CSE-376 (Spring 2021) Homework Assignment #3

Version 2 (2021-03-24)

Due Tuesday, 4/13/21 @ 7:00 pm EST (19:00 ET)

This assignment is worth 20–25% of your grade, prorated to all assignments and exams we would have (if any).

1. Purpose

To become familiar with low-level Unix/POSIX system calls related to process and job control, file access, and IPC (pipes and redirection). You will write a mini-shell with basic operations (a small subset of Bash's functionality). Expected length of this C program is about 2,000 lines of code (not very long, but the code will be challenging, so start early). You will work alone on this assignment (no teams allowed).

Note: as we progress in this advanced course, I purposely give less detail in the assignments. I expect you to show maturity and independence, weigh various design factors, come up with a reasonable solution, implement it, and justify it in your README. In other words, you'll have a lot of flexibility in this assignment to choose how to implement features.

2. Task

Write a C program named "tish" (TIny SHell) that performs a subset of commands you're familiar with from other shells like GNU's Bash. You're welcome to study the code for bash, but the code you submit should be your own!

When you start your shell, you should be able to type commands such as this and see their output:

```
$ ./tish
tish> ls
<output from ls>
tish> ls -l
<verbose output from ls>
tish> exit
$
```

You will have to parse the command line and then use fork(2), clone(2), and/or execve(3) (or flavors of exec, such as execl, execle, etc.). Programs you run should output to stdout and stderr (errors); programs you run should take input from stdin by default. You will have to study the wait(2) system call and its variants, so your shell can collect and return the proper status codes of processes it forks, when they terminate, etc. Don't spend time writing a full parser in yacc/lex: use plain str* functions to do your work, such as strtok(3); you can

assume that any amount of whitespace delimits arguments for this assignment. You may use any system call (section 2 of the man pages) or library call (section 3 of the man pages) for this assignment, other than system(3).

Your tish should inherit its environment variables from its parent login shell (bash), so you can use variables like $PATH to figure out where programs are so you could execute them. Study the 3rd arg to main() called “envp”.

Note: 'exit' is not a program you'll execute, but a built-in special command that should exit(3) from your shell.

Another built-in command you should support is 'cd' to change directory using the chdir(2) system call; and 'pwd' via getcwd(3) to print the current working directory:

```
tish> pwd
/home/user
tish> ls
<shows files in /home/user>
tish> cd /tmp
tish> pwd
/tmp
tish> ls
<shows files in /tmp>
```

2.1. Job Control Support

Support the following special job-control syntax.

```
tish> CMD1 &
tish>
```

This "&" will place the program running in the background, so you get back to the shell and be able to run another command. The command:

```
tish> jobs
```

should list all background running jobs, their name, PID, job number, etc. (just like bash does) with their status (running or suspended). It should also print the exit status code of background jobs that just ended. The command

```
tish> fg 3
```

should make job number 3 in your list to go to the foreground (and resume execution if it is not running/stopped). The command

```
tish> bg 2
```

should cause suspended (stopped) program 2 to run in the background.

If you hit Control-C, a foreground program should receive a SIGINT signal (which may cause it to abort). If you hit Control-Z, a foreground program should be suspended and added to the list of jobs (i.e., you send it a SIGTSTP signal to suspend it; fg sends it a SIGCONT to resume running).

The command

```
tish> kill -N JOB
```

where N is a legitimate signal number (see the kill(2) system call) and JOB is a job number listed in the "jobs" command.

2.2. Redirection Support

Support redirection of stdin/stdout/stderr, and combinations of those. For example

```
tish> ls -l > newfile
```

should run "ls -l" and store the output in a new file called "newfile". Similarly:

```
tish> somecommand 2> err.log
```

should run "somecommand" and redirect its stderr to "err.log". And this is how you'd redirect stdin to a program.

```
tish> sort -nr < /tmp/inputfile
```

You'll have to learn how to manipulate file descriptors carefully using system calls such as open, close, read/write, dup/dup2, and more.

2.3. Variables and Echo Support

Support the 'echo' command via printf, with simple variables that can be assigned and viewed:

```
tish> echo $PATH
/bin:/usr/bin:/usr/local/bin
tish> FOO=bar
tish> echo $FOO
bar
```

For parsing purposes, assume that variables start with a '$' symbol, follow with the variable name as alphabetical letters, and ends with SPACE or EOL (End of Line).

