Deliverables

There are two deliverables for this assignment: C source files named `parser.c` and `shell.c`. Please commit these files to GitHub and submit via Gradescope. Only one member of each pair needs to do this, though do remember to add the other person's name to the submission. You are expected to practice pair programming, where both partners are working together on a shared screen to complete this assignment. See the explanation of pair programming in the syllabus for more-detailed guidelines.

Part 1: Parsing a Command-Line

All semester, we've used a command-line environment to perform various tasks: examine the contents of files, navigate between folders, compile and execute C programs, etc. In all these interactions, we've been dealing with two separate programs. Firstly, there is the *terminal* that deals with the low-level details of user interaction—it decodes the user's key presses as characters and draws these on the screen. The *shell* is the program that is responsible for interpreting the command typed in by the user and executing it (typically, by calling other programs). On this assignment, you will be implementing your own shell that mimics the behavior of the bash shell that runs on the Linux computers in Watson 132.

Your first task is to implement a command-line parser. Consider what you are specifying when you type in a command like the following into a shell:

```
gcc -Wall -g foo.c
```

You are instructing the system to launch the program named `gcc`; you are also supplying several command-line arguments (`-Wall, -g, foo.c`) that are to be passed to this program. The parser's job is to break up a user-typed command into its constituent parts, a process known as *tokenization.*

You should implement your parsing library in a file named `parser.c`. This library should implement the `parseCommand` function that is declared in the header file `parser.h`. This allows any other program that includes `parser.h` to call your `parseCommand` function. Your library file may contain additional helper functions (indeed, to avoid style deductions, it *should*), but these helper functions should not be exported, and therefore should not be added to `parser.h`. The `parseCommand` function has the following signature:

```
char** parseCommand(const char* cmdLine, int* background);
```

Given a string `cmdLine`, this function should return an array of strings containing the individual tokens from `cmdLine`, as illustrated in Figure 1. The `const` keyword in the declaration of `cmdLine` indicates that the string pointed to by `cmdLine` cannot be modified. You will need to dynamically allocate space for the `tokens` array holding pointers to each token, and also for the character arrays that hold the individual tokens. Each token should store one word from the command line, starting with a non-whitespace character and ending at the next whitespace (or the end of the command line). In performing the tokenization, `parseCommand` should treat all whitespace characters—space, tab, or newline—the same.

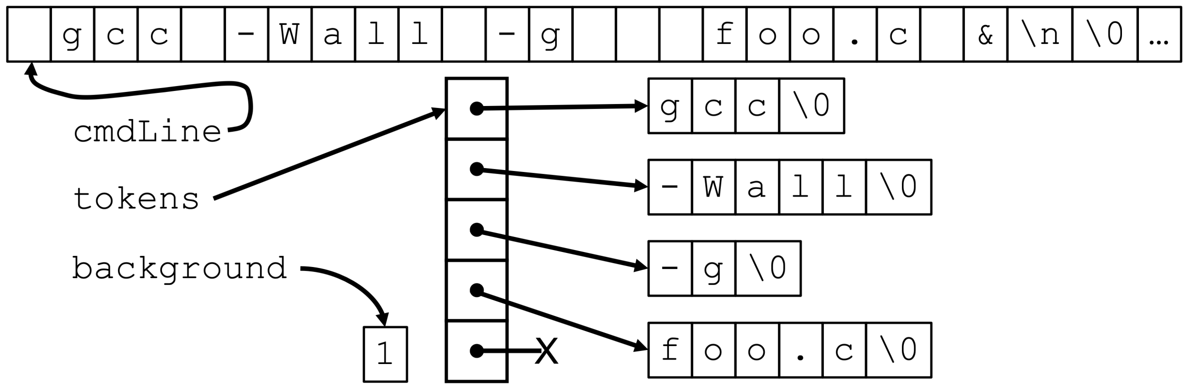

Figure 1: An array of strings containing the tokens extracted from a sample command line string. The `...` indicates `cmdLine` can contain more characters after the final `'\0'`, but we don't care what they are. Note that the fourth element in the `tokens` array contains a `NULL` pointer, and that each individual token contains no whitespace.

The returned array of strings should not contain any "slack" space. Say the string `cmdLine` contains four distinct tokens—then the returned array should contain exactly five `char*` elements, one for each of the tokens, plus one extra element at the end containing a `NULL` pointer, to indicate "end-of-array". Similarly, each token (string) must occupy exactly as many bytes as there are characters in that token, plus the extra byte for the end-of-string character `'\0'`. Here's another example—suppose the input string is:

```
"    ./schelling       small.txt        2  \n"
```

Then the function should allocate space for an array of four character pointers. The first `char*` should point a dynamically-allocated array of 12 characters, starting with the `'.'`, and ending with an end-of-string after the `'g'`. The second pointer should point to an array of 10 characters, that holds the string the `"small.txt"`, including the `'\0'`. The third should point to the string `"2"`, and the final `char*` pointer should be `NULL`. Figure 1 is also an example of what this should look like right before `parseCommand` returns.

