Overview

For this assignment you'll write a simple shell, which handles simple commands 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-length lines, but should support at least 16 tokens on a line. This lets you use fixed-sized arrays, which make for much simpler C code.

What your shell WILL support is:

- the `cd`, `pwd`, and `exit` built-in commands
- external programs
- redirection of external program input and output to/from files
- pipes between programs

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 (a) the list of include files to use for a library function and (2) the arguments and return value. (the "RETURN VALUE" section is often near the end of a long man page)

Instructions

You are given the following files:

- parser.c, parser.h – command-line parser that handles quotes, etc.
- shell56.c – skeleton code for your shell
- Makefile – to build it. (with two targets – "shell56" (default) and "clean")

The rest of this document describes the steps you will need to take to create your shell.

We urge you to try implementing the shell in the order suggested, and to **TEST AFTER EVERY NEW PIECE OF FUNCTIONALITY**. Please don't try to write the whole thing before you start testing.

Deliverable

We will grade the "shell56.c" file in your Khoury GitHub repository.

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 */
```

There's a line at the top of `main` that figures out whether the shell is interactive; use the boolean value it calculates.

TEST IT: You should be able to compile and 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)

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

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

Things to know before the remaining steps:

Command status: Each command will have an integer *status,* which indicates whether it succeeded or failed; in each step your code will need to keep track of the status, but you won't use it until Step 4, where you implement the **$?** variable, which expands to the decimal status of the last call.

Tokenizer: The command line tokenizer is documented at the end of this document, in the section Command Line Tokenizer. You've been given a shell56.c file that already calls it; you probably won't need to change it.

String equality: C is **really** dumb about strings - if you have a `char*` pointer and you use == to compare it to a fixed string (e.g. `p == "cd"`), C will happily check whether the address in the variable **p** is the same as the address of the fixed string. (spoiler: it's not, and never will be)

Instead you need to use the `strcmp` function ("man 3 strcmp"), which returns zero if two strings are equal; e.g.

```
if (strchr(token,"cd") == 0) { do_cd(); }
```

(Yes, 0 means false in C, and non-zero is true. It's this way because `strcmp` returns negative and positive values to indicate whether one string is lexically "less than" or "greater than" the other, and evidently no one thought of writing a different library function to just check equality)

Which shell?: In a few cases (e.g. testing exit) I'll refer to the "normal" shell - that's what you're typing in before you run your own shell.

Step 2, Internal commands: cd, pwd, and `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, which you can get a pointer to with the `getenv` library function:

```
char *dir = getenv("HOME").
```

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

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 (for "$?" later on) 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 after the `pwd` command.

`exit` takes zero or 1 argument:

- more than 1 argument: print `"exit: too many arguments"` to stderr and sets status=1.
- single argument: call `exit(atoi(arg))`, using `atoi`("man 3 atoi" if you're really curious) to convert the argument from a string to an integer.
- 0 arguments: call `exit(0)`

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 your shell 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.

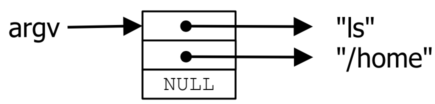
After `fork()` ("man 2 fork") you'll want to do the following:

- re-enable "^C" (see above)
- use the `execvp` library function ("man 3 execvp") to exec the indicated command[1]

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 to ask yourself - 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 process 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 IT:

- 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`.

1 The "p" on the end of "execvp" means that it looks up the command in each of the directories in your search path (the PATH environment variable) and execs the first file with that name that it finds.

Debugging Step 3

Debugging what's going on in a subprocess is hard. A few different ways to debug it:

GDB: If you're debugging with GDB, you may find this command useful:

```
set follow-fork-mode child
```

For more documentation on using follow-fork-mode see this link: link

printf: I'm not a fan of printf debugging, but it's not a terrible way to debug your fork/exec code. Make sure that (a) you have a way of telling whether a printf came from the parent or child, and (b) you call `fflush(stdout)` after every `printf`.

strace: The ”`strace -f`” command can be very useful, although verbose - `strace` prints out every system call that a process makes, and `-f` means to follow into child processes.

Here’s a selection of the 140 lines it prints out for my shell when it runs the `ls` command. (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$
```

Note that we're running shell56 as a command under shell56, and then exit it with a single command "exit 5".

If you think back to the explanation of how stack frames work, you'll realize that `qbuf[]` needs to be either a local variable in `main` or a global variable - it can't be a local variable in a function that returns before we use the pointer.

