Assignment 6

cpe 357 Winter 2024

```
As soon as we started programming, we found to our surprise that it wasn't
as easy to get programs right as we had thought.  Debugging had to be
discovered.  I can remember the exact instant when I realized that a large
part of my life from then on was going to be spent in finding mistakes in
my own programs.
                -- Maurice Wilkes, designer of EDSAC, on programming, 1949
```

— /usr/games/fortune[1]

Due by 11:59:59pm, Friday, March 15th[2].
This assignment is to be done individually.

Program: The Minimally Useful SHell v.2.0 (`mush2`)

This assignment requires you to write a simple shell. `mush2` 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 `mush2` is run with no argument it reads commands from stdin until an end of file (`^D`) is typed. If `mush2` is run with an argument, e.g. "`mush2 foofile`," it will read its commands from `foofile`.

  This must 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.** `mush2` supports redirection of standard in (`<`) and standard out (`>`) from or into files. `mush2` also supports pipes (|) to connect the standard output of one command to the standard input of the following one.

- **Support for quoted strings.** `mush2` supports double quotes to allow arguments with embedded spaces.

- **Support for backslash escapes.** `mush2` supports the use of backslash ('\') to cause the following character to be taken literally.

- **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, `mush2` catches the resulting `SIGINT` and responds appropriately. `SIGINT` does not need to be forwarded to children—the tty driver will take care of that—but after the signal, the shell should not die. Rather it should wait for any running children to terminate and reset itself to a sane state.

[1] `fortune` actually has this one wrong. Wilkes did say this, but he did it in a lecture at The Computer Museum in Boston, September 21, 1979.

[2] Note: Be aware that last quarter our final was moved to the common final time on the Saturday after the end of classes (something over which I have no control). This could happen again, so plan accordingly

If the shell receives a `SIGINT` while reading input, it should abandon that command line.

After handling a `SIGINT`, an interactive shell should reset and re-prompt (if appropriate). A shell in batch mode will terminate. (Think of this as applying the *Principle of Least Astonishment.*)

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 `mush2` prompt is to be "`8-P `" (read it sideways).
- 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.
- `mush2` must handle redirection. Output files for redirection (those created with `>`) should be truncated to zero length if they exist. If they are created, rw permissions for all should be offered.
- `mush2` 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[3].

  `cd` takes either zero or one argument. If the argument is present, it is the name a directory to which to change. If the argument is absent, `cd` changes to the user's home directory. The user's home directory is either the value of the environment variable "HOME" (preferred) or can be determined by looking up the user's password entry (if "HOME" is not defined). If both fail, print "unable to determine home directory" to stderr and give up.
- `mush2` 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.
  - failure of any system calls (e.g. `chdir(2)`).
- `mush2` may not place any limits on the number of commands in a pipeline, the number of their arguments, or the length of a command line. (Luckily, the parser takes care of most of this.)
- After a `SIGINT`, the shell should wait for the children to terminate before returning to the prompt, if appropriate.

[3]If `cd` *does* appear in a pipeline you may either refuse to execute the pipeline, or do something "reasonable." Since the only built-in command is `cd`, and it has neither input nor output, it'd be harmless to execute it as part of a pipeline.

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.

Good news: I've done the parser for you. (See the next section.) Even so, be sure you know how to use it before moving on to the process-launching and signal-handling parts of the project. The subdir "parseline" includes an example.

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 using `dup2(2)`. 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` before launching any children.

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 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 you know how many children you have and count properly[4].

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

The Mush library

Parsing is a common problem for programs. Usually, the parsing is straightforward and can be done by something as simple as C's `strtok(3)` or Java's StringTokenizer and a few conditional statements, but sometimes the rules are more complicated. (Consider parsing Java, say.) Because the problem is so common, however, tools have been developed to help build parsers. This program uses two of them:

`lex(1)`	A program that generates tokenizers based on *regular expressions*. See the man page for lex for detailed explanations, but in general it reads input until it matches a token description. That token is returned to the caller. The function to call the scanner is called `yylex()`.
`yacc(1)`	A program that generates a parser from a set of grammatical rules. The left hand side of each rule consists of a symbol that can be made out of the series of symbols on the right hand side. Rules separated by vertical bars represent alternatives. `Yacc` stands for "Yet Another Compiler-Compiler."

The inputs to `lex` and `yacc` consist of rules for matching inputs and then actions to be performed when each input is matched. The actions have been removed from the figures, but you can see them in the source files: `lexer.l` and `parser.y`. The tokenizing rules for `mush2` are shown in Figure 1, and the grammar is shown in Figure 2.