The shell you write in Part 2 will support running programs in the background—here's what that means within the context of this assignment. The `cmdLine` argument passed to your `parseCommand` function may contain an ampersand character (&). If this is the case, then your function should set the value pointed to by the parameter `background` to 1; otherwise, it should set the value to 0. You may assume that an ampersand will only ever appear at the end of the string. An ampersand is *never* to be considered a token, or part of a token—it's simply a way for the user to express that the program should be run in the background. Thus, ampersands should never appear in the array returned by `parseCommand`. As a further example, the following input strings:

```
"cat hamster.txt &"
"cat hamster.txt&"
"    cat      hamster.txt          &  "
```

should all return the same array of tokens, and none of those tokens should contain ampersands. Further, the value pointed to by `background` should be set to 1 in all three cases.

Notes and Constraints

- You should use the built-in C function `isspace`, from the `ctype.h` library to identify whitespace characters.
- You may *not* use the `strtok` function from the `string.h` library (though you are welcome to compare against it for testing purposes). You are strongly encouraged to use any other built-in string manipulation routines like `strncpy`, `strlen`, and so on.
- It might be helpful to make a copy of the `cmdLine` that you can modify. `strdup` is useful for this, but remember to free any memory you use for scratch space when you're done with it.
- Perform incremental testing—there's no reason to put off testing until you've written the entire parser. Plan out your functional decomposition *before* you implement your solution, so that you can test your helper functions as soon as they are written. Add your testing code to `parserTest.c`, since `parser.c` is intended as a library and therefore should not contain a `main`.

- Don't try do do everything in one pass over the string; it's completely fine to run through `cmdLine` (or a copy of it) many times. For example, you can make subsequent operations simpler by first looking for an `'&'` and recording its position (or replacing it with a `'\0'` in the copy).
- Test your parser thoroughly. There are many corner cases—identify them carefully. Your parser should not crash for any user input! Don't forget to test the return value of `malloc`. Your parser should exit with return code 1 in the event of memory allocation failures.
- Your `parseCommand` function should not cause any unnecessary output to be printed to the screen.
- You can use CNTRL-C to kill your program if it gets stuck in an infinite loop.

Part 2: Building a Shell

Your second task is to use the parser to complete the implementation of your very own shell in a file named `shell.c`. When this program is run, it should execute the following algorithm

1. Print a prompt and wait for the user to type in a command.
2. Read in the command string entered by the user.
3. Tokenize the command line string.
4. If the command (i.e., the first token in the parsed list) is not `exit`, then fork a child process to execute the command and wait for it to finish. (Unless the command is to be run in the background, in which case the shell does *not* wait.)
5. Repeat until the user enters the command `exit` to terminate the shell program.

Further details about each of these steps can be found below.

Reading and Tokenizing User Input

- Print the following prompt to the screen indicating that your shell is ready to accept input, and wait for the user to type in input. Note that there is a single space after the prompt (i.e., after the `>` character).

```
catshell> 
```

- Use the `fgets` function rather than `scanf`/`fscanf` to read in the user entered command from the terminal. The `scanf` function stops reading in a string once it encounters a whitespace character; this is inconvenient when you are writing a shell since the user will often type in a command string that consists of several space-separated tokens. The `fgets` function is much better suited to this purpose. Refer to the manual page for more information; you can access the manual with the command `man fgets`.
- You may assume that the command typed in by the user will be at most 1000 characters in length. `#define` a constant with this value, instead of hardcoding it into your program.
- Tokenize the string using your solution to Part 1. Debug that solution carefully, and check with Bryce or the tutors for help with subtle bugs.

Shell Features

Your shell should support the following features.

Running commands in the foreground

For example, suppose the user types in the following command[1]:

```
catshell> sleep 2
```

Your shell process should fork a child that executes the `sleep` command (by calling `execvp`) and wait for that child to exit before proceeding. You can accomplish this by calling `waitpid` in the parent (i.e., the shell) and passing in the process ID of the child process (i.e., the return value of the `fork` call).

Running commands in the background

For example, suppose the user types in the following command:

```
catshell> sleep 3 &
```

Your shell process should fork a child to execute `sleep`, but it should *not* wait for the child to exit in this case. Instead, after forking the child process, it should immediately return to step

[1]Note that `sleep` is a standard Unix/Linux utility command—it is a program that "does nothing" for the specified amount of time. For example, `sleep 5` causes execution to be suspended for 5 seconds.

