Prof. Heonyoung Yeom

System Programming (001)

Lab #2 / Shell Lab

: Writing Your Own Unix Shell

Report

Dept. of Computer Science and Engineering
2021-13194
Jaejun Ko

2023.4.14

Lab #2 Report
Shell Lab : Writing Your Own Unix Shell

Introduction

In this lab session, we will implement a simple Unix shell program, `tsh`, which supports job control. We expect to be familiar with process control and signaling by implementing this.

The `tsh` Specification

`tsh` should support the following built-in commands:

- The `quit` command terminates the shell.
- The `jobs` command lists all background jobs.
- The `bg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the background. The `<job>` argument can be either a PID or a JID.
- The `fg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the foreground. The `<job>` argument can be either a PID or a JID.

Also, typing `ctrl-c` (`ctrl-z`) should cause a SIGINT (SIGTSTP) signal to be sent to the current foreground job, as well as any descendants of that job (e.g., any child processes that it forked). The signal should have no effect if there is no foreground job.
If the command line ends with an ampersand `&`, then `tsh` should run the job in the background. Otherwise, it should run the job in the foreground.
Each job can be identified by either a process ID (PID) or a job ID (JID), which is a positive integer assigned by `tsh`. JIDs should be denoted by the prefix '`%`' on the command line. For example, "`%5`" denotes JID 5, and "`5`" denotes PID 5. (We have provided you with all of the routines you need for manipulating the job list.)
Finally, `tsh` should reap all of its zombie children. If any job terminates because it receives a signal that it didn't catch, then `tsh` should recognize this event and print a message with the job's PID and a description of the offending signal.
Most of the functionality was already implemented in the skeleton code. In this assignment, the following functions should be implemented:

- `eval`: Parses and interprets the command line.
- `builtin_cmd`: Recognizes and interprets the built-in commands: `quit`, `fg`, `bg`, and `jobs`.
- `do_bgfg`: Implements the `bg` and `fg` built-in commands.
- `waitfg`: Waits for a foreground job to complete.
- `sigchld_handler`: Catches SIGCHILD signals.
- `sigint_handler`: Catches SIGINT (`ctrl-c`) signals.
- `sigtstp_handler`: Catches SIGTSTP (`ctrl-z`) signals.

Implementation

In the eval function, the command line that the user has just typed in is evaluated. If the user has requested a built-in command (quit, jobs, bg, or fg) then execute it immediately. Otherwise, fork a child process and run the job in the context of the child. If the job is running in the foreground, wait for it to terminate and then return. For background children not to receive SIGINT (SIGTSTP) from the kernel when the user hit ctrl-c (ctrl-z) at the keyboard, each child process must have a unique process group ID. SIGCHLD should be blocked in the parent process until the addjob is executed since sigchld_handler calls deletejob when the child process is terminated, and it will cause an error if the job is not added yet. Of course, the child process must receive SIGCHLD correctly, hence it should be unblocked. To put the child in a new process group whose group ID is identical to the child's PID, the child process should call setpgid(0, 0) is called after the fork, but before the execve.

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    int bg;
    pid_t pid;
    sigset_t sset;

    bg = parseline(cmdline, argv);
    if(!builtin_cmd(argv)){
        while(sigemptyset(&sset) < 0);
        while(sigaddset(&sset, SIGCHLD) < 0);
        while(sigprocmask(SIG_BLOCK, &sset, NULL) < 0);
        if((pid = fork()) == 0){
            while(setpgid(0, 0) < 0);
            while(sigprocmask(SIG_UNBLOCK, &sset, NULL) < 0);
            if(execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found\n", argv[0]);
                exit(0);
            }
        }
        int added = addjob(jobs, pid, bg ? BG : FG, cmdline);
        while(sigprocmask(SIG_UNBLOCK, &sset, NULL) < 0);
        if(!added) return;
        if(!bg) waitfg(pid);
        else printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
    return;
}
```

The following function is the builtin_cmd function. If the user has typed a built-in command, then execute it immediately. If not, it returns 0, which means the input command is not a built-in command. If the user has typed only '&', there's nothing to execute, so it returns 1 to prevent the eval function create a child process.

