CS 252: Systems Programming Fall 2024 Project 3: Shell Interpreter

Part 1: Monday, September 23 11:59pm

Goals

The goal is to build a shell interpreter combining behavior from common shells like Bash and csh. Some skeleton code is provided.

Deadlines

- Part 1: Monday, September 23 11:59pm
- Part 2: Wednesday, October 2 11:59pm
- Part 3: Friday, October 11 11:59pm
  All extra credit parts are due with the final submission.

Testing

Much of the shell will be graded using au

- `./testall partN` (where N = 1, 2, c
- `./testall` with no arguments runs

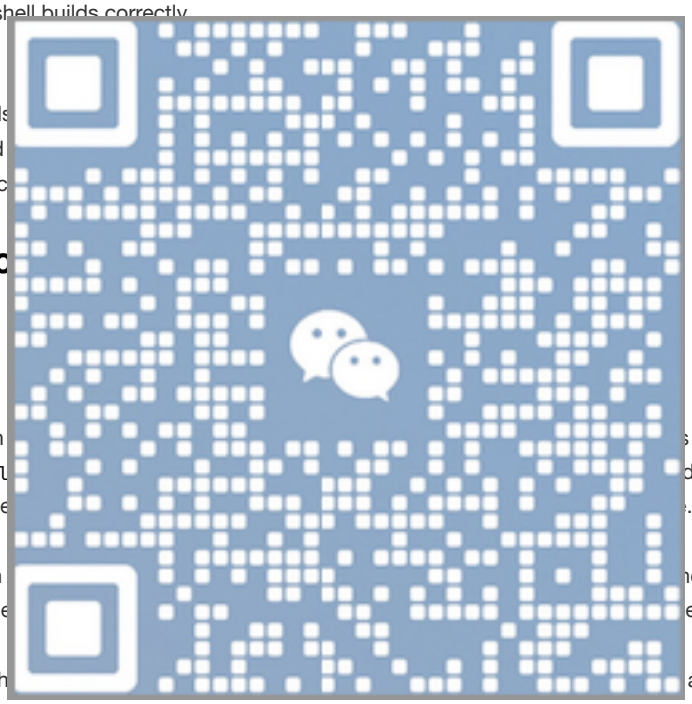
The Assignment - Part 1

1. **Login and Build**:
   - Log in to a CS department mach ctory, and run
     `git clone ~cs252/repos/$USE`
   - Build the shell by typing `make a`
2. **Parsing and Executing Commands**
   - **Scanner and Parser**:
     - Use Lex and Yacc to write a a structure representing a shell command. The `struct sin` e `struct command` represents a list of simple commands (which ca l error redirection.
     - Look through the `Makefile` ated from `shell.l` and `shell.y` and then compiled.
   - **Accepting more complex commands**:
     - Modify `shell.l` and `shell.y` to support the grammar:

       ```
       cmd [arg]* [ | cmd [arg]* ]* [ [> filename] [< filename] [2> filename]
       [ >& filename] [>> filename] [>>& filename] ]* [&]
       ```

     - Test with commands like `ls`, `ls -al`, `ls -al aaa bbb cc`, `ls -al aaa bbb cc > outfile`, etc.
3. **Executing Commands**
   - **Single command process creation and execution**:
     - For each single command, create a new process using `fork()` and call `execvp()` to execute the executable. If the command is not in the background, the shell waits for the last single command to finish using `waitpid()`. Refer to the man pages for function details. The file `ls_grep.c` is an example of process creation and redirection.
     - After this, commands like `ls -al` and `ls -al /etc &` should be executable.
   - **File redirection**:
     - If the command specifies IO redirection files, create them as necessary. Use `dup2()` to change file descriptors (0 for standard input, 1 for output, 2 for error).
     - Examples: `ls -al > out`, `cat -q cat 2> dog`, etc.
     - Notes:
       - `2>` redirects stderr to the specified file.