1 in the algorithm—print out the prompt, and read in the next user command. The child process will execute concurrently while the parent process (i.e., the shell) processes other commands. Your shell should be able to run any number of processes in the background—thus, if you type the following commands in quick succession:

```
catshell> sleep 10 &
catshell> sleep 10 &
catshell> sleep 10 &
catshell> ps
```

then you should see a list of all three child `sleep` processes displayed (the Unix `ps` command lists currently running user processes).

Note, however, that if you completely forget about the children, you will end up with *zombie* processes (because the OS keeps exited processes around until their parents clean them up). It is therefore your shell's responsibility to *reap* the background processes when they exit. Rather than actually collecting return values from our background processes, we'll take the easy way out by telling the OS we want to *ignore* them (yes: zombie, reap and ignore are the technical terms). We'll register this behavior by including the following line near the top of `main`:

```
signal(SIGCHLD,SIG_IGN);
```

What this does is tell the operating system that when our process receives a `SIGCHLD` signal (which happens whenever a child process exits) we want to ignore the signal. This in turn lets OS clean up its data structures for the child process (which it was keeping around only in case we wanted to know what hapend to the child), so that child processes are automatically reaped. **You only need to register this behavior once**—thereafter all received `SIGCHLD` signals will be ignored.

Built-in `exit` command

Your shell should treat the command `exit` differently from any other user input. If the user enters any command string of which the first token is `exit`, your program should respond on its own, terminating the shell process by cleaning up and returning 0 from `main`.

Implementation Advice

- Write helper functions as you develop your code—there's not much point in writing your complete solution first, and *then* going back to add helper functions. This task has a lot of moving parts—designing and writing helper functions early will ensure that you're not overwhelmed.
- Perform incremental testing—as soon as you add one piece of functionality, thoroughly test your shell and make sure that it works. Here's my recommended implementation order:

    1. Write the main loop for printing out a prompt, getting user input, and parsing it using the library Part 1.
    2. Add support for the `exit` command.
    3. Add support for running commands in the foreground.
    4. Add support for running commands in the background.

- Test the return value of all system calls, including `fork` and `execvp`. You should call `exit(1)` for unrecoverable errors, but print out an error message first.
- Make sure that after a call to `execvp` you *always* have some error-handling code, in case the command the user requested doesn't work! Think about what error message to print in this case: what would `bash` do?
- Be careful to avoid memory leaks—recall that the parser allocates space on the heap to hold both pointers and the tokens themselves. It is your shell's job to free up this space after it has executed a command. Since a shell is a long-running process, memory leaks are particularly nasty! If unfreed mallocs keep piling up, your program will eventually run out of heap space and crash.
- You may want to add the statement `fflush(stdout)` immediately following any `printf` statements in your shell—this will ensure that any messages you print out are not "buffered" by the operating system and are printed out right away. This is particularly important when debugging segfaults.
- Your shell should not print any unnecessary output to to the screen.
- You can use CNTRL-C to kill your program if it gets stuck in an infinite loop.

Code Style Guidelines

Your code should abide by the guidelines outlined in the included *C Style Guide* document. The following (non-exhaustive) checklist may be handy when preparing your code for submission.

Checklist

- Correctness and Defensive Programming
  - ☐ The source files compile with no error messages when using the `-Wall` flag.
  - ☐ The program runs without generating any `valgrind` errors or `valgrind` warnings about memory leaks.
  - ☐ The program checks the return values of all calls to the library functions listed in the style guide.
- Documentation
  - ☐ A comment header at the top of each C file has a brief description of the program and the names of all group members.
  - ☐ Every function has its own comment header that briefly describes its purpose, parameters, and return values.
  - ☐ Comments inside functions describe the sub-tasks solved by the function, when appropriate.
- Readability
  - ☐ Variable and function names are meaningful.
  - ☐ Local variables are used to the maximum extent possible, without resorting to unnecessary global variables.
  - ☐ The code is indented neatly and consistently.
  - ☐ Excessively long lines of code are avoided, wrapping to the next line when necessary.
  - ☐ "Magic numbers" and other constants have been eliminated using the `#define` directive.
  - ☐ Boolean expressions are used in the simplest form possible.
- Structure and Parsimony

- ☐ The code is free of unnecessary complications and redundant fragments.
- ☐ The code displays evidence of thoughtful design, with an appropriate decomposition and helper functions.

Submission

Submit the files `parser.c` and `shell.c` by pushing to GitHub and uploading to Gradescope. Only one member of your team needs to upload to Gradescope, though you should remember to attach your partner's name to the submission. Note that not following the given specifications — misnaming your files, misnaming your functions, having your functions accept an incorrect number of parameters or parameters in the incorrect order, not returning the correct value in your functions, etc. — will result in significant autograder failures, so read the handout again carefully if this happens.