Jacob Saltzman
28 February 2017
CS 4414: Operating Systems

Project 1: Shell++

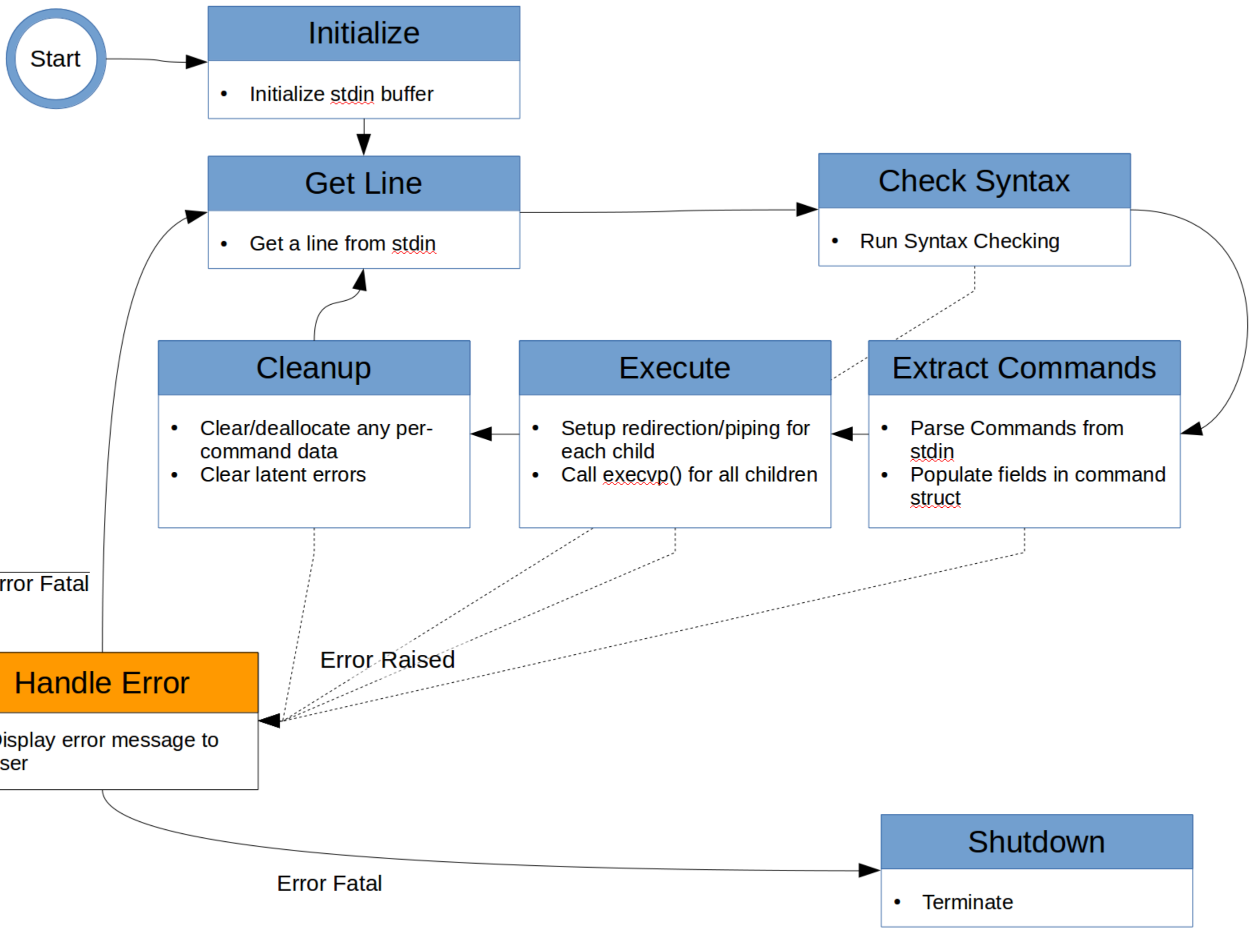
Purpose and Specification

The purpose of this project was to create a simple UNIX shell supporting pipes and redirects. The constraints were as follows:

- The shell must be capable of supporting unlimited pipes and redirects, up to 80 characters per line
- The shell must exit when the 'exit' command is supplied
- The shell must handle and display errors properly to the user
- The syntax must obey the specified lexical properties (these can be found in the assignment document.

