Assignment 6

cpe 357 Winter 2018

```
Some people claim that the UNIX learning curve is steep, but at least you
only have to climb it once.
```

— /usr/games/fortune

Due by 11:59:59pm, Thursday, March 15th.
For this assignment you may work with a partner[1]. Be sure both names appear in the README.

Program: The Minimally Useful SHell (`mush`)

This assignment requires you to write a simple shell. `mush` has nowhere near the functionality of a full-blown shell like `/bin/sh` or `/bin/csh`, but it is fairly powerful for its size and has most features that one would want in a shell, including:

- **Both interactive and batch processing.** If `mush` is run with no argument it reads commands from stdin until an end of file (`^D`) is typed. If `mush` is run with an argument, e.g. "`mush foofile`," it will read its commands from `foofile`.

  This may not be accomplished by duping the file to stdin. It is important to retain the original standard input for executed pipelines, otherwise a script starting with the line `cat` would proceed to cat the rest of the script.

- **Support for redirection.** `mush` supports redirection of standard in (`<`) and standard out (`>`) from or into files. `mush` also supports pipes (|) to connect the standard output of one command to the standard input of the following one.

- **A built-in `cd` command.** Recall that `cd` cannot be run as a child process because it would change the child's working directory, not the parent's.

- **Support for `SIGINT`.** When the interrupt character (^C) is typed, `mush` catches the resulting `SIGINT` and responds appropriately. That is, the shell doesn't die, but it should wait for any running children to terminate and reset itself to a sane state.

More Specifics

The above is the executive summary, but, as always, the Devil is in the details. These are some more specific requirements to guide your shell-building:

- In deference to the time of the quarter, the `mush` prompt is to be "`8-P `".

- The shell should only print its prompt when both stdin and stdout are ttys. The rationale for this is that if stdin isn't a tty, the shell is not being driven interactively. If stdout isn't a tty, the user will not see the prompts even if printed.

- A pipe (|) connects the standard output of one command to the standard input of the following one. For example, "`ls | sort`" makes the output of `ls` the input of `sort`. A series of commands separated by pipes is a pipeline.

[1] The same one as Asgn 5.

- `mush` must handle redirection as in `parseline`. Output files for redirection (those created with `>`) should be truncated to zero length if they exist.
- `mush` can assume that the built-in `cd` command will be the first and only command on the line. That is, `cd` will not be part of a pipeline[2]. `cd` requires an argument of a directory to which to change.
- `mush` must handle malformed or unexecutable commands gracefully. It should print an error message, clean up, and return to the prompt. This includes, but is not limited to:
  - non-existent commands or input files
  - ambiguous inputs or outputs. For example, in the pipeline "`ls | sort < foo`", the input to `sort` is specified to be two different things.
  - command-lines that exceed the required limits (see below).
- An error in any stage of a pipeline—non-existent file, bad command, etc.—kills the entire pipeline. That is, if a long pipeline starts with a bad redirect, abandon the pipeline.
- After a `SIGINT`, the shell should wait for the children to terminate before returning to the prompt.
- `mush` must support the following minimum size limits. That is, for each of these items, your shell needs to support at least that many, but may support more. If a limit is exceeded, `mush` should print an error message, consume the rest of the line, and re-prompt. The limits for your shell should be specified in your README.

Command line length:	at least 512 bytes
Commands in a pipeline:	must support at least 10
Arguments to a command:	must support at least 10

  The fact that these maxima exist will allow you to avoid the use of dynamic data structures. My first version of the shell did not have a single call to `malloc()` in it.

How to do it

Shell writing is a tricky business. A great deal of design consideration is necessary and not always in the places where one would think: probably 80% of my development time on the prototype was spent parsing the command line and getting the pieces into the right shape.

You have already built a version of the command-line parser as Asgn 5.

Making sure you have that under control before moving on to the process-launching and signal-handling parts of the project. To help with command-line cracking, the C library has a plethora of string processing routines available. `man string` for information on these.

Once you have the command line parsed, you need to open file-descriptors for the standard inputs and outputs of each future child. For file redirects, use `open(2)` to open or create the file. For pipes, create the pipe with `pipe(2)` and connect the ends appropriately. If there is no redirection, leave the child's stdin or stdout the same as the parent's.

The `cd` command is a special case; the shell executes it itself rather than spawning a child. Check for `cd` while parsing the command line.

Now, after parsing and determining that the command is not built-in, begin launching children. For each command, fork a child process. When the child process begins executing, it needs to dup

[2]If `cd` *does* appear in a pipeline you may either refuse to execute it, or do something "reasonable."

the appropriate file descriptors to its standard input and output (if necessary), close all other open file descriptors, and then exec its command.

