Introduction

In this assignment you will write **smallsh** your own shell in C. smallsh will implement a command line interface similar to well-known shells, such as bash. Your program will

1. Print an interactive input prompt
2. Parse command line input into semantic tokens
3. Implement parameter expansion
- Shell special parameters `$$`, `$?`, and `$!`
- Tilde (~) expansion
4. Implement two shell built-in commands: `exit` and `cd`
5. Execute non-built-in commands using the the appropriate `EXEC(3)` function.
- Implement redirection operators '<' and '>'
- Implement the '&' operator to run commands in the background
8. Implement custom behavior for `SIGINT` and `SIGTSTP` signals

Learning Outcomes

After successful completion of this assignment, you should be able to do the following

- Describe the Unix process API (Module 4, MLO 2)
- Write programs using the Unix process API (Module 4, MLO 3)
- Explain the concept of signals and their uses (Module 5, MLO 2)
- Write programs using the Unix API for signal handling (Module 5, MLO 3)
- Explain I/O redirection and write programs that can employ I/O redirection (Module 5, MLO 4)

Program Functionality

The following steps will be performed (if appropriate) in an infinite loop:

1. Input
2. Word splitting
3. Expansion
4. Parsing

5. Execution
6. Waiting

The loop is exited when the built-in `exit` command is executed. EOF on stdin will be interpreted as an implied `exit $?` command.

Whenever an explicitly mentioned error occurs, an informative message shall be printed to **stderr** and the value of the "$?" variable shall be set to a non-zero value. Further processing of the current command line shall stop and execution shall return to step 1.

All other errors and edge cases are unspecified.

1. Input

Managing background processes

Before printing a prompt message, smallsh shall check for any un-waited-for background processes in the same process group ID as smallsh, and print the following informative message to **stderr** for each:

- If **exited**: `"Child process %d done. Exit status %d.\n", <pid>, <exit status>`
- If **signaled**: `"Child process %d done. Signaled %d.\n", <pid>, <signal number>`

If a child process is **stopped**, smallsh shall send it the `SIGCONT` signal and print the following message to **stderr**: `"Child process %d stopped. Continuing.\n", <pid>` (See `KILL(2)`)

Any other child state changes (e.g. WIFCONTINUED) shall be ignored.

Note, the pid of a process is of type pid_t, and should be printed using the %jd format specifier. This also requires explicitly casting the value to intmax_t because of the way variadic arguments (used by printf functions) work. For example:

```
#include <stdint.h>

/* pid_t pid; */

fprintf(stderr, "Child process %jd done. Exit status %d\n",
(intmax_t) pid, status);
```

The "$?" and "$!" variables are not modified in this stage.

The prompt

Smallsh shall print a prompt to **stderr** by expanding the **PS1** parameter (see parameter expansion below).

Explanation:

“PS1” stands for “Prompt String 1”. There are three PSx parameters specified in POSIX:

- PS1 (default: “$” for users, “#” for root user): Printed before each new command line
- PS2 (default: “>”): Printed before each continued line of an incomplete command, such as when a newline is entered following an unmatched quote.
- PS4 (default: “+”): Printed before each command is executed, along with the command string, when the shell is in “trace” mode (for debugging).

PS3 is used by some shells for the built-in `select` command. The `select` command was not included in POSIX standards, so PS3 also was not included.

Smallsh does not support line continuation or tracing, and does not implement the `select` builtin, so we will only use **PS1** in this assignment.

Reading a line of input

After printing the command prompt, smallsh shall read a line of input from stdin. (See `GETLINE(3)`)

If reading is interrupted by a signal (see signal handling), a newline shall be printed, then a new command prompt shall be printed (including checking for background processes), and reading a line of input shall resume. (See `CLEARERR(3)`, and don’t forget to reset errno).

2. Word splitting

The line of input shall be split into words, delimited by the characters in the **IFS** environment variable, or `" \t\n"` (<space><tab><newline>), if **IFS** is unset (NULL).

A minimum of 512 words shall be supported.

Note: unset (NULL) is not the same as empty (“”). An empty IFS string is *valid*, and the entire command line would be interpreted as a single word.

2. Expansion

Each of the following tokens shall be expanded within each word:

- Any occurrence of `"~/"` **at the beginning of a word** shall be replaced with the value of the **HOME** environment variable. The "/" shall be retained. (see `GETENV(3)`)
- Any occurrence of `"$$"` within a word shall be replaced with the proccess ID of the smallsh process (see `GETPID(3)`).
- Any occurrence of `"$?"` within a word shall be replaced with the exit status of the last foreground command (see waiting).
- Any occurrence of `"$!"` within a word shall be replaced with the process ID of the most recent *background* process (see waiting).

If an expanded environment variable is unset, it shall be interpreted as an empty string (“”). This includes the **PS1** variable as described above.

The **$?** parameter shall default to 0 (“0”).

The **$!** parameter shall default to an empty string (“”) if no background process ID is available.