[4]Probably good advice in real life, too.

Note: The `lex` input file, `lexer.l` is somewhat complicated by code necessary to make it portable to different versions of lex. The GNU version of `lex` is called `flex`. It is *mostly* compatable, but some things are different as can be seen in the file. The GNU version of `yacc` is called `bison` and is also mostly compatable.

If you're getting worried: it is not necessary to understand `lex` and `yacc` to do this assignment. This information is included so you can know what you're working with if you want. These are very powerful and useful tools.

To use the parser: Read a command line (probably using `readLongString()`) and pass it to the function `crack_pipeline()`. This function calls the parser and returns a `struct pipeline` as described in `mush.h` (and in Figure 3). The pipeline structure contains the original command line, a length, and an array of structs describing each stage of the pipeline. The `struct stage` contains the broken out arguments for the stage (in, appropriately, `argc` and `argv`) and well as filenames for redirection of the input or output of the stage. If these are `NULL`, the stage expects its input or output (as appropriate) to be the original stdin, original stdout, or a pipe depending on where it is in the pipeline. If it's a filename, the input or output should be that file. The parser itself checks to be sure that no inputs or outputs are multiply defined.

The full set of errors that the parser detects is shown in Table 1.

`E_NONE`	No error
`E_NULL`	Invalid null command
`E_EMPTY`	Empty command line (not an error, per se, but not a command either)
`E_BADIN`	Ambiguous input redirection
`E_BADOUT`	Ambiguous output redirection
`E_BADSTR`	Unterminated string
`E_PARSE`	Generic parse error

Table 1: Errors detected by the mush parser.

To use the library: I have built the parser and bundled it up into a library for you to use. The functions in the library are given in Table 2.

The library itself is available on the CSL machines in `~pn-cs357/Given/Mush/libmush`. The 32-bit version is in the `lib` subdirectory, and the 64-bit version is in `lib64`. The header file, `mush.h`, is in the `include` subdirectory.

To use such a library file, lib*name*.a, you can do one of two things. First, you can simply include it on the link line like any other object file:

```
% gcc -o prog prog.o thing.o libname.a
```

Second, you can use the compiler's library finding mechanism. The `-L` option gives a directory in which to look for libraries and the `-l`*name* flag tells it to include the archive file `lib`*name*`.a`:

```
% gcc -o prog prog.o thing.o -L. -lname
```

Compile: gcc -g -Wall -I ~pn-cs357/Given/Mush/include -o foo.o foo.c

Link: gcc -g -Wall -L ~pn-cs357/Given/Mush/lib64 -o prog foo.o -lmush

For the 64-bit version, you'd use `lib64`, or you can put both on the link line and let the linker choose.

Note: When linking a program, the linker resolves symbols (finds missing names) in the order in which it encounters them. This means that **the order in which libraries appear on the link line matters.**

```
%{
 /* declarations, and code to get lex to read from a
  * string rather than stdin omitted
  */
%}

wordchar  ([^" \t\n<>|]|\\")
word {wordchar}+
quoted \"({wordchar}|[ \t])*\"?
string ({word}|{quoted})

%%

\%[^\n]*                { /* ignore comment */; }

[ \t]+                  { /* ignore whitespace*/; }

"\n"                    { lineno++; return T_newline; }

"<"                     { return T_from; }

">"                     { return T_into; }

"|"                     { return T_pipe; }

{string}                {
                          int token,ok;
                          ok=matchquotes(yytext);
                          yylval.v.string=cleancpystring(yytext);
                          if ( ok  ) {
                            token=T_string;
                          } else {
                            fprintf(stderr,"Unterminated string, line %d.\n",
                                    lineno);
                            token=T_badstring;
                          }
                          lineno += countlines(yylval.v.string);
                          return token;
                        }

%%
/* code to get lex to read from a string rather than stdin omitted */
```

Figure 1: The scanner's tokenizing rules.

```
%token T_string
%token T_pipe
%token T_into
%token T_from
%token T_newline
%token T_badstring

%%
cl:         commandlist T_newline { /* action omitted */}
        | commandlist             { /* action omitted */}
        ;
commandlist: command              { /* action omitted */}
        | commandlist T_pipe command
                                  { /* action omitted */}
        ;
command:   wordlist               { /* action omitted */}
wordlist: /* empty */             { /* action omitted */}
        | wordlist word           { /* action omitted */}
        | wordlist T_from word    { /* action omitted */}
        | wordlist T_into word    { /* action omitted */}
        ;
word:       T_string              { /* action omitted */}
        | T_badstring             { /* action omitted */}
        ;

%%
```

