

In this assignment you will write **smallsh** your own shell in C. smallsh will implement a subset of features of well-known shells, such as bash. Your program will

1. Provide a prompt for running commands
2. Handle blank lines and comments, which are lines beginning with the `#` character
3. Provide expansion for the variable `$$`
4. Execute 3 commands `exit`, `cd`, and `status` via code built into the shell
5. Execute other commands by creating new processes using a function from the `exec` family of functions
6. Support input and output redirection
7. Support running commands in foreground and background processes
8. Implement custom handlers for 2 signals, `SIGINT` and `SIGTSTP`

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

1. The Command Prompt

Use the colon `:` symbol as a prompt for each command line.

The general syntax of a command line is:

```
command [arg1 arg2 ...] [< input_file] [> output_file] [&]
```

...where items in square brackets are optional.

- You can assume that a command is made up of words separated by spaces.
- The special symbols `<`, `>` and `&` are recognized, but they must be surrounded by spaces like other words.

- If the command is to be executed in the background, the last word must be `&`. If the `&` character appears anywhere else, just treat it as normal text.
- If standard input or output is to be redirected, the `>` or `<` words followed by a filename word must appear after all the arguments. Input redirection can appear before or after output redirection.
- Your shell does not need to support any quoting; so arguments with spaces inside them are not possible. We are also not implementing the pipe `|` operator.
- Your shell must support command lines with a maximum length of 2048 characters, and a maximum of 512 arguments.
- You do not need to do any error checking on the syntax of the command line.

2. Comments & Blank Lines

Your shell should allow blank lines and comments.

- Any line that begins with the `#` character is a comment line and should be ignored. Mid-line comments, such as the C-style `//`, will not be supported.
- A blank line (one without any commands) should also do nothing.
- Your shell should just re-prompt for another command when it receives either a blank line or a comment line.

3. Expansion of Variable `$$`

Your program must expand any instance of `$$` in a command into the process ID of the shell itself. Your shell does not otherwise perform variable expansion.

4. Built-in Commands

Your shell will support three built-in commands: `exit`, `cd`, and `status`. These three built-in commands are the only ones that your shell will handle itself - all others are simply passed on to a member of the `exec()` family of functions.

- You do not have to support input/output redirection for these built in commands
- These commands do not have to set any exit status.
- If the user tries to run one of these built-in commands in the background with the `&` option, ignore that option and run the command in the foreground anyway (i.e. don't display an error, just run the command in the foreground).

`exit`

The `exit` command exits your shell. It takes no arguments. When this command is run, your shell must kill any other processes or jobs that your shell has started before it terminates itself.

`cd`

The `cd` command changes the working directory of `smallsh`.

- By itself - with no arguments - it changes to the directory specified in the `HOME` environment variable
 - This is typically not the location where `smallsh` was executed from, unless your shell executable is located in the `HOME` directory, in which case these are the same.
- This command can also take one argument: the path of a directory to change to. Your `cd` command should support both absolute and relative paths.

`status`

The `status` command prints out either the exit status or the terminating signal of the last foreground process ran by your shell.

- If this command is run before any foreground command is run, then it should simply return the exit status 0.
- The three built-in shell commands do not count as foreground processes for the purposes of this built-in command - i.e., `status` should ignore built-in commands.

5. Executing Other Commands

Your shell will execute any commands other than the 3 built-in command by using `fork()`, `exec()` and `waitpid()`

- Whenever a non-built in command is received, the parent (i.e., `smallsh`) will fork off a child.
- The child will use a function from the `exec()` family of functions to run the command.
- Your shell should use the `PATH` variable to look for non-built in commands, and it should allow shell scripts to be executed
- If a command fails because the shell could not find the command to run, then the shell will print an error message and set the exit status to 1
- A child process must terminate after running a command (whether the command is successful or it fails).

6. Input & Output Redirection

You must do any input and/or output redirection using `dup2()`. The redirection must be done before using `exec()` to run the command.

