Welcome to CS134! In this class, assignments are designed to build on top of one another, culminating in a cohesive piece of software that you will be asked to integrate. The first component of this puzzle is to implement the **shell**. A shell describes a wrapper for a terminal, which will be fully implemented in the devices/drivers assignment.

This assignment has three primary goals: ① you will review C programming to re-orient yourselves with the syntax, the debugging tools, etc...; ② you will develop a front-end shell that will ultimately serve as the interface into the OS you will develop over this course; and ③ you will start to develop proficiency in Linux.

Collaboration Policy

This assignment is intended to be implemented as a *pair-programming* assignment! That means that you and your partner are working on the assignment **together**.

After the initial programming phase of the assignment, you and your partner will participate in a *code review* of another group's submission. Be sure to submit your code review on Gradescope!

Submission

You are asked to submit three deliverables for this assignment: ① the programming component, ② your code review for your partnered group, and ③ a reflection to the review of your code. In addition, in-class participation is required on **September 9th** to both give and receive feedback for full credit.

Getting Started

This assignment is designed to run on the course virtual machine (e.g., `ssh stum2025@itbdcv-lnx04p.campus.pomona.edu`). The stencil for assignments in this class assume particular compiler versions and processor features, so using the VM will make your life easier.

Download the source for this assignment to get started! You can find all of the tasks that you will need to implement by running "`grep "TODO: implement me!" -r .`" from the "shell-stencil" repository. In the repository, you will find a `Makefile` to help you build the various sources for each part of this assignment.

Deadlines

Section	Deadline	Submission URL
C Strings	Sept 2	(suggested)
Programming Component	Sept 7 (EOD)	HW 1 Programming
Code Review	Sept 9 (before class)	HW 1 Code Review
Code Review Reflection	Sept 9 (EOD)	HW1 Code Review Reflection
Post-Review Resubmission	Sept 13 (EOD)	HW1 Programming (late due date)

1 C Strings

When implementing an OS, all libraries need to be self-contained to avoid the infinite regress problem that can occur with dynamic linkers. That is, importing the string library into a file (e.g., `#include <string.h>`) is not allowed. For this reason, operating systems must implement their own versions of various useful libraries to remain functional. You **should not** import any additional libraries outside from the stencil in your solution!

Your Task

The first part of this assignment asks you to implement various useful functions from the `string` library so that it may later be imported for other functionality in the shell. These files is located in `util/string.{h,c}`. The header file, `string.h`, provides the declarations of the functions to be implemented with a description of their functionality. In particular, you are asked to implement each of:

- `char *strncat(char *dest, const char *src, long n)`
- `int strncmp(const char *s1, const char *s2, long n)`
- `long strlen(char *str)`
- `char *strstr(const char *haystack, const char *needle)`

Hints and Suggestions

Strings in C

Remember, strings in C are really expressed as arrays of `char`s. Every string ends with a NULL char to indicate the end of the string (e.g., `'\0'`). Given this, we can modify any string by performing array manipulation.

Man Pages

The header file provides a (minimal) description of each of these functions. In particular, it refers you to the *man pages* associated with each function. These can be accessed by running `man <num> <entry>`[1] to access the description of the inputs and return values of the function associated with "entry."

Testing

Don't forget to test your code! The Makefile has a target for "string" that will compile this file into an object file (using gcc with `-c` to produce a `.o` binary; these are binaries without main functions associated). To build a test, import the string header file into your test file (with a main function) and create test cases from there! When compiling, be sure to include the `bin/string.o` in compile line. See the `shell` and `readfile` tests as examples.

[1] Note, the "num" field is optional, but can help distinguish between entries with similar names in other subsystems of the Linux device.

Code Review Notes

When reviewing your peer group's code, be sure to design cases that test the functionality thoroughly. This means considering the edge cases:

- what happens if a parameter is much longer than n?
- shorter than n?
- has non-human readable characters?
- etc...?

For each of the tests, think about your inputs and outputs. What would you expect the output behavior to be for each of the inputs? You may want to refer to the built-in string library (`#include <string.h>`) to compare behaviors!

2 Parsing User Input

A key job of implementing a terminal shell is digesting user inputs and transforming them into meaningful terminal tasks. In order to do so, the shell must manage an *input engine*. There are two key reasons for doing so: for one, doing so keeps ingesting of user input modular to the mechanism by which the user passes information (this will allow us to later integrate the `tty` device driver); additionally, this introduces the idea of kernel subsystems, which will be critical to the design of the body of the operating system.

The input subsystem maintains several dynamically sized objects through the duration of its execution. To make things easier, this system source includes `#include <stdlib.h>`. The C standard library has many functionalities, but the **only allowable functions** for this part are `malloc`, `realloc`, and `free`. Eventually, you will implement your own memory allocation system as part of the body of the OS.

Your Task

This part of the assignment asks you to implement various functions relevant to input string parsing and maintaining the input parsing subsystem. The header file for this task is located at `include/input.h` and the source is defined in `src/input.c`. Much like in the previous section, the header file describes the functionality of each function. In this section, hints for implementing these functions can be found in the source files. These hints provide implementation notes, error and corner case handling, etc... but is non-exhaustive! In particular, your task is to implement each of:

- `void userinput_init()`
- `void userinput_reset()`
- `void userinput_cleanup()`
- `long handle_user_input(const char *user_input, long strlen, char **command)`
- `long tokenize_input(char *str, long strlen, char ***tokens)`

Hints and Suggestions

Return Values in C

Recall from CS105, if we want to have some variable in a caller function modified by the callee, we often pass these values by reference. That is, in the caller function we may have a local variable `int var;` (living on the stack of the caller). The callee function can then modify this value if it has a reference to it as a parameter.

