2(read_fd, 0)`, `close(read_fd)`, `close(write_fd)`
- parent: `close(read_fd)`, `close(write_fd)`

You'll need to scan your array of tokens looking for "|", and split the line into separate commands that will be piped into each other. Note that you can split the array into parts by replacing "|" with NULL:



This is where it helps to have factored out the code which forks and execs a command.
Make sure you close all the file descriptors that need to be closed - in particular, if you leave an extra copy of the "write" side of a pipe open, the reader will never see end-of-file.
Make sure you don't close the original copies of standard input and standard output - depending on how you factor things, you might find it helpful to make copies of stdin/stdout with `dup` so that your function can always close file descriptors after `fork`.

Finally, keep an array of all the child process IDs, and wait for each of them, one at time, using the logic from above.

Step 6: File redirection

For each pipeline stage, scan for '>" and "<", and replace standard input and output appropriately.

Note that "<" (or ">") may be followed by zero, one, or multiple words before ">" ("<") or end of line:

- zero words: don't redirect
- more than one: redirect to the first one

If you've factored out a "launch" function which takes an argv pointer and file descriptors for stdin and stdout, you can make a "wrapper" for it which checks for file redirection and replaces the appropriate file descriptors if necessary. (make sure you close any file descriptors that aren't needed)

Hint: you can use the `lsof` command to list open file descriptors, to make sure you're not leaking. E.g. from another terminal:

```
hw1$ ps aux |grep shell56
pjd       118403  0.0  0.0   2196   780 pts/4    S+   01:29   0:00
./shell56
hw1$ lsof -a -d 0-999 -p 118403
COMMAND    PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
shell56 118403  pjd    0u   CHR  136,4      0t0    7 /dev/pts/4
shell56 118403  pjd    1u   CHR  136,4      0t0    7 /dev/pts/4
shell56 118403  pjd    2u   CHR  136,4      0t0    7 /dev/pts/4
hw1$
```

(the man page for `lsof` is horrible. The specific options used here will list all open files that are "normal", i.e. with file descriptors 0–999 (`-d 0-999`), AND (`-a`) are open in a specific process (`-p 118403`).

Just like before, the `strace -f` command can be quite useful.

```
hw1$ echo 'ls | cat' | strace -f ./shell56
execve("./shell56", ["./shell56"], 0xffffeccb9e08 /* 24 vars */) = 0
  ... 240 more lines...
```

Background Information

Command Line Tokenizer

The simplest way of tokenizing a line in C is to use the `strtok` library function, or the slightly less horrible `strsep`, which overwrite whitespace characters to split a line into multiple strings.
An example:

start with the line `"ls | cat"`, zero out the whitespace characters:

```
['l']['s'][' ']['|'][' ']['c']['a']['t'][ 0 ]
    -> ['l']['s'][ 0 ]['|'][ 0 ]['c']['a']['t'][ 0 ]
```

and keep pointers to the beginning of each region of non-whitespace characters:

```
argv[]         ['l']['s'][ 0 ]['|'][ 0 ]['c']['a']['t'][ 0 ]
+-----+          ^              ^         ^
|  *--|----------+              |         |
+-----+                         |         |
|  *--|-------------------------+         |
+-----+                                   |
|  *--|-----------------------------------+
+-----+
| ... |
```

Problem: this breaks when you don't have any whitespace, like `"ls|cat"`
The parser you're given handles this by copying the input string into a second buffer, rather than modifying it in place:

```
        input string:                                  buffer:
['l']['s']['|']['c']['a']['t'][ 0 ]     [ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 0 ]

  output:
     argv[]                          -> ['l']['s'][ 0 ]['|'][ 0 ]['c']['a']['t'][ 0 ]
     +-----+                              ^              ^         ^
     |  *--|------------------------------+              |         |
     +-----+                                             |         |
     |  *--|---------------------------------------------+         |
     +-----+                                                       |
     |  *--|-------------------------------------------------------+
     +-----+
     |  0  |  <- terminated with NULL pointer (see arg formats in "man 3 execvp")
     +-----+
     | ... |
```

The skeleton code you're given shows an example of how to use it.

For a "real" shell you'd probably use a tokenizer and parser based on the standard compiler tools `lex`and `yacc`, creating an abstract syntax tree of linked "token" objects. That's far too complicated for this assignment, so we have a simple tokenizer that does a pretty good job of splitting simple lines with redirection symbols and single and double quotes, and returns pointers to strings rather than more complex structures.

The parser is not guaranteed to be bug-free, but your code will only be tested against cases where the

ASCII characters

By default C uses the basic 8-bit ASCII character set, rather than the much larger Unicode character set used in today's user interfaces. To see the actual character set, we can print out a string containing the bytes 1 through 255, with a 256th byte as the null terminator:

```
cat > test.c <<EOF
#include <stdio.h>
int main(void) {
    char c, buf[256];
    for (int i = 0, c = 1; i < 256; i++)
    buf[i] = c++;
    printf("%s", buf);
}
EOF
gcc test.c
./a.out | od -A d -t c
```

You should see the following - note that offsets (left column) are in decimal, while non-printing characters are printed in octal, which no one uses anymore. (“od” = “octal dump”)

The “missing” character at the end of the second line is actually a space, `' '`, and there are several backslash-style escaped characters, of which the only ones we care about are `\n` (newline) and sometimes `\t` (tab).

```
0000000  001 002 003 004 005 006  \a  \b  \t  \n  \v  \f  \r 016 017 020
0000016  021 022 023 024 025 026 027 030 031 032 033 034 035 036 037    
0000032    !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /   0
0000048    1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?   @
0000064    A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P
0000080    Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _   `
0000096    a   b   c   d   e   f   g   h   i   j   k   l   m   n   o   p
0000112    q   r   s   t   u   v   w   x   y   z   {   |   }   ~ 177 200
0000128  201 202 203 204 205 206 207 210 211 212 213 214 215 216 217 220
0000144  221 222 223 224 225 226 227 230 231 232 233 234 235 236 237 240
0000160  241 242 243 244 245 246 247 250 251 252 253 254 255 256 257 260
0000176  261 262 263 264 265 266 267 270 271 272 273 274 275 276 277 300
0000192  301 302 303 304 305 306 307 310 311 312 313 314 315 316 317 320
0000208  321 322 323 324 325 326 327 330 331 332 333 334 335 336 337 340
0000224  341 342 343 344 345 346 347 350 351 352 353 354 355 356 357 360
0000240  361 362 363 364 365 366 367 370 371 372 373 374 375 376 377
```