Shell Lab

System Programming lab 4

자유전공학부 2018-14000

박진솔

Function Descriptions

eval

This function takes care of the overall shell. Since specific comments are within the tsh.c file, I will just explain the overall flow of the function. First, *parseline* is called, and the arguments are parsed into a form of a 2D array. For example, if the command was "*ls | sort*", then argv[0][0] contains ls, and argv[1][0] contains sort. Then, with *!builtin_cmd(argv)*, I implicitly called the *builtin_cmd* function that returns 0 if the given command is no built in. So going into this 'if' means that the given command is not built in.
If the command is not built in, make a *pipefd* array of size [rpipe-1][2]. Looping around a for loop for rpipec times, I first masked the SIGCHLD signal. This will later be unblocked, and this is to avoid the race condition where the child is reaped by *sigchld_handler* before the parent calls *addjob*.
Then call pipe(pipefd[i]). This enables the i-th pipe which is between the i-th and (i+1)-th command.
After forking, the parent and the child now shares the pipe.

If child, then there are a lot to do. These explanations below are comments from my tsh.c file, but I attach them here as well.

1. Pipe handling: redirecting inputs and outputs
   The important thing is that the read is redirected from the previous pipe.

```
/* ************************************PIPE*************************************
 * if not the last command, then redirect STDOUT to the pipe write end
 * if no the first command, then redirect STDIN to the pipe read end
 * in each case, close the unused end,
 * call dup2,
 * then close the used end to decrease refcnt
 * ************************************PIPE*************************************/
```

2. Redirection: handling the ">" character from command

```
/* ********************************REDIRECTION**********************************
 * first, find the position of ">" character by looping around 'while'
 * if the found position (dirsign below) is not 0, open a new file
 * the file name is the string after the ">" character
 * redirect STDOUT to the file
 * make the two last strings of the command as null
 *    - this is needed to later pass onto execvp propery
 * ********************************REDIRECTION**********************************/
```

3. Execution : use *execvp* to execute

```
/* ************************************EXECUTION************************************
 * call exevp with the command argv[i][0]
 * and arguments argv[i]
 * if return value is < 0, means it failed. print error msg
 * if errno is ENOENT, print error msg
 * ************************************EXECUTION************************************/
```

If parent, the first close the previous pipes pipefd[i-1][0] and pipefd[i-1][1]. This is because the parent no longer needs the previous pipe's read and writer ends when executing the i-th command. Then, check if the command is to run in the background or the foreground.
If the command tells to run the job in the background, then first add the job, then unblock the SIGCHLD signal. Then, parent waits for foreground job to terminate. If command tells to run the job in the foreground, then first add the job, unblock SIGCHLD signal.

builtin_cmd

In this function, it takes care of the four possible built in commands. If command is "quit", then just exit. If command is "jobs" then call *list jobs*. If command is "bg" or "fg", then call *do_fgfg* function.

do_bgfg

In this function, there are three things to do in large.

1. Handle errors: check if there is an argument, and if that is valid.
2. Get the corresponding job: check if the given argument is PID or JID.
   then use *getjobjid* or *getjobpid* function to get the job.
3. Finally, check if command is bg or fg.
   - if bg: change state of job to BG, and send SIGCONT to same process group.
   -if fg: change state of job to FG, and sent SIGCONT signal to same process group. then, wait for the foreground job to finish.

wait_fg

In this function, a meaningless loop is run while there exists a process with a given pid in the foreground process group.

sigchld_handler

This is a handler for the SIGCHLD signal. Call *waitpid* with WNOHANG | WUNTRACED option. This option enables the function to return 0 immediately if. Non of children in wait set has stopped or terminated, or return the PID of one of the stopped or terminated children. The status can tell how the process has terminated.

1. Terminated normally: just delete the job.
2. Terminated abnormally: means the sigint_handler was called.
   deletes the job.
3. When stopped: change the state of the job to ST.

sigint_handler

Get pid of current foreground process. Then, send a SIGINT signal to the entire foreground process group.

sigtstp_handler

Get pid of current foreground process. Then, send a SIGTSTP signal to the entire foreground process group.

Difficulties & Thoughts

The following were what I thought was difficult:

1. Understanding the background - foreground move of jobs

- Before learning system programming, I didn't even know there was a way to run a job in the background, and did not know that signals could be sent to only a certain group. I didn't really had an idea of what a background process or a foreground process is. However by making my own tiny shell, and using functions such as *wait_fg*, I learnt the difference between the foreground and background processes, and how the shell reacts to each.

2. Dealing with Pipes.

- dealing with a single pipe was hard enough, but in this lab I needed to arrange multiple pipes. It was very confusing when to make the *fdarray*, when to call *pipe()*, and when to close them. I took a long time debugging because I thought that I didn't need to close the pipes at the parent since this time, the pipes are being shared by the children. However, thinking about how the file descriptors were being duplicated using *fork()*, giddied out that the previous pipes that are no longer being used need to be closed at the parent't part.

In conclusion, the shell lab was really time-consuming. In the memory lab, the professor helped us a lot, and told as how most of the code worked, so could implement the expand heap, best fit, and next fit easily. However, this time, not much explanation about the code was provided, and I needed to take a long time understanding what each of the functions do.
Still, the time I spent on this lab and understanding how the sell works was really helpful. Thank you TAs so much for answering all my questions!