- An input file redirected via stdin should be opened for reading only; if your shell cannot open the file for reading, it should print an error message and set the exit status to 1 (but don't exit the shell).
- Similarly, an output file redirected via stdout should be opened for writing only; it should be truncated if it already exists or created if it does not exist. If your shell cannot open the output file it should print an error message and set the exit status to 1 (but don't exit the shell).
- Both stdin and stdout for a command can be redirected at the same time (see example below).

7. Executing Commands in Foreground & Background

Foreground Commands

Any command without an `&` at the end must be run as a foreground command and the shell must wait for the completion of the command before prompting for the next command. For such commands, the parent shell does NOT return command line access and control to the user until the child terminates.

Background Commands

Any non built-in command with an `&` at the end must be run as a background command and the shell must not wait for such a command to complete. For such commands, the parent must return command line access and control to the user immediately after forking off the child.

- The shell will print the process id of a background process when it begins.
- When a background process terminates, a message showing the process id and exit status will be printed. This message must be printed just before the prompt for a new command is displayed.
- If the user doesn't redirect the standard input for a background command, then standard input should be redirected to `/dev/null`
- If the user doesn't redirect the standard output for a background command, then standard output should be redirected to `/dev/null`

8. Signals SIGINT & SIGTSTP

SIGINT

A CTRL-C command from the keyboard sends a SIGINT signal to the parent process and all children at the same time (this is a built-in part of Linux).

- Your shell, i.e., the parent process, must ignore SIGINT
- Any children running as background processes must ignore SIGINT
- A child running as a foreground process must terminate itself when it receives SIGINT
 - The parent must not attempt to terminate the foreground child process; instead the foreground child (if any) must terminate itself on receipt of this signal.
 - If a child foreground process is killed by a signal, the parent must immediately print out the number of the signal that killed it's foreground child process (see the example) before prompting the user for the next command.

SIGTSTP

A CTRL-Z command from the keyboard sends a SIGTSTP signal to your parent shell process and all children at the same time (this is a built-in part of Linux).

- A child, if any, running as a foreground process must ignore SIGTSTP.
- Any children running as background process must ignore SIGTSTP.
- When the parent process running the shell receives SIGTSTP
 - The shell must display an informative message (see below) immediately if it's sitting at the prompt, or immediately after any currently running foreground process has terminated
 - The shell then enters a state where subsequent commands can no longer be run in the background.
 - In this state, the & operator should simply be ignored, i.e., all such commands are run as if they were foreground processes.
- If the user sends SIGTSTP again, then your shell will
 - Display another informative message (see below) immediately after any currently running foreground process terminates
 - The shell then returns back to the normal condition where the & operator is once again honored for subsequent commands, allowing them to be executed in the background.
- See the example below for usage and the exact syntax which you must use for these two informative messages.

Sample Program Execution

Here is an example run using smallsh. Note that CTRL-C has no effect towards the bottom of the example, when it's used while sitting at the command prompt:

```
$ smallsh
: ls
junk  smallsh  smallsh.c
: ls > junk
: status
exit value 0
: cat junk
junk
smallsh
smallsh.c
: wc < junk > junk2
: wc < junk
      3      3      23
: test -f badfile
: status
exit value 1
: wc < badfile
cannot open badfile for input
: status
exit value 1
: badfile
badfile: no such file or directory
: sleep 5
^Cterminated by signal 2
: status &
terminated by signal 2
: sleep 15 &
background pid is 4923
: ps
  PID TTY          TIME CMD
 4923 pts/0      00:00:00 sleep
 4564 pts/0      00:00:03 bash
 4867 pts/0      00:01:32 smallsh
 4927 pts/0      00:00:00 ps
:
: # that was a blank command line, this is a comment line
:
background pid 4923 is done: exit value 0
```

```
: # the background sleep finally finished
: sleep 30 &
background pid is 4941
: kill -15 4941
background pid 4941 is done: terminated by signal 15
: pwd
/nfs/stak/users/chaudhrn/CS344/prog3
: cd
: pwd
/nfs/stak/users/chaudhrn
: cd CS344
: pwd
/nfs/stak/users/chaudhrn/CS344
: echo 4867
4867
: echo $$
4867
: ^C^Z
Entering foreground-only mode (& is now ignored)
: date
Mon Jan  2 11:24:33 PST 2017
: sleep 5 &
: date
Mon Jan  2 11:24:38 PST 2017
: ^Z
Exiting foreground-only mode
: date
Mon Jan  2 11:24:39 PST 2017
: sleep 5 &
background pid is 4963
: date
Mon Jan  2 11:24:39 PST 2017
: exit

$
```