Once the children are launched, the parent needs to wait for them all to terminate. Be sure to keep a list of their process ids so you can know when they have all terminated.

When all of the children have terminated, the shell process resets itself, flushes stdout and prints another prompt to do it again.

Tricks and Tools

There are many library routines and system calls that may help with implementing `mush`. Some of them are listed in Figure 1.

`fork()`	creates a new process that is an exact image of the current one

`execl()`
`execlp()`
`execle()`
`execv()`
`execvp()`
	known collectively as "the execs", this family of functions overwrites the current process with a new program. For this assignment, look closely at `execlp()` and `execvp()`.

`kill()`	sends a signal to a process
`wait()` `waitpid()`	waits for a child process to terminate
`pipe()`	returns an array of two connected file descriptors, one for reading and the other for writing
`isatty()`	for determining whether a particular file descriptor is associated with a tty
`feof()`	Determines whether a FILE * stream has reached its end of file. This is useful for telling when a stdio function's return value of EOF means end-of-file or interrupted system call.

`sscanf()`
`strchar()`
`index()`
`strtok()`
`strpbrk()`
etc.
	The string functions, defined in `string.h` and `strings.h`, are helpful for parsing strings

`isspace()` etc.	One of many functions defined in `ctype.h` for text processing. Very useful.

Figure 1: Some potentially useful system calls and library functions

Other useful things to remember in no particular order (read these and save time debugging):

- It can't be said too many times: stdout is buffered. Be sure to `fflush()` stdout after commands complete. There may be unwritten output still in the buffer.
- Remember that signal masks are inherited over a `fork()` and `exec()`. If you block `SIGINT` while launching your pipeline stages, remember to unblock it in each child before the `exec()`. If you forget, the children will be ever-after deaf to `SIGINT`.
- Keep a list (or at least a count) of child processes. This is the only way to know how many to wait for.

- Remember that `wait()` will get interrupted by the signal handler, so be sure to check its return value to be sure a child actually exited. If you don't, you risk losing count of your children.
- Setting up pipelines involves opening a lot of file descriptors. Be careful to remember to close these file descriptors when done. If you don't, the file descriptor table can fill up and prevent you from running any more commands.

  After each `fork()`, the child has a copy of all file descriptors open in the parent. All unneeded ones should be closed before the `exec()`.
- Be particularly careful to close the *parent's* copy of the write-end of a pipe. The process reading from the read-end will never get an end of file from the pipe until all open descriptors to the write end are closed. *If you forget to do this, pipelines like "*`ls | more`*" will hang forever.* (Even when the `ls` terminates, the parent still has an open descriptor to the pipe.)
- For what it's worth, my implementation—documentation and all—is approximately 1000 lines of C code. I expect mine is more verbose than yours will be. I have also implemented features not required here.

Coding Standards and Make

See the pages on coding standards and make on the cpe 357 class web page.

What to turn in

Submit via `handin` on hornet to the `asgn6` directory of the `pn-cs357` account:

- your well-documented source files.
- A makefile (called `Makefile`) that will build your program with "`make mush`".
- A README file that contains:
  - Your name(s). In addition to your names, please include your Cal Poly login names with it, in parentheses. E.g. `(pnico)`
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.

  The README file should be **plain text,** and should be named "README", all capitals with no extension.

Sample Runs

Below are some sample runs of `mush`. I will also place an executable version in `~pn-cs357/demos` so you can run it yourself.

```
% mush
8-P ls | grep o | sort | wc
      7       7      64
8-P ls Makefile mush > file1
8-P cat file1
```

```
Makefile
mush
8-P cat < file1 > file2
8-P cat file2
Makefile
mush
8-P cat > file2 | sort
cat: Ambiguous Output
8-P ls | more < file2
more: Ambiguous Input
8-P a | b | c | d | e | f | g | h | i | j | k
Pipeline too deep.
8-P a b c d e f g h i j k l m n o
Too many arguments.
8-P foo
foo: No such file or directory
8-P echo "hi" > foo
8-P cat foo
"hi"
8-P foo
foo: Permission denied
8-P ls -l foo
-rw-------    1 pnico    pnico           5 Mar  5 22:37 foo
8-P rm foo
8-P cd foo
foo: No such file or directory
8-P ls > foo
8-P cd foo
foo: Not a directory
8-P rm foo
8-P mkdir foo
8-P cd foo
8-P pwd
/home/pnico/CalPoly/Class/cpeX357/f10/Asgn/asgn6/Soln/foo
8-P
8-P ^C
8-P
8-P sleep 20
^C                                <--- interrupts the sleep
8-P
8-P ^D
%
%
% cat > commands
/bin/echo -n Hello,
echo world
^D
% mush commands
```

```
Hello,world
%
```