Expansion shall occur in the forward direction in a single pass, and expanded text shall not participate in further expansion token recognition (expansion is not recursive).

3. Parsing

The words are parsed syntactically into tokens in the following order. Tokens recognized in a previous step do not take part in further parsing.

1. The first occurrence of the word “#” and any additional words following it shall be ignored as a comment.
2. If the last word is “&”, it shall indicate that the command is to be run in the background.
3. If the last word is immediately preceded by the word "<", it shall be interpreted as the filename operand of the input redirection operator.

4. If the last word is immediately preceded by the word ">", it shall be interpreted as the filename operand of the output redirection operator.

Steps 3 and 4 may occur in either order (output redirection may occur before input redirection).

All words not listed above shall be interpreted as regular words and form the command to be executed and its arguments. The tokens listed above shall **not** be included as command arguments.

A valid command line may be either:

```
[command] [arguments...] [> outfile] [< infile] [&] [# [comment...]]
```

```
[command] [arguments...] [< infile] [> outfile] [&] [# [comment...]]
```

If "<", ">", and "&" appear outside of the end-of-line context described above, they are treated as regular arguments:

```
$ echo test < > & test
```

```
test < > & test
```

5. Execution

If at this point no command word is present, smallsh shall silently return to step 1 and print a new prompt message. This shall not be an error, and “$?” shall not be modified.

Built-in commands

If the command to be executed is `exit` or `cd` the following built-in procedures shall be executed.

Note: The redirection and background operators mentioned in Step 4 are ignored by built-in commands.

exit

The `exit` built-in takes one argument. If not provided, the argument is implied to be the expansion of “$?”, the exit status of the last foreground command.

It shall be an error if more than one argument is provided or if an argument is provided that is not an integer.

Smallsh shall print "\nexit\n" to **stderr**, and then exit with the specified (or implied) value. All child processes in the same process group shall be sent a SIGINT signal before exiting (see `KILL(2)`). Smallsh does *not* need to wait on these child processes and may exit immediately.

(See, `EXIT(3)`)

cd

The `cd` built-in takes one argument. If not provided, the argument is implied to be the expansion of "~/", the value of the **HOME** environment variable.

If shall be an error if more than one argument is provided.

Smallsh shall change its own current working directory to the specified or implied path. It shall be an error if the operation fails.

(See `CHDIR(2)`)

Non-Built-in commands

Otherwise, the command and its arguments shall be executed in a new child process. If the command name does not include a "/", the command shall be searched for in the system's **PATH** environment variable (see `EXECVP(3)`).

If a call to `FORK(2)` fails, it shall be an error.

In the child:

- All signals shall be reset to their original dispositions when smallsh was invoked. Note: This is not the same as SIG_DFL! See `oldact` in `SIGACTION(2)`.
- If a filename was specified as the operand to the input ("<") redirection operator, the specified file shall be opened for reading on stdin. It shall be an error if the file cannot be opened for reading or does not already exist.
- If a filename was specified as the operand to the output (">") redirection operator, the specified file shall be opened for writing on stdout. If the file does not exist, it shall be created with permissions 0777. It shall be an error if the file cannot be opened (or created) for writing.

- If the child process fails to exec (such as if the specified command cannot be found), it shall be an error.
- When an error occurs in the child, the child shall immediately print an informative error message to **stderr** and exit with a non-zero exit status.

6. Waiting

Built-in commands skip this step.

If a non-built-in command was executed, and the “&” operator was not present, the smallsh parent process shall perform a blocking wait (see `WAITPID(2)`) on the foreground child process. The "$?" shell variable shall be set to the exit status of the waited-for command. If the waited-for command is terminated by a signal, the "$?" shell variable shall be set to value `128 + [n]` where `[n]` is the number of the signal that caused the child process to terminate.

If a child process is **stopped**, smallsh shall send it the `SIGCONT` signal and print the following message to **stderr**: `“Child process %d stopped. Continuing.\n”, <pid>` (See `KILL(2)`). The shell variable "$!" shall be updated to the pid of the child process, as if it had been a background command. Smallsh shall no longer perform a block wait on this process, and it will continue to run in the background.

Any other child state changes (e.g. WIFCONTINUED) shall be ignored.

Otherwise, the child process runs in the "background", and the parent smallsh process does *not* wait on it. Running in the background is the *default* behavior of a forked process! The "$!" value should be set to the pid of such a process.

Signal handling

Smallsh shall perform signal handling of the SIGINT and SIGTSTP signals as follows:

The SIGTSTP signal shall be ignored by smallsh.

The SIGINT signal shall be ignored (SIG_IGN) at all times except when reading a line of input in Step 1, during which time it shall be registered to a signal handler which does nothing.

Explanation:

SIGTSTP (CTRL-Z) normally causes a process to halt, which is undesirable. The smallsh process should not respond to this signal, so it sets its disposition to SIG_IGN.