```
int builtin_cmd(char **argv)
{
    char* command = argv[0];
    if(!strcmp(command, "quit")){
        exit(0);
        return 1;
    }
    else if(!strcmp(command, "bg") || !strcmp(command, "fg")){
        do_bgfg(argv);
        return 1;
    }
    else if(!strcmp(command, "jobs")){
        listjobs(jobs);
        return 1;
    }
    else if(!strcmp(command, "&")) return 1;
    return 0;
}
```

The following function is the `do_bgfg` function. It executes the built-in `bg` and `fg` commands. If the user did not give any argument, it prints an error message and returns. In the case the given argument is not in an appropriate form, it prints the corresponding message and returns. After that, find the job corresponding to the given argument from the job list. Again, it prints an error message if there does not exist such a job. If the job is found successfully, send `SIGCONT` to the process group. If the command is `bg`, set the job's state to `BG`, and print a log about the job. if the command is `fg`, change its state to `FG`, and wait for the process to end.

```
void do_bgfg(char **argv)
{
    if(argv[1] == NULL){
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
    struct job_t *job;
    if(argv[1][0] == '%'){
        job = getjobjid(jobs, atoi(&argv[1][1]));
        if(job == NULL){
            printf("%s: No such job\n", argv[1]);
            return;
        }
    }
    else if(isdigit(argv[1][0])){
        pid_t pid = atoi(argv[1]);
        job = getjobpid(jobs, pid);
        if(!job){
            printf("(%d): No such process\n", pid);
            return;
        }
    }
    else {
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    if(kill(-(job->pid), SIGCONT) < 0) return;
    if(!strcmp(argv[0], "bg")){
        job->state = BG;
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    }
    else {
        job->state = FG;
        waitfg(job->pid);
    }
    return;
}
```

In the `waitfg` function, it waits for the foreground job to terminate. By checking if we can find the job and its state is `FG`, it considers the foreground job running. Since the foreground job is unique and the signal should be handled in handlers, the busy loop around the `sleep` function, not `waitpid`, is used.

```
void waitfg(pid_t pid)
{
    struct job_t *job;
    job = getjobpid(jobs, pid);
    while(pid && job->state==FG) if(sleep(1) > 0) continue;
    return;
}
```

The following functions are signal handlers, `sigchld_handler`, `sigint_handler`, and `sigtstp_handler`. Whenever the shell receives `SIGCHLD` (by the kernel), `SIGINT`, or `SIGTSTP`, it calls `sigchld_handler`. It loops while there are terminated or stopped children. If `WIFSIGNALED(status)`, it means the child process is terminated by `ctrl-c`, so the handler should delete the job. If `WIFSTOPPED(status)`, it means the child process is terminated by `ctrl-z`, so the handler should change its state. otherwise, it must be terminated normally, so the handler just deletes the jobs from the list.

```
void sigchld_handler(int sig)
{
    pid_t pid;
    int status;
    struct job_t *job;
    while((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0){
        job = getjobpid(jobs, pid);
        if(WIFSIGNALED(status)){
            printf("Job [%d] (%d) terminated by signal %d\n",
                    job->jid, pid, WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if(WIFSTOPPED(status)){
            printf("Job [%d] (%d) stopped by signal %d\n",
                    job->jid, pid, WSTOPSIG(status));
            job->state = ST;
        }
        else deletejob(jobs, pid);
    }
    return;
}
```

If the user hit `ctrl-c`, `sigint_handler` will be called. If there is a foreground job running, send `SIGINT` to all processes in its process group. If the user hit `ctrl-z`, `sigtstp_handler` will be called. If there is a foreground job running, send `SIGTSTP` to all processes in its process group.

```
void sigint_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    if(pid != 0) if(kill(-pid, sig) < 0) return;
    return;
}

