Total Points: 100

COP4342 – UNIX Tools

Assignment #3: A Simple Unix Shell

Instructor: Dr. Robert Van Engelen   Teaching Assistant: Imran Chowdhury   Spring 2018

Description: The bash shell utility on UNIX and UNIX-based platforms is a command-line shell to access the resources of the operating system via built-in commands, by starting processes to run programs and utilities.

Assignment: In this programming assignment, you will implement a customized simplified version of a Unix Shell called “cssh”. There are no built-in programming structures or variables in cssh. The cssh shell program should be able to:

- Execute programs that are in the current directory or on the $PATH environment variable and that are compiled executables (object files) with permission ‘x’ set, and handle IO redirects for programs.
- Pass arguments that are separated by spaces and string arguments quoted with single ’-quotes, such as ’this is a single argument’, to commands.
- Exit on the ``exit’’ command.
- Change directory with the ``cd’’ command.

Command line options: In your “cssh”, it must support the following forms of basic commands:

./program *ARG1 ARG2* …	Execute program (with permission ‘x’), arguments are passed to it
ls *DIR*	Execute /bin/ls with *DIR* argument (**ls** is on **$PATH**)
exit	Exit from “cssh”
cd *DIR*	Change directory to DIR
< *FILE* and > *FILE*	Handle IO redirects with < and/or > (space between < or > and ***FILE***)

Exit code: the “cssh” should exit with EXIT_SUCCESS after the “exit” command (its arguments can be ignored).

Helpful reading: to understand the system calls that you will need to make in the C/C++ code for this assignment, which are fork(2), wait(2), execlp(3), execvp(3), signal(3), open(2), close(2), dup(2), dup2(2), please read this first: http://www.cs.fsu.edu/~engelen/courses/COP4342/unix_ch7.pdf

Sample C code: To read shell input from the keyboard, we use readline(3) which is installed on Linux and most Unix systems. It is not available in Visual studio out-of-the-box, so when developing code in Visual Studio use read(2) instead. Here is some sample code on how to read input with readline(3) and tokenize the string into string tokens separated by white space using strtok(3):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <readline/readline.h>
#include <readline/history.h>

static const char prompt[] = "cssh> ";
static const char sep[] = " \t\n\r"; // word separators
```

```
int main()
{
  int ac;       // arg count
  char *av[10]; // arg array of pointers to values
  while (1)
  {
    char *arg, *line;
    int i;
    // prompt then read line
    line = readline(prompt);
    if (line == NULL)
      break;
    // tokenize the line into av[]
    ac = 0;
    for (arg = strtok(line, sep); arg && ac < 10; arg = strtok(NULL, sep))
      av[ac++] = arg;
    // print the argument array, from argument 0 to ac (max 9)
    for (i = 0; i < ac; ++i)
      printf("arg[%d] = %s\n", i, av[i]);
    // exit?
    if (ac > 0 && strcmp(av[0], "exit") == 0)
      break;
    // line and av[] is no longer used, free the malloc-ed line
    free(line);
  }
  exit(EXIT_SUCCESS);
}
```

To compile this sample C code, save it to readline_test.c and then: **gcc -o readline_test readline_test.c -lreadline**

Here is a sample C code with read(2) instead, which is not so nice, because you won't have a history mechanism:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

static const char prompt[] = "cssh> ";
static const char sep[] = " \t\n\r"; // word separators

int main()
{
  int ac;       // arg count
  char *av[10]; // arg array of pointers to values
  int tty = open("/dev/tty", O_RDWR); // open tty for read/write
  if (tty == -1)
  {
    fprintf(stderr, "can't open /dev/tty\n");
    exit(EXIT_FAILURE);
  }
  while (1)
  {
    char *arg, line[256]; // buffer to hold line of input
    int i;
    // prompt then read line
    write(tty, prompt, sizeof(prompt) - 1);
    i = read(tty, line, sizeof(line));
    if (i <= 0)
      break;
    line[i] = '\0';
    // tokenize the line into av[]
    ac = 0;
    for (arg = strtok(line, sep); arg && ac < 10; arg = strtok(NULL, sep))
      av[ac++] = arg;
    // print the argument array, from argument 0 to ac (max 9)
    for (i = 0; i < ac; ++i)
      printf("arg[%d] = %s\n", i, av[i]);
    // exit?
    if (ac > 0 && strcmp(av[0], "exit") == 0)
      break;
  }
  exit(EXIT_SUCCESS);
}
```

Sample C code: Use the following example that is based on the **helpful reading** (see above), to execute a command with arguments:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>

int main()
{
  int ac;       // arg count
  char *av[10]; // arg array of pointers to values
  int pid;      // process id
  int status;   // child process exit status
  int w;
  void (*istat)(int), (*qstat)(int);
  int tty = open("/dev/tty", O_RDWR); // open tty for read/write;
  if (tty == -1)
  {
    fprintf(stderr, "can't open /dev/tty\n");
    exit(EXIT_FAILURE);
  }
  // set up a simple example command to execute with arguments
  ac = 3;
  av[0] = "ls";
  av[1] = "-a";
  av[2] = "-l";
  // fork this process, creating a copy of this running program
  if ((pid = fork()) == 0)
  {
    // this is the forked child process that is a copy of the running program
    dup2(tty, 0); // force stdin from tty
    dup2(tty, 1); // force stdout to tty
    dup2(tty, 2); // force stderr to tty
    close(tty);
    // last argument must be NULL for execvp()
    av[ac] = NULL;
    // execute program av[0] with arguments av[0]... replacing this program
    execvp(av[0], av);
    fprintf(stderr, "can't execute %s\n", av[0]);
    exit(EXIT_FAILURE);
  }
```