        - `>&` redirects both stdout and stderr to the specified file.
        - `>>` appends stdout to the specified file.
        - `>>&` appends both stdout and stderr to the specified file.
    - **Pipes**:
        - Use `pipe()` to create a pipe for inter-process communication. Redirect the output of one command to the input of the next using `dup2()`. See the example in `ls_grep.c`.
        - Commands like `ls -al | grep command` and `ls -al | grep command | grep command.o > out` should work.
    - **isatty()**:
        - When the shell uses a file as standard input, it should not print a prompt. Use `isatty()` to determine if the input is from a file or terminal. This is required for the automated tests to pass.
    - **Exit**:
        - Implement a special command `exit` that exits the shell when run. It should be parsed by the shell and cause the shell to exit without creating a new process.
4. **Submission**
    - By Monday, September 23 11:59pm:
        - Log in to a CS department machine.
        - Navigate to the `proj3` directory.
        - Run `make clean`.
        - Run `make` to check if the shell builds correctly.
        - Run `make submit_part1`.
    - Standard rules apply:
        - Follow the coding standards
        - The shell must compile and
        - Do not look at others' sourc



Part 2: Wednesday, Oc

Adding Features - Part 2

1. **ctrl-C**
    - Make the shell behave like bash s running, it should interrupt and usually terminate the command. If `ctrl` ded and a fresh prompt is printed. See `crtlc.c` for an example of dete.
2. **Zombie Elimination**
    - Set up a signal handler to catch ndler should call `waitpid()` to clean up zombie child processes. The she ess exits in the form “[PID] exited.”
3. **Quotes**
    - Add support for quotes in the sh a single argument. For example, `myshell> ls "command.cc Makefile"` should list `command.cc` and `Makefile` if they exist. The quotes should be removed before using the argument.
4. **Escaping**
    - Allow the escape character. Any character after `\` can be part of an argument, including special characters like quotation marks and ampersands. For example, `myshell> echo \"Hello between quotes\"` should print `"Hello between quotes"`, and `myshell> echo this is an ampersand \& this is an ampersand &` should print the correct text with the ampersand.
5. **Builtin Functions**
    - Implement the following builtin commands:
        - `printenv`: Prints the environment variables of the shell. The environment variables are stored in `char **environ`, a null-terminated array of strings. Refer to the `environ` man page.
        - `setenv A B`: Sets the environment variable `A` to value `B`.
        - `unsetenv A`: Un-sets environment variable `A`.
        - `source A`: Runs file `A` line-by-line as if it were typed into the shell by a user.
        - `cd A`: Changes the current directory to `A`. If no directory is specified, defaults to the home directory. Refer to the `chdir()` man page.
    - These built-ins should be used like other commands, including with redirection and piping.
6. **“.shellrc”**
    - When the shell starts, it should attempt to run “source.shellrc” equivalent.
7. **Submission**

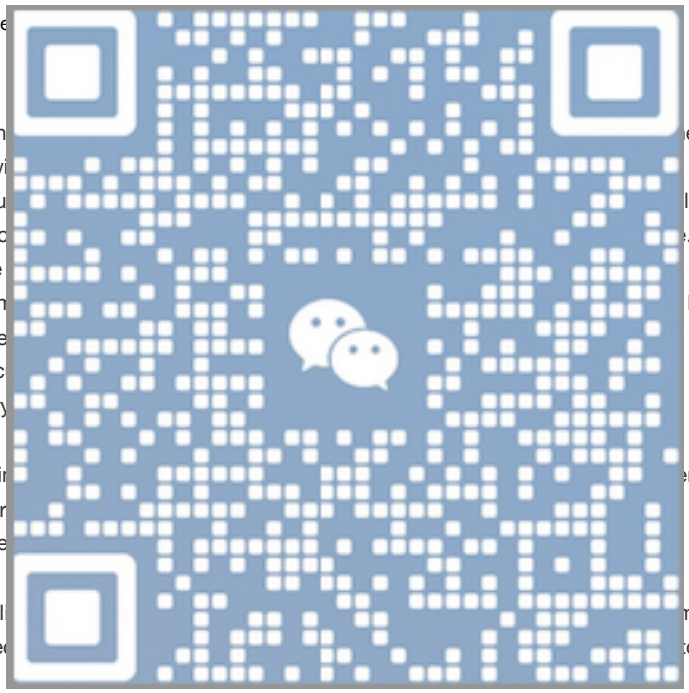
- By Wednesday, October 2 11:59pm:
  - Log in to a CS department machine.
  - Navigate to the `proj3` directory.
  - Run `make clean`.
  - Run `make` to check if the shell builds correctly.
  - Run `make submit_part2`.
- Standard rules apply as in Part 1.

