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 $!
- Generic parameters as ${parameter}

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 or when the end of input is reached. End of input will be interpreted as an implied **exit $?** command (i.e. smallsh exits with the status of the last foreground command as its own exit status).

Smallsh can be invoked with no arguments, in which case it reads commands from stdin, or with one argument, in which case the argument specifies the name of a file (script) to read commands from. These will be referred to as interactive and non-interactive mode, respectively.

In non-interactive mode, smallsh should open its file/script with the CLOEXEC flag, so that child processes do not inherit the open file descriptor.

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

This is the first step of your program, but one of the **last** things you will actually implement. You should skip this subsection, and come back to it later when you are working on implementing background processes.

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

In **interactive** mode, smallsh shall print a prompt to **stderr** by expanding the **PS1** parameter (see parameter expansion below). Otherwise nothing shall be printed.

Explanation:

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

- PS1 (default: "$" for users, "#" for root user **if unset**): Printed before each new command line
- PS2 (default: ">" **if unset**): Printed before each continued line of an incomplete command, such as when a newline is entered following an unmatched quote. **We don't use PS2 in this assignment**
- PS4 (default: "+" **if unset**): Printed before each command is executed, along with the command string, when the shell is in "trace" mode (for debugging). **We don't use PS3 in this assignment.**

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

Smallsh shall read a line of input from stdin, if in interactive mode, or from the specified script file, if in non-interactive mode.

See `GETLINE(3)`

If reading in interactive mode 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. It's not documented in GETLINE(3), but getline calls READ(2) internally, which can fail and set errno to EINTR.

2. Word splitting

The line of input shall be split into words, delimited by whitespace characters (ISSPACE(3)) including <newline>. If a backslash appears in the input, it is removed and the next character is included in the current word, even if it is a whitespace character.

If a '#' comment character is encountered at the beginning of a new word, that character and any characters following it will be removed from the input.

In other words, "Hello\ World" in the input becomes a single word "Hello World", while "Hello World" is parsed as two words: "Hello" and "World". This is not the primary purpose of this assignment, and sample code is provided to accomplish this task, below.

A minimum of 512 words shall be supported.

3. Expansion

Each of the following shall be recognized and expanded within each word:

- 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).
- Any occurrence of "`${parameter}`" within a word shall be replaced with the value of the corresponding environment variable named *parameter* (see GETENV(3))

In any situation where an expanded environment variable is unset (i.e. getenv returns NULL), it shall be expanded as an empty string (““). This includes PS1.

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

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

Expansion is not recursive, and is performed in a single forward pass through each word.

This is not the primary purpose of this assignment, and sample code is provided to accomplish this task, below.

4. Parsing

The words are parsed syntactically into tokens.

If the last word is "&", it is interpreted as the "background operator".

Any occurrence of the words ">", "<", or ">>" shall be interpreted as redirection operators (write, read, append).

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 invoke undefined behavior when used with built-in commands (they may be treated as regular arguments, ignored, etc, and will not be tested in this way).

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 exit with the specified (or implied) value. Smallsh does *not* need to wait on 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 **HOME** environment variable. (i.e. `cd` is equivalent to `cd $HOME`)

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:

- 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.

- 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)`.
- The list of words are scanned left-to-right for redirection operators ("<", ">", ">>"). If a redirection operator is found, the word following it shall be interpreted as a path to a file. If there is no following word, or if redirection fails for any reason, it shall be an error.
    - "<" - Open the specified file for reading on stdin.
    - ">" - Open the specified file for writing on stdout. If the file does not exist, it shall be created with permissions 0777. The file shall be truncated.
    - ">>" - Open the specified file for appending on stdout. If the file does not exist, it shall be created with permissions 0777.
- Redirection operators and their filename arguments, along with the "&" background operator shall be removed from the list of arguments prior to executing the non-built-in command.
- If the child process fails to exec (such as if the specified command cannot be found), it shall be an error.

6. Waiting

Built-in commands skip this step.

If a non-built-in command was executed, and the background 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, while waiting on a foreground child process, it 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 in interactive mode, 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.

In non-interactive mode, smallsh will not handle these signals specially.

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. The getline() function will exit with an error and set errno to EINTR, since it uses read() internally (this is not documented in the man page for getline).

This literally looks like:

```
void sigint_handler(int sig) {}
```

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, I 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. You want to avoid crashing whenever possible.

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

Create a list of words. Iterate over the input line with a pointer to the current character, and an index into the wordlist. Here's a very simple outline of this procedure:

```
(Example code moved to ED)
```

Expansion

You will need to search each word for a "$" character, then decide what to do based on whether the next character is a "?", "!", "$", or "{".

(Example code moved to ED)

Parsing

Iterate over the wordlist, copying **pointers to words** into a list of child arguments as you go. You do not need to copy the actual contents of strings, here, just pointers! If you hit an operator, do the appropriate redirection, and then continue, skipping over the operator and its operand when copying the word pointers.

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

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. You don't need to systematically free all allocated objects before smallsh exits.

Testing your program

Smallsh does not recognize fancy format specifiers used in our PS1 prompts, and the default prompt will look garbled, 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
$
```

What to turn in?

- You may only use C99 standard (-std=c99)
- Submit as many source files as you want.
- You **must** include a makefile with your submission. It does not need to be fancy, but it should build your project:

An example makefile is shown below:

```
smallsh: smallsh.c
  gcc -std=c99 -o smallsh smallsh.c
```

For a multi-source project, it might look like:

```
smallsh: smallsh.c parser.c
  gcc -std=c99 -o smallsh smallsh.c parser.c [etc.]
```

Grading Criteria

This assignment is graded out of 180 possible points. To receive any credit, the shell must be able to accept input from a file argument and run arbitrary non-built-in commands with several arguments.

The following rubric items will be tested:

- Removing comments (#) during word splitting - 5 points
- Variable Expansion (total of 40 points):
    - $$ - 5 points
    - $? - 7 points
    - $! - 8 points
    - ${parameter} - 15 points
    - Multiple parameters in multiple words - 5 points
- Built-in Commands (15 points):
    - exit - 5 points
    - cd - 10 points
- Redirection operators (total of 45 points)
    - input "<" - 15 points
    - output ">" - 10 points
    - append ">>" - 5 points
    - Multiple redirection operators in one command - 15 points
- Background command operator "&" (total of 45 points):
    - Runs in background - 10 points
    - Reports exit status correctly - 10 points
    - Reports signaled status correctly - 5 points
    - Sends SIGCONT to stopped **bg**-process - 10 points
    - Sends SIGCONT to stopped **fg**-process - 5 points
- Interactive mode (total of 30 points)
    - Prints PS1 prompt - 10 points
    - Correctly ignores SIGTSTP - 5 points
    - Correctly resets signals in child process - 5 points
    - Correctly ignores SIGINT when not blocked reading input - 5 points
    - Correctly responds to SIGINT when it interrupts input reading - 5 points