University of Colorado - Boulder

CSPB 2400
Computer Systems | Summer 2024

Lab 5: Shell Lab

Sam Walker

Hoang Truong

Monday, August 5, 2024

College of Engineering & Applied Science
UNIVERSITY OF COLORADO BOULDER

Part 1 - Execution Check

GitHub Repo Link: Repo Link [https://github.com/cu-cspb-2400-summer-2024/lab5-shelllab-sawa9885]

Screenshot of grading:

```
jovyan@jupyter-sawa9885:~/CSPB 2400/lab5-shelllab-sawa9885$ python3 shellAutograder.py
================================================================================
Running trace 01...
        Passed.
================================================================================
Running trace 02...
        Passed.
================================================================================
Running trace 03...
        Passed.
================================================================================
Running trace 04...
        Passed.
================================================================================
Running trace 05...
        Passed.
================================================================================
Running trace 06...
        Passed.
================================================================================
Running trace 07...
        Passed.
================================================================================
Running trace 08...
        Passed.
================================================================================
Running trace 09...
        Passed.
================================================================================
Running trace 10...
        Passed.
================================================================================
Running trace 11...
        Passed.
================================================================================
Running trace 12...
        Passed.
================================================================================
Running trace 13...
        Passed.
================================================================================
Running trace 14...
        Passed.
================================================================================
Running trace 15...
        Passed.
================================================================================
Running trace 16...
        Passed.
Total Passed: 16/16     Grade: 100%
```

Fig. 1

Part 2-i - Eval Function

Eval Function Code:

```c
void eval(char *cmdline)
{
  char *argv[MAXARGS];

  int bg = parseline(cmdline, argv);
  if (argv[0] == NULL)
    return;   /* ignore empty lines */
  if (!builtin_cmd(argv)) {
    pid_t pid = fork(); // Fork a child process
    if (pid == 0) { // Child process
      setpgid(0, 0); // Set the process group ID to the child's PID
      if (execve(argv[0], argv, environ) < 0) { // Execute the command
        printf("%s