Hints & Resources

1. The Command Prompt

Be sure you flush out the output buffers each time you print, as the text that you're outputting may not reach the screen until you do in this kind of interactive program. To do this, call `fflush()` immediately after each and every time you output text.

Consider defining a struct in which you can store all the different elements included in a command. Then as you parse a command, you can set the value of members of a variable of this struct type.

2. Comments & Blank Lines

This should be simple.

3. Expansion of Variable \$\$

Here are examples to illustrate the required behavior. Suppose the process ID of `smallsh` is 179. Then

- The string `foo$$$$` in the command is converted to `foo179179`
- The string `foo$$$` in the command is converted to `foo179$`

4. Built-in Commands

It is recommended that you program the built-in commands first, before tackling the commands that require `fork()`, `exec()` and `waitpid()`.

The built-in commands don't set the value of status. This means that however you are keeping track of the status, don't change it after the execution of a built-in command.

A process can use `chdir()`

[\(Links to an external site.\)](#)

to change its directory. To test the implementation of the `cd` command in `smallsh`, don't use `getenv("PWD")` because it will not give you the correct result. Instead, you can use the function `getcwd()`

[\(Links to an external site.\)](#)

. Here is why `getenv("PWD")` doesn't give you the correct result:

- `PWD` is an environment variable.
- As discussed in Module 4, Exploration: Environment "When a parent process forks a child process, the child process inherits the environment of its parent process."

- When you run smallsh from a bash shell, smallsh inherits the environment of this bash shell
- The value of `PWD` in the bash shell is set to the directory in which you are when you run the command to start smallsh
- smallsh inherits this value of `PWD`.
- When you change the directory in smallsh, it doesn't update the value of the environment variable `PWD`

5. Executing Other Commands

Note that if `exec()` is told to execute something that it cannot do, like run a program that doesn't exist, it will fail, and return the reason why. In this case, your shell should indicate to the user that a command could not be executed (which you know because `exec()` returned an error), and set the value retrieved by the built-in `status` command to 1.

Make sure that the child process that has had an `exec()` call fail terminates itself, or else it often loops back up to the top and tries to become a parent shell. This is easy to spot: if the output of the grading script seems to be repeating itself, then you've likely got a child process that didn't terminate after a failed `exec()`.

You can choose any function in the `exec()` family. However, we suggest that using either `execlp()` or `execvp()` will be simplest because of the following reasons

1. smallsh doesn't need to pass a new environment to the program. So the additional functionality provided by the `exec()` functions with names ending in `e` is not required.
2. One example of a command that smallsh needs to run is `ls` (the graders will try this command at the start of the testing). Running this command will be a lot easier using the `exec()` functions that search the `PATH` environment variable.

6. Input & Output Redirection

We recommend that the needed input/output redirection should be done in the child process.

Note that after using `dup2()` to set up the redirection, the redirection symbol and redirection destination/source are NOT passed into the `exec` command

- For example, if the command given is `ls > junk`, then you handle the redirection to "junk" with `dup2()` and then simply pass `ls` into `exec()`.

7. Executing Commands in Foreground & Background

Foreground Commands

For a foreground command, it is recommend to have the parent simply call `waitpid()` on the child, while it waits.

Background Commands

The shell should respect the input and output redirection operators for a command regardless of whether the command is to be run in the foreground or the background.