Figure 2: The parser's grammar.

```
typedef struct clstage *clstage;

struct clstage {
  char *inname;                /* input filename (or NULL for stdin) */
  char *outname;               /* output filename (NULL for stdout)  */
  int   argc;                  /* argc and argv for the child        */
  char **argv;                 /* Array for argv                     */
};

typedef struct pipeline {
  char          *cline;        /* the original command line  */
  int           length;        /* length of the pipeline     */
  struct clstage *stage;       /* descriptors for the stages */
} *pipeline;
```

Figure 3: The pipeline structures.

`char *readLongString(FILE *infile);`	reads an arbitrarily long string from the given FILE * up until it encounters a newline or EOF, removing the terminating newline, if any. On success, returns a pointer to the string in a newly-`malloc()`ed buffer. On error or EOF, returns `NULL`. (This includes being interrupted by a signal handler in which case `errno` will be `EINTR`.) The caller is responsible for freeing the allocated string.
`pipeline crack_pipeline(char *line);`	Take the given string and parse it into a `pipeline`. On success returns a pointer to the new pipeline. The `pipeline` retains a pointer to `line`, but `free_pipeline()` does not free it. On error, returns NULL.
`void print_pipeline(FILE *where, pipeline cl);`	Prints out the contents of the passed pipeline in an aesthetically pleasing way. Useful for debugging. `print_pipeline()` changes its behavior a little if the environment variable "`PARSELINE`" is set.
`void free_pipeline(pipeline cl);`	Frees the given command line, all its stages, and all associated strings.

Table 2: The contents of libmush.a

Tricks and Tools

There are many library routines and system calls that may help with implementing `mush2`. Some of them are listed in Figure 3.

`fork(2)`	creates a new process that is an exact image of the current one

`execl(3)`
`execlp(3)`
`execle(3)`
`execv(3)`
`execvp(3)`
`execve(2)`
	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()`.

`exit(3)` `_exit(2)`	terminates the calling process. The `_exit()` version does so "immediately" without flushing buffers or calling any `atexit()` handlers. This may be what you want after a failed `exec()`.
`wait(2)` `waitpid(2)`	waits for a child process to terminate
`pipe(2)`	returns an array of two connected file descriptors, one for reading and the other for writing
`isatty(3)`	for determining whether a particular file descriptor is associated with a tty
`feof(3)` `ferror()`	Determines whether a FILE * stream has reached its end of file or encountered an error. This is useful for telling when a stdio function's return value of EOF means end-of-file or interrupted system call.
`clearerr(3)`	Resets a stdio stream's EOF and error flags.
`char *getenv(3)`	Read the value of an environment variable.
`getuid(2)`	return the calling process's user id
`getpwuid(3)`	look up the password entry for the given uid (to get username)

Table 3: Some potentially useful system calls and library functions

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

- 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 track 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 750 lines of C code. I expect mine is considerably more verbose than yours will be. (The first version was about 450.) I have also implemented features not required here.
- To stop valgrind from complaining about the parser leaking memory, call `int yylex_destroy(void)` when you're done.

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 the CSL machines 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 mush2`".
- A README file that contains:
  - Your name(s). In addition to your name(s), 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.

Demo version of `mush2`

There are a few differences between this spec and the published demo version. I believe my version is a superset of the spec, but...

1. my `mush2` supports an option, `-p` that puts it in parse only mode. This makes debugging the parser part easier and seemed reasonable to leave in. If coupled with `-v`, it'll tell you what it did.
2. I got a little carried away, so my `mush2` supports a handful more builtin commands, including `exit`, `bye`, `cd`, `chdir`, `echo`, and `source`.

   These (should) behave appropriately in a pipeline.

Sample Runs

Below are some sample runs of `mush2`. I will also place an executable version in `~pn-cs357/demos` so you can run it yourself.

Note that in the runs below that "." is in my PATH.

```
% mush2
8-P ls | grep o | sort | wc
      7       7      64
8-P ls Makefile mush2 > file1
8-P cat file1
Makefile
mush2
8-P cat < file1 > file2
8-P cat file2
Makefile
mush2
8-P cat > file2 | sort
cat: ambiguous output
8-P ls | more < file2
more: ambiguous input
8-P echo a b c d e f g h i j
a b c d e f g h i j
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/cpe357/f15/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
% mush2 commands
Hello,world
%
```