```
    close(tty);
    // disable interrupt (^C and kill -TERM) and kill -QUIT
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    // wait until forked child process terminated, get its exit status
    while ((w = wait(&status)) != pid && w != -1)
      continue;
    if (w == -1)
      status = -1;
    // restore interrupt and quit signals
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    // done with this example
    exit(EXIT_SUCCESS);
  }
```

Note that at most 8 arguments can be passed to a command (not counting IO redirects), since the `av[]` value right after the last `av[]` argument must be set to NULL for execvp(3) to work, where we allocated 10 array elements `av[0..9]`, and `av[0]` is set to the name of the command to execute. See the man page of execvp (man 3 execvp).

Implementing your shell: we suggest the following steps to implement cssh:

- First of all, you are not allowed to use system(3) directly anywhere in your code. If your code contains this call, you will receive zero points for this assignment.
- Implement cssh using the example code provided herein. The first version of your shell can be directly based on these examples and should be able to handle command execution. Test your implementation before implementing the ``cd'' command and IO redirects.
- To implement ``cd'', see the man page of chdir(2) for help.
- To implement ``> *FILE*'' redirect where there is always spacing between the ``>'' and *FILE*, check if the array `av[]` has a ``>'' string and if so open the file specified in `av[]` using open(2) for writing and use `dup2(fdout, 1)` with this open file descriptor `fdout` (see the man page of dup(2) and dup2(2)). Then modify the array `av[]` to remove these two entries from the array, shifting other array elements to ensure the array is not destroyed. Also reduce `ac` by 2. Do not forget to `close(fdout))` in the forked child and parent processes, because we dup-ed `fdout`.
- Likewise, to implement ``< *FILE*'' open the file specified for reading with open(2) and use `dup2(fdin, 0)` with this open file descriptor `fdin`. Adjust the array after extracting ``<'' and *FILE*. Do not forget to `close(fdin)` in the forked child and parent processes, because we dup-ed `fd`.
- You should now be able to handle IO redirects with the proper adjustment to `av[]` and `ac`, such as ``sort < somefile1.txt > somefile2.txt'' and in fact the IO redirects can be reordered as in a real shell such as ``sort > somefile2.txt < somefile1.txt'' and even ``< somefile1.txt > somefile2.txt sort''.
- Check that your cssh can execute commands such as **echo**, **ls** and **pwd**. Also check that you can execute a local program with **./program** and arguments.
- Remove strtok(3) and replace it with your own tokenizer function that tokenizes arguments that are spaced apart by space (0x20) and tabs (0x09). A quoted argument with a '-quote should be stored in `av[]` as one entry without the '-quotes. A simple trick is to change the line string by replacing white space by `\0` and by setting the `av[]` pointers to the location of each argument in the line string. Now, each array argument is a `\0`-terminated string. Do something similar with each '-quoted argument.

- Test if argument parsing works by checking you can execute **echo Hello ’world   !!’ Bye!** which has a quoted argument, such that the wide spacing in the argument is preserved when passed to **echo**.

Submission Requirements: A tar file named ‘yourCSusername_asg3.tar’ containing the following files:-

[1]. The C or C++ implementation of the above described *custom basic Unix Bash Shell*. Do not include object files or binaries.

Grading Policy: Submitted code must compile with the ‘-Wall -ansi -pedantic’ flags without any warning messages to conform to the ANSI ISO standard. Code that does not compile or generates errors (Segmentation Fault etc.) on execution will receive zero grade points. Also, when using system(3) in your code or calling a shell such as “sh” will result in zero grade points. Points distribution:-

[1]. Code Readability	(20 points)
[2]. Test Cases	(80 points)

Individual parts of the implementation will not be graded separately for correctness. There will be several cases to test the implementation logic as a whole. Also, keep in mind that your code will be tested on *linprog*. Students should test their code thoroughly on the *linprog* server before submission

Bonus to earn 2% Extra Credit for your final grade:

You can earn extra credit to count towards your final grade:

- Implement a pipe ``|’’ between two commands using pipe(2) to create two pipe descriptors, such that the first is dup-ed as 1 for the first command that is forked+execvp-ed and the second is dup-ed as 0 for the second command that is forked+execvp-ed. Note that you should execute two fork() and two execvp() and pass the argument array av[] to both but in a way that ensures that each command receives its portion of the arguments. IO redirects may not work yet, since your code must now extract two sets of these as well.
- After implementing the pipe(2) call, the dup2(2) calls, and changes to split av[] up to two parts to pass to the two commands, check that **cat somefile1.txt | sort** works as expected.
- Change your code to implement IO redirects for both commands. You may assume that the first command has no ``>’’ redirect and that the second command has no ``<’’ redirect.
- Check that **cat < somefile1.txt | sort > somefile2.txt** works as expected.
- You can expect spacing to surround ``|’’.

Miscellaneous:

Honor code policy will be strictly enforced. Write the code by yourself and protect your submission.

Key Concepts: *Processes, Process fork and wait, Argument passing, Interactive input, IO Redirects, Pipes, String Operations.*

UNIX API Calls: *readline*(3), *read*(2), *write*(2), *chdir(), fork(), wait(), waitpid(), dup(), dup2(), open(), close(), popen(), pclose(), pipe(), signal(), execvp*().