Also support the special variable '$?' which should be the value of the exit code of the last command.

```
tish> ls
foo.c Makefile
tish> echo $?
0
tish> ls /blah
/blah: no such file or directory
tish> echo $?
1
```

2.4. Non-Interactive Support

Most shells can be run interactively as well as non-interactively. In non-interactive mode, you can put the shell commands in a plain file. For example, if you put this in a file called "foo.sh":

```
$ cat foo.sh
ls -l
echo hello world
```

Then you can run these commands in series as follows:

```
$ sh foo.sh
```

You can also make the file, called a shell script, executable, and then run it directly like any other program. For that, you need the file to start with a special character sequence "#!" (called a 'shebang') followed by the path of the shell:

```
$ cat foo.sh
#!/bin/sh
ls -l
echo hello world
$ chmod u+x foo.sh
$ ./foo.sh
```

Add support for tish to run non-interactively: this boils down to basically supporting the optional file argument, so this works

```
$ ./tish testscript
```

or

```
$ chmod u+x testscript
./testscript
```

You will also have to support a comment character '#' so if you see a line starting with '#' in the script, you should ignore it. The comment command could also be run non-interactively and "do nothing".

```
$ ./tish
tish> #this is some dummy text
tish>
```

3. Debugging

If you start tish with -d, it should display debugging info on stderr:

- Every command executed should say "RUNNING: <cmd>".
- When a command ends you should say "ENDED: "<cmd>" (ret=%d)" and show it's exit status.
- Optionally, add anything else to the debugging output (be creative as it'll help you debug the code).

If you build your shell correctly, you should be able to build and run your code from the first two homework assignments inside tish.

4. Test Scripts

You know the drill. Write and include as many or as few test scripts/programs to show us that your features work. The easier it'd be for us to test your code, the happier the grader(s) will be ...

5. Style and More

Aside from testing the proper functionality of your code, we will also evaluate the quality of your code. Be sure to use a consistent style, well documented, and break your code into separate functions and/or source files as it makes sense.

To be sure your code is very clean, it must compile with "gcc -Wall -Werror" without any errors or warnings! If the various sources you use require common definitions, then do not duplicate the definitions. Make use of C's code-sharing facilities.

You must include a README file with this and any assignment. The README file should describe what you did, what approach you took, results of any measurements you made, which files are included in your submission and what they are for, etc. Feel free to include any other information you think is helpful to us in this README; it can only help your grade. The code you write should be your own, but if you want to use any online code, you must clear it with me (note: github sources NOT allowed), and cite it both in your code and your README. Make sure to reasonably adhere to the requirements of the README.

6. Submission

You will need to submit all of your sources, headers, scripts, Makefiles, and README. Submission is accepted via GIT only! Do not submit regenerable files like binaries, *.o files, or any temp files — only true "source" files.

PLEASE test your submission before submitting it, by checking it out in a separate directory, compiling it cleanly, and testing it again. DO NOT make the common mistake of writing code until the very last minute, and

then trying to figure out how to use GIT and skipping the testing of what you submitted. You will lose valuable points if you do not get to submit on time or if your submission is incomplete!

(General GIT submission guidelines are available on the class website.)

7. Extra Credit

If you do any of the extra credit work, then your EC code must be wrapped in the following macro exactly. Please do not use any other ifdef other than EXTRA_CREDIT precisely as spelled; deviations will result in point deductions.

```
#ifdef EXTRA_CREDIT
     // EC code here
#else
     // base assignment code here
#endif
```

This extra credit is worth a total of 30-34 extra points (the main assignment is worth 100 points).

7.1. Time Counting (5 points)

Support time counting. If you start tish with -t, it should count how long each program ran and print stats when the program ends. The output should be something like:

```
$ tish -t
tish> du -sh /usr
4.3MB /usr
TIMES: real=23.7s user=12.1s sys=7.0s
```

Study time based functions and getrusage(2).

7.2. Pipes (15 points)

Support pipes, so you could run multiple commands such as

```
tish> find /usr/include | egrep -v /sys/ | sort | uniq | wc -l
```

Note that you should handle the situation where one types ^C to abort a program: the whole pipe sequence should end; also, if a program in the middle of a pipe fails, the entire pipe should be aborted. When in doubt, follow what regular shells like GNU bash do. Study the pipe(2) and dup/dup2 system calls.

7.3. File Globbing (10 points)

Support file "globbing" for extensions, such as

```
tish> ls *.jpg
```

the above should print all the file names that end with ".jpg". Only support *.[EXTENSION]. That is, you'll need to check to see if an argument starts with an '*', then use readdir(2) and getdents(2) as needed to read all files from the current directory, match them -- using strstr(3) -- and add them to list of args you pass to exec(3). In other words, your shell will be exec-ing a command that'll be as if you typed the full names of all the files on the command line one by one.

7.4. Early submission (4 points max)

To incentivize you to submit your code earlier than the deadline, we'll give you 2 points if you submit your assignment at least 24 hours before the official deadline; and 4 points if you submit at least 48 hours before. Note that we count the LAST git-push to your repo as the last time you submitted the assignment, even if you made a very small change.

Good luck.

8. Copyright Statement

9. ChangeLog

- V1: initial draft
- V2: TA review
-