void sigtstp_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    if(pid != 0) if(kill(-pid, sig) < 0) return;
    return;
}
```

Result

To compare the outputs of `tsh` and `tshref`, which gives the output of the reference solution on all traces, the following script is used. `sed`s are used to ignore PID and trivial differences.

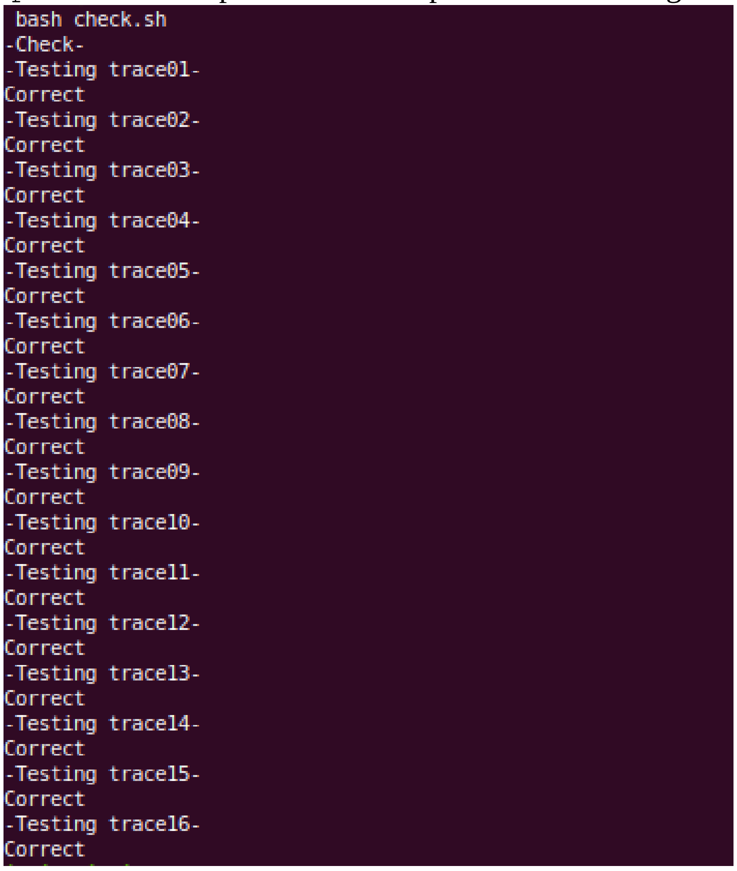
```
#!/usr/bin bash

mkdir -p my_output
mkdir -p answer
make > /dev/null

echo "-Check-"
for i in $(seq -f "%02g" 1 16)
do
        echo "-Testing trace$i-"
        make test$i > my_output/$i.txt
        make rtest$i > answer/$i.txt
        sed -i 's/tshref/tsh/' answer/$i.txt
        sed -i 's/rtest/test/' answer/$i.txt
        sed -i 's/(\b[0-9]\+\b)/(00000)/' answer/$i.txt
        sed -i 's/(\b[0-9]\+\b)/(00000)/' my_output/$i.txt
        sed -i 's/\b[0-9]\+\b pts/00000 pts/' answer/$i.txt
        sed -i 's/\b[0-9]\+\b pts/00000 pts/' my_output/$i.txt
        diff my_output/$i.txt answer/$i.txt && echo "Correct"
done

rm -rf my_output
rm -rf answer
```

The following is the output of the script. There can be some diffs when the shell executes `/bin/ps` since the output contains the process not running on the shell.



```
 bash check.sh
-Check-
-Testing trace01-
Correct
-Testing trace02-
Correct
-Testing trace03-
Correct
-Testing trace04-
Correct
-Testing trace05-
Correct
-Testing trace06-
Correct
-Testing trace07-
Correct
-Testing trace08-
Correct
-Testing trace09-
Correct
-Testing trace10-
Correct
-Testing trace11-
Correct
-Testing trace12-
Correct
-Testing trace13-
Correct
-Testing trace14-
Correct
-Testing trace15-
Correct
-Testing trace16-
Correct
```

Discussion

What was Difficult?

The most difficult part when implementing `tsh` was writing the `eval` function. There were many factors to think about in that the order of the code was a very important part, and the signal blocking had to be changed.

Considering when `waitpid` should be used was also tricky. As we discussed in the implementation part, `waitpid` should not be used in `waitfg`. If it is used, the `sigchld_handler` might be unable to catch all the signals from terminated children.

Something New and Surprising

Contrary to that I thought a foreground job would be executed in the original process, it was new that it was executed in a child process created to perform the job in the process and the parent process waits for it to terminate. Considering the characteristics of the function `execve`, it is a natural implementation, but I did not know exactly how the shell worked before I took a system programming class. It was an opportunity to realize once again that most of the work, such as process management or executing another program, can be done in C language.

Conclusion

In this lab session, we wrote a shell program, `tsh`, which supports job control. It has four built-in commands focused on managing processes. By directly implementing the signal handler, there were some improvements in understanding how the signal handler works. Also, the usage of various system calls gave some inspiration on how the C program can be improved in the future.