System Programming - Shell Lab Report

2019 Spring, 2017-18570 Sungchan Yi

The first thing that came to my mind was to implement the signal handlers for SIGTSTP and SIGINT, which would be used to exit out of a program inside the shell. The implementation wasn't very hard, I just fetched the PID of the foreground process and sent a kill signal to the process group.

The pdf said that it is required to check the return value of every single system call, so I used a wrapper function that would check the return value for me. I wrote wrapper functions for `kill`, `fork`, with the first letter of each system call capitalized. (`Kill`, `Fork`)

Now the next thing to do was to implement the `eval` function. I referred to the code in the textbook, which helped me a lot. Then I figured I had to implement `builtin_cmd` function.

The `builtin_cmd` function should be able to handle `quit`, `jobs`, `bg` and `fg`. I just used `strcmp` to compare the input with the built in commands. And when it is a built in command, execute the command.

If it's not a built in command, call `Fork` and `execve`. Furthermore, check whether the process should run in the background and use `addjob` to add the job to joblist.

Next, it would be natural to implement `do_bgfg` for `bg`, `fg` commands. This was the hard part. First I would check it the arguments are correct. Then I would determine whether the job id or process id was given. Then depending on the command, change the state of the job accordingly and send a SIGCONT signal. While implementing the part for `fg` command, I figured that I need the `waitfg` function because after the `fg` command, I have to wait for the foreground job to finish. So the shell must wait while the `fgpid(jobs)` is equal to given `pid`.

Up to this part, pretty much of the functions were implemented. `sigchld_handler` was left. The handler was quite confusing to implement, so I referred to the textbook for more details. The textbook had info on WIFEXITED, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG function. So I used them to check the state of the child process.

While implementing the `eval` function, I also found out that I should check the return values of `sigprocmask` functions, to appropriately block the signals while forking a child process. So I also implemented wrappers for those mask functions, `Sigemptyset`, `Sigaddset`, `Sigprocmask`, and `Setpgid`.

With those functions implemented, I blocked signals before `Fork` and unblocked signals before `execve`, in the child process, unblocked signals after `addjob` was called.

The lab was pretty much done and I created a python script to check the test cases, and everything was OK.

Then I noticed that I used `printf` functions, which are not aync-signal-safe. So in the signal handlers, I tried to use the `write` function, the only async-signal-safe function for printing. But to use a format string in write, I had to write to a buffer and then use `write`. But to write to

a buffer, I needed `sprintf` function, which is not async-signal-safe. So I thought of using `for` loop to assign each character to a buffer, but that would violate the rule: "Keep your handlers simple as possible". So I disassembled the `tshref` and checked that they used `printf`. So I also stuck to `printf`.

This homework was very interesting because I am really fond of the shell in Linux. It was a great chance writing my own shell.

I asked TAs for how to handle the async-signal-safe functions, TAs suggested that I use the safe I/O functions. When I searched for SIO in the textbook, it used `#include "csapp.h"`. So I went online and downloaded `csapp.h`, `csapp.c` and used the SIO functions from the header file. The `sigchld_handler` function is now async-signal-safe.