Design

The code was written in pure C. A state-machine design pattern seemed natural for this project. The high-level state machine used for the lifetime of the shell is shown below:



The state machine design pattern involves the use of three important components, namely the global data available to all states; the current state; and the current error. These were created via the following typedefs:

```
/* State defining enum */
typedef enum {
      INITIALIZE,
      GETLINE,
      CHECK_SYNTAX,
      EXTRACT_CMDS,
      EXECUTE,
      HANDLE_ERROR,
      CLEANUP,
      SHUTDOWN
} sppstate_t;

/* Global Machine Data */
typedef struct {
      char* stdin_buffer;
            // Buffer populated by stdin
      cmd_t* cmds;
            // Array of commands (1 per token group, separated by pipes ‘|’)
      int cmdscount;
            // Size of the command array
} sdata_t;

/* Error Cluster */

typedef struct {
      int eflag; // 1 if error, 0 otherwise
      int ecode; // Error code if applicable
      char msg[STAT_MESSAGE_MAX_CHARS]; // Error message
      int xflag; // 1 if exit, 0 elsewise
      int fatal; // 1 if the error is fatal, 0 otherwise
} error_t;
```

The pseudocode for the operation of the state machine is as follows:

```
void main(){
      sppstate_t cstate = INITIALIZE;
      sdata_t sdata;
      error_t serror;
      while(cstate != SHUTDOWN) IterateStateMachine(cstate, &sdata, &serror);
}
sppstate_t IterateStateMachine(sppstate_t statein, sdata_t* sdata, error_t* serror)
{
      switch(statein) {
      case(INITIALIZE): InitStateMachine(sdata, serror); break;
      case(GETLINE): GetLine(sdata, serror); break;
      ... <etc> ...

      }
}
```

A major advantage of this approach is flexibility. Since the state machine data and errors are typedefs, it is trivially easy to add new data to the typedef, add or remove states, or change error handling routines without major refactoring. It is also easy to add or remove states without changing other states. In essence, the code can be highly decoupled.

Syntax Checking

The bulk of syntax checking is performed via <regex.h>. The following regex checks to ensure there are no dangling pipes or redirects, and ensures that only the legal characters [A-Z, a-z, 0-9, ‘.’, ‘|’, ‘>’,’<’, and ‘_’] are allowed:

```
#define ALLOWED_CHARS_REGEX (const char*) "^[A-Za-z0-9[:space:]\\.\\_\\-]+([<|>]?
      [[:space:]]*[A-Za-z0-9\\._]+[A-Za-z0-9[:space:]\\.\\_\\-]*)*$"
```

The webapp Regxr (http://regxr.com) was very helpful in crafting this regex. Syntax checking for piping and redirect rules was performed by scanning over each token group and ensuring that statements were well crafted (no double redirects, or redirects where pipes were consuming the corresponding stdin/stdout).

Parsing

Commands were parsed from the stdin buffer into an array of data structures specifically tailored for handling commands whose typedef is shown below. This array had n+1 elements for every n pipes.

```
      typedef struct {
            char* buffin; // raw text of command
            char** args;  // Args to be provided to execvp
            int argsct;   // Count of args

            int has_stdout_redir; // Flag if command has > redir
            int has_stdin_redir;  // Flag if command has < redir

            char* stdout_redir_fn; // Filename of > redir
            char* stdin_redir_fn;  // Filename of < redir
      } cmd_t;
```

Using a very simple state machine, the buffer was split on pipe character via strtok(), and each command group was parsed to fill in the appropriate fields of cmd_t.

Execution

For commands without pipes, execution was relatively straightforward. The parent process was forked via fork(). In the child process, dup2(), open(), and close() were used to open and/or close file redirection, and `execvp(sdata->args[0], sdata->args)` was called on the appropriate process.

For piped process, the following procedure was used (pseudocode below)

```
pipes_array[number_of_commands - 1][2] = create_pipes(number_of_commands - 1)

for i = 1:number_of_commands:
      pid[i] = fork()
      if command[i] is child:
            if command[i] is first commands in token group:
                  dup2(pipes_array[i][stdout], stdout)
                  close_all_other_pipe_descriptors(pipes_array)
                  set_stdin_redir()
            else if command[i] is last command in token group:
                  dup2(pipes_array[i-1][stdin], stdin)
                  close_all_other_pipe_descriptors(pipes_array)
                  set_stdout_redir()
            else:
                  dup2(pipes_array[i][stdin], stdin)
                  dup2(pipes_array[i-1][stdout], stdout)
                  close_all_other_pipe_descriptors(pipes_array)

            execvp(command[i].args[0], command[i].args)

      end if
end for

for i = 1:number_of_commands:
      wait(pid[i])
```

This method forks all commands of the parent, then sets the appropriate file descriptors for the pipes, then sets the appropriate file redirects for the first and last commands, if applicable. The parent then waits for all child processes to complete via wait().

Challenges Encountered

A major challenge to overcome was the behavior of piped commands. A key insight was that all unused pipes have to be closed for each child process even if they were being used by other processes. This is because calling fork() copies the memory contents of the parent into the child, creating an entirely separate set of file descriptors. In initial, broken versions of my code, these file descriptors were left dangling and child processes would get stuck waiting for data on open pipes.

A very obscure problem I encountered was a runtime free() error in <regex.h> caused by calling regexec() on strings with pipe character (e.g. “|”) without recompiling the regex via regcomp(). This was solved by calling regcomp() on every new input, instead of once during initialize.

Conclusion

A shell meeting the design specification was produced relatively quickly by adhering to good design patterns. Although the assignment stated that the program should be quote “less than 400 lines”, I believe that my program 740 lines including headers and comments, is significantly easier to follow and debug than a similar program that could be squirreled into 400 lines and be impossible to follow, debug, or extend.