The SIGINT (CTRL-C) signal normally causes a process to exit immediately, which is not desired. When delivered by a terminal, the terminal also clears the current line of input; because of this, we want to reprint the prompt for the restarted line of input. This is accomplished by registering a signal handler (which does nothing) to the SIGINT signal; when an interruptible system call (such as `read()`) is blocked and a signal with a custom handler arrives, the system call will fail with errno=EINTR (interrupted) after the signal handler returns. This allows us to escape the blocked read operation in order to reprint the prompt. If the disposition were set instead to SIG_IGN, the system call would *not* be interrupted, as with SIGTSTP above.

Additional hints and guidance

Strategic use of goto vs continue

The main body of the program should be an infinite loop. Several points during execution are cancelation points that cause the loop to immediately repeat (such as a syntax error). The continue keyword can be used to immediately begin the loop from the beginning. However, this will not work inside a nested loop. Instead, we recommend that you use strategically placed labels and goto statements to return to well-defined points in the program, as necessary.

Handling errors and edge cases

In a project of this size, it is imperative that you carefully handle errors emitted by standard library functions. Debugging a program that does no error checking of return values is incredibly time consuming because it can be very difficult to determine the genesis of a bug.

Any edge cases not mentioned in this specification should be interpreted reasonably, but ultimately are up to you to handle however you see fit.

The command prompt

Given the relative complexity of the other components of this project, the command prompt is a common source of confusion for students. I suggest that you use the `GETLINE(3)` function to safely grab an entire line of input at a time. The signal handlers mentioned above will successfully interrupt this call, which means you should check its return value and, if necessary, errno, before processing further. Make sure you declare your line pointer outside of the loop that calls getline so that you don't have a memory leak on each loop iteration.

```
char *line = NULL;
size_t n = 0;
for (;;) {
  /* ... */
  ssize_t line_length = getline(&line, &n, stdin); /*
Reallocates line */
  /* ... */
}
```

```
for (;;) {
  /* ... */
  char *line = NULL; /* Created and destroyed on each iteration
*/
  size_t n = 0;
  ssize_t line_length = getline(&line, &n, stdin); /* leaking
memory */
  /* ... */
}
```

Word splitting

I highly recommended that you simply use strtok for this task. This function accepts a string of multiple possible delimiters, so you can simply pass it the **IFS** environment variable (or " \t\n" if **IFS** is unset). Strtok modifies in-place, replacing all delimiter characters with null bytes '\0', effectively breaking one string into several. On each successive call, it returns a pointer to the beginning of the next piece of the parent string. At the end of the input, it returns NULL.

It's probably best that you make a copy of each of these words as you tokenize (using e.g. `strdup(3)`), because you will need to be able to reallocate during the expansion stage. Make sure to `free()` any of these duplicated strings before you "forget" about them (i.e. a pointer referring to a string goes out of scope). Put all of these pointers in a

big array of `char *[]`, or allocate (and reallocate) a `char **` and fill that if you're feeling fancy.

If you don't make a copy of each word, they'll all be packed together inside a single character array object, split with '\0' bytes by the strtok. You cannot individually reallocate them in that state, because they are not independent objects, but contained within a single large char array!

Expansion

I highly recommend writing a generic string search and replace function for this. I've provided the following video (https://www.youtube.com/watch?v=-3ty5W_6-IQ) tutorial on how to implement string search and replace (as well as an example of how not to implement one).

Parsing

This is relatively straightforward. Iterate over the wordlist forwards until reaching the end or a "#", then look backwards for a "&", and then two steps back looking for "<" and ">". Keep track of where the arguments end and the tokens begin so that you can easily strip them out of the list later before calling exec() in the child. Remember, the special tokens always appear at the end of a line of input, never in the middle.

Input and Output redirection

Perform all redirection inside the child process (after calling fork), so that you don't change the parent process's file descriptors. Keep in mind, `open()` will assign to the lowest available file descriptor, so simply closing a standard stream file descriptor and then opening a new file is a common approach to redirection. See `dup2(2)` for more information about how to associate a specific open file with a specific file descriptor, in a more general sense. Either approach should be sufficient.

Don't forget the `mode` argument to `open()` when using the `O_CREAT` flag! Very common mistake.

Memory Leaks

It's impossible to avoid "leaking" memory in the child, because allocated memory holds the command-line arguments that are passed to the exec() function. You will not be evaluated on memory leaks, but do pay attention to your allocated variables on each loop iteration. If each word is allocated as a dynamic character array, those should be freed and/or realloc'd on each loop. Remember, things declared in loops go in and out of scope on each iteration, so pointers declared inside a loop's scope will lose references to allocated memory between iterations; be careful to declare variables outside of the main loop if they need to retain their values.

Testing your program

Smallsh does not recognize fancy format specifiers used in our PS1 prompts, and the default prompt will look like:

```
$./smallsh
```

```
\n\[\]\u\[\]@\[\]\H\[\]:\[\]\w\n\[\]\$\[\]
```

This is completely fine, or you can run your shell with a temporarily modified PS1 variable, for testing purposes:

```
PS1="$ " ./smallsh
```

```
$
```