Part 3: Friday, October 11 11:59pm

Moar Features - Part 3

1. **Subshells**
   - Implement subshells. Any argument of the form $(command and args) will be processed by a child shell process, and the output will be fed back into the original parent shell. For examp (expr 1 + 1) `becomes` echo 2 `, and` echo a b > dir; ls $(cat dir) `lists the contents of directories` a `and` b`.
   - Steps:
     - Extract the command between "$(" and ")" in `shell.l`.
     - Invoke the shell as a child p communicate (one to write the command to the child and c
     - Place the output obtained fr verse order. Do not use `popen()` or a temporary file for interproce
2. **Environment variable expansion**
   - Implement environment variable it will be expanded to the value corresponding to the variable va `$}` (PID of the shell process), `${?}` (return code of the last executed (run the last command again), `${_}` (last argument in the previous co an use `realpath()` to expand the relative path).
3. **Tilde expansion**
   - When `~` appears by itself or be If `~` appears before a word, the characters after `~` up to the firs in. Tilde expansion should not happen within double quotes.
4. **Wildcarding**
   - Implement wildcarding as follow
     - First, handle wildcarding on
       - Before inserting a new d character (* or?). If so, insert the file names that match the v
       - Use `opendir` and `rea` `xec` to find the entries that match the wildcard (convert from wildcards to regular expressions as needed, see `regular.cc` example).
       - Commands like `echo *`, `echo *.c`, `echo shell.?` should work.
     - Then make it work for any path (examples: `echo */*`, `echo */*/*`). Do not use the `glob()` call.
5. **Edit mode**
   - `tty_raw_mode.c` contains sample code to change the terminal's input from canonical to raw mode. Implement a full line editor in `read_line.c`.
   - Add the following code to `shell.l` after the `#include` lines:

```
%{
#include <string.h>
#include "y.tab.h"
////////////// Start added code ///////////
extern "C" char *read_line();
int mygetc(FILE *f) {
  static char *p;
  char ch;
  if (!isatty(0)) { // stdin is not a tty. Call real getc
    return getc(f);
  }
  // stdin is a tty. Call our read_line.
  if ((p == NULL) || (*p == 0)) {
    p = s; char *s = read_line();
  }
  ch = *p;
  p++;
  return ch;
}
#undef getc
#define getc(f) mygetc(f)
////////////// End added code
%}
%%
```



- Modify the `Makefile` by definin e line editor.
- In `read_line.c` , add the follow
  - Left arrow key: Move the cu line.
  - Right arrow key: Move the c .
  - Delete key (ctrl-D): Remove
  - Backspace key (ctrl-H): Ren left.
  - Home key (ctrl-A): Move the
  - End key (ctrl-E): Move the c
- Do not use the `readline` library

6. **History**
   - Implement a history list. Every ti nt the following editor commands:
     - Up arrow key: Shows the pr
     - Down arrow key: Shows the
7. **ctrl-R**
   - Implement the `ctrl-R` feature l mmand history by typing part of a command. Pressing “Enter” exe tches.
8. **Submission**
   - By Friday, October 11 11:59pm:
     - Add a `README` to the `proj3/` directory with:
       - Features that work as specified in the handout.
       - Features that do not work as specified in the handout.
       - Extra features implemented.
     - Log in to a CS department machine.
     - Navigate to the `proj3` directory.
     - Run `make clean` .
     - Run `make` to check if the shell builds correctly.
     - Run `make submit_part3` .
   - Standard rules apply as in Part 1.

Extra Credit

1. **Process Substitution (8 points)**
   - Implement process substitution. Use `mkdtemp()` to generate a temporary directory for a named pipe. Create a named pipe with `mkfifo()` within the temp directory. Connect the substituted command’s output to the named pipe and pass the pipe name to the other command. Clean up resources (fifo and directory) after the command is executed using `unlink()` and `rmdir()` functions.

2. **Path Completion (2 points)**
    - Implement path completion. When the key is typed, the editor attempts to expand the current word to a matching file similar to Bash.
3. **Aliases (2 points)**
    - Implement aliases. Allow users to define commands that encompass other commands (e.g., create shortcuts or add default arguments). For example, an alias for `ls` to always colorize the output, list entries by columns, and append an indicator.
4. **Jobs (8 points)**
    - Implement job control features like in Bash:
        - `jobs` : Display the status of jobs in the current session (refer to the `jobs` man page for operands).
        - `fg` : Run jobs in the foreground (refer to the `fg` man page for operands).
        - `bg` : Run jobs in the background (refer to the `bg` man page for operands).
        - `ctrl-Z` : Suspend a foreground process by sending it a `SIGTSTP` signal (implement a similar signal handler as for `ctrl-C`).

Grading

- Rough breakdown:
    - Part 1: 30 points (tested with `./testall part1`)
    - Part 2: 30 points (tested with `./testall part2`)
    - Part 3: 30 points (tested with `./testall part3`)
    - Grading of `readline` and `ctrl`
- Penalties:
    - -5 points for memory leaks.
    - -5 points for file descriptor leaks
      The parts build on each other, ar