- This means that a background command should use `/dev/null` for input only when input redirection is not specified in the command.
- Similarly a background command should use `/dev/null` for output only when output redirection is not specified in the command.

Your parent shell will need to periodically check for the background child processes to complete, so that they can be cleaned up, as the shell continues to run and process commands.

- Consider storing the PIDs of non-completed background processes in an array. Then every time BEFORE returning access to the command line to the user, you can check the status of these processes using `waitpid(...NOHANG...)`.
- Alternatively, you may use a signal handler to immediately `wait()` for child processes that terminate, as opposed to periodically checking a list of started background processes

The time to print out when these background processes have completed is just BEFORE command line access and control are returned to the user, every time that happens.

8. Signals SIGINT & SIGTSTP

Reentrancy is important when we consider that signal handlers cause jumps in execution that cause problems with certain functions. Note that the `printf()` family of functions is NOT reentrant. In your signal handlers, when outputting text, you must use other output functions!

What to turn in?

- You can only use C for coding this assignment and you must use the gcc compiler.
- You can use C99 or GNU99 standard or the default standard used by the gcc installation on os1.
- Your assignment will be graded on os1.
- Submit a single zip file with all your code, which can be in as many different files as you want.
- This zip file must be named `youronid_program3.zip` where youronid should be replaced by your own ONID.
 - E.g., if chaudhrn was submitting the assignment, the file must be named `chaudhrn_program3.zip`.
- In the zip file, you must include a text file called `README.txt` that contains instructions on how to compile your code using gcc to create an executable file that must be named `smallsh`.
- Your zip file should not contain any extraneous files. In particular, make sure not to zip up the `__MACOSX` directories.
- When you resubmit a file in Canvas, Canvas can attach a suffix to the file, e.g., the file name may become `chaudhrn_program3-1.zip`. Don't worry about this name change as no points will be deducted because of this.

Caution

During the development of this program, take extra care to only do your work on os1, our class server, as your software will likely negatively impact whatever machine it runs on, especially before it is finished. If you cause trouble on one of the non-class, public servers, it could hurt your grade! If you are having trouble logging in to any of our EECS servers because of runaway processes, please use this page to kill off any programs running on your account that might be blocking your access:

[T.E.A.C.H. - The Engineering Accounts and Classes Homepage](#)


[Links to an external site.](#)

Grading Criteria

This assignment is worth 20% of your grade and there are 180 points available for it. 170 points are available in the test script, while the final 10 points will be based on your style, readability, and commenting. Comment well, often, and verbosely: we want to see that you are telling us WHY you are doing things, in addition to telling us WHAT you are doing.

Once the program is compiled, according to your specifications given in README.txt, your shell will be executed to run a few sample commands against (ls, status, exit, in that order). If the program does not successfully work on those commands, it will receive a zero. If it works, then the grading script will be run against it (as detailed below) for final grading. Points will be assigned according to the grading script running on our class server only.

Grading Method

Here is the grading script [p3testscript](#) 

Download p3testscript

. It is a bash script that starts the smallsh program and runs commands on smallsh's command line. Most of the commands run by the grading script are very similar to the commands shown in the section Sample Program Execution. You can open the script in a text editor. The comments in the script will show you the points for individual items. Use the script to prepare for your grade, as this is how it's being earned.

To run the script, place it in the same directory as your compiled shell, chmod it (`chmod +x ./p3testscript`) and run this command from a bash prompt:

```
$ ./p3testscript 2>&1
```

or

```
$ ./p3testscript 2>&1 | more
```

or

```
$ ./p3testscript > mytestresults 2>&1
```

Do not worry if the spacing, indentation, or look of the output of the script is different than when you run it interactively: that won't affect your grade. The script may add extra colons at the beginning of lines or do other weird things, like put output about terminating processes further down the script than you intended.

If your program does not work with the grading script, and you instead request that we grade your script by hand, we will apply a 15% reduction to your final score. So from the very beginning, make sure that you work with the grading script on our class server!