Step 5: File redirection

[Note - you do NOT have to redirect output from the built-in commands, just from external ones]

To implement this you'll need to scan your array of tokens for '>" and "<", and replace standard input and output appropriately. You'll need to use the 'dup2' system call to replace standard input and standard output – for details see the "description" section of the man page: "man 2 dup" or "man 2 dup2" (they're 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); // error check omitted
dup2(fd, 0);
close(fd);
```

Now standard input (file descriptor 0) 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. For output redirection you'll use the following code to open a file:

```
fd = open(filename, O_WRONLY|O_CREAT|O_TRUNC, 0666);
```

TEST IT:

- `ls > file.out`
- `cat < file.out`

Failure cases

open failure: You need to check whether `open` fails, which it indicates by returning `-1`. If so, you should (a) set status to 1, (b) print the following message to stderr

```
fprintf(stderr, "%s: %s\n", filename, strerror(errno))
```

and (c) loop around to read your next line of input. This will print out something like "/not-a-file: No such file or directory"

weird input: Note that "<" (or ">") may be followed by zero, one, or multiple words before ">" ("<") or end of line. You can handle this any that seems reasonable to you (e.g.: don't redirect; print error message; redirect to/from one of the candidates) and set status to anything you want, as long as you don't crash.

Your code may be tested to see that it handles the cases like the following ones without crashing:

```
ls > a > b > >
cat < a b c < d
```

"Leaking" file descriptors

If you're not careful you'll end up leaving file descriptors open in the parent process. An easy way to check is to see what file descriptors are being returned, in the debugger or via printfs - if you're closing everything properly you'll see the same ones each time, while if you're leaking them they'll keep counting upwards.

Step 6: Pipes

Limitations:

- it's OK if you don't handle more than 4 pipeline stages.
- You don't need to be able to pipe the output of built-in commands like `pwd`.

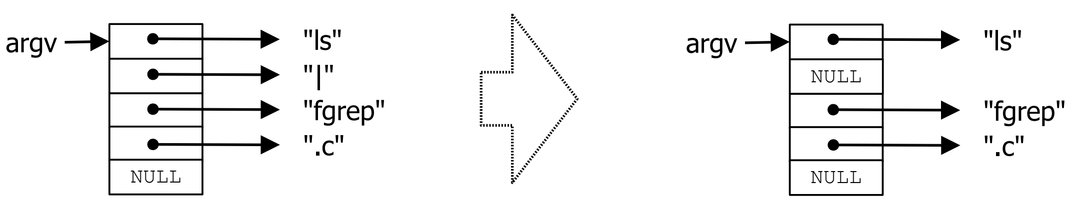
To implement pipes you'll need to use the `pipe` and `dup2` system calls.

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

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, and you'll need to close all of the copies that aren't being used as input and/or output in the appropriate child processes.
Thus for the `ls|grep` pipeline we'll need to do the following operations:

- 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)`

To do this, 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.

TEST IT:

```
ls | grep M
```

Things to beware of:

- Close all file descriptors that need closing - if you don't, commands like `ls|cat` will hang.
- Make sure you keep your original standard input and output in the parents, or copies of them. If you're calling `open("/dev/tty",...` you're doing it wrong, and will fail tests.
- Keep an array of all the child process IDs, and wait for each of them, one at time, using the logic from above.
- Make sure you handle bogus cases like "ls | | cat" – depending on the structure of your code, you may want to (a) print an error message ("syntax error") and ignore the line or (b) treat multiple pipe characters as a single one.

You should set status (for `$?`) to the status of the last command in the pipeline, but I'm not checking.

Background Information

Command Line Tokenizer

The C standard library has only the most rudimentary string handling functions, so applications that do any real string processing have to do it on their own. “Real” shells typically use standard compiler tools (lex and yacc or bison) to parse an input line into linked “token” objects; since this is a systems class, not a compiler class, we use some much simpler code in parser.c

Here’s the interface:

```
int parse(char *line,
          int   argc_max, char **argv,
          char *buf, int buf_len);
```

You pass it the following arguments:

- line – the line you want to parse
- argc_max, argv – array of argc_max character pointers to hold output
- buf, buf_len – buffer to copy tokens into

The return value is the actual number of tokens it put into the argv[] array.

To understand how it works, here’s what it does with the input line `"ls|cat"` - it copies each token into ‘buf’, and puts a pointer to tat copy into the argv[] array. The return value in this case would be 3:

```
        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.

The parser is a hack and is not guaranteed to be bug-free, but your code will only be tested against cases where it works properly.

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
```