If there is a function `void fn(int *x)`, we can call `fn(&var);` to have the callee function using exactly the instance of `var` that we care about. This is often used as a mechanism to return several values to the caller and/or use the return value to notify the caller if there was an error during its execution.

Given this, pay close attention to how the parameters to `handle_user_input` and `tokenize_input` are described, which may dictate how you implement the body of these functions.

Function Roles

If it is difficult to imagine the role of these functions, you may want to look at how they are utilized in `test/{shell.c,readfile.c}`.

Testing Your Implementation

To test your implementation, you may want to produce some minimal examples as per the string library, or wait until you've implemented the next section and test your code using the `shell` or `readfile` executables (described in next section).

3 Execution from User Inputs

Once user inputs have been parsed, the next job of the shell is to trigger process execution. At this point, we have not yet implemented the notion of a process, so the best way for us to implement this behavior is to rely on the native host's (e.g., the VM) ability to execute the process. We can do so by heavily relying on the `execve` function. Much like the input parsing subsystem, execution will also work from a subsystem that requires initialization and cleanup at the beginning and end of the execution of the shell.

Your Task

While this part of the assignment asks you to implement the subsystem initialization and cleanup functions, the crux of this portion of the assignment is to implement the `execute_process` function. The header file for this task is located at `include/exec.h` and the source file is defined in `src/exec.c`. In particular, your task is to implement each of:

- `void exec_init()`
- `void exec_cleanup()`
- `pid_t execute_process(const char *command, char **argv)`

Hints and Suggestions

Executing a Process

It will be helpful to closely examine the documentation for the `execve` function call when implementing this part of the assignment. Think about what happens when `execve` succeeds versus fails – given this, what needs to happen around this call in order to achieve the desired functionality.

Note, the `execute_process` function is not very complex (+/- 15 lines of code), but can be tricky to debug. Think about various debugging strategies to ensure that you get the signals you need to implement the correct functionality.

Testing Your Implementation

To test your implementation of the execution (and input) subsystems, you can use the two provided implementations of the shell. In the `test` directory, you will find a `shell.c` and `readfile.c` file. These files behave similarly in that, assuming correct implementation of the subsystems, they should both implement a functional shell.

In the `shell.c` implementation, the shell is interactive. Upon running this implementation, the user is prompted to input a command that is ingested by the shell to be executed as a new process. By contrast, the `readfile.c` implementation takes an additional command line parameter, which is the path to a shell file to execute. Here, you can include several commands (separated by newline) to be executed by the shell.

Compile these shells by running `make clean shell readfile`. This will produce new executables in the `bin` directory that can be run either via:

- `./bin/shell` or
- `./bin/readfile <path/to/input.sh>`

See the next section for inspiration of various tests to run as part of your code review for both the input and execution subsystems.

4 Using Your Shell

The goal of this section is to provide an intuition of some of the test cases that you should implement in your code review, as well as building your familiarity with the Unix shell[2]! Once you have successfully implemented your input and execution subsystems, you should test the following functionalities of the shell:

- print effective userid
- update user's authentication tokens
- exit a login shell
- print name of current/working directory
- list directory contents
- change the current directory to dir; if dir is not supplied, the value of the HOME shell variable is the default
- make/remove directories
- copy files and directories
- move (rename) files
- remove files or directories
- change file mode bits
- concatenate files and print on the standard output
- opposite of more
- show who is logged on and what they are doing
- show a listing of last logged in users
- using pipes to combine functionality
- controlling process functionality

Not all of these functions are expected to work, so be sure to reason about the expected inputs and outputs of each function before making observations about the true observed behaviors. As before, be sure to test with well formatted inputs, poorly formatted inputs, incorrect inputs, etc... Note, when running these utilities you will need to use the absolute path to the binary where these utilities are stored rather than their alias (as in the native shell). For example, to run "`ls`" you will need to run "`/bin/ls`."

You may want to test these command-line utilities in the native shell (e.g., on the VM without your implementation) to sanity check and learn their behaviors!

[2]This is largely based on the The Unix Tutorial from UC Berkeley.

5 Reflection

As part of your submission to the programming part of this assignment, please include a file called "reflection.txt" that answers the following:

1. How long did you spend on this assignment?
2. What was your main takeaway from this assignment? (1-2 sentences)
3. What questions do you still have?

6 (Optional) Extensions

As you will have observed, the provided shell implementation provides the basic functionality for a usable shell. Unfortunately, this shell is significantly less functional than the deployed shells in Linux-based platforms in (at least) two critical ways.

6.1 Recent Command Navigation

Normal Linux shells support using the "up" and "down" arrows to navigate through recently executed commands. The provided interactive shell in `shell.c` does not support this function.

6.2 Line Navigation

Within a command, Linux shells allow quick traversal to the beginning and end of the command for editing by using "`ctrl+A`" to jump to the beginning of the line and "`ctrl+E`" to jump to the end of the line. The provided interactive shell in `shell.c` does not support this function.

6.3 Putting Them Together!

Once you have implemented searching through recent commands and the ability to chain "`ctrl+`" in your shell, you can combine these functionalities to implement reverse search (e.g., "`ctrl+R`" and typing in a substring of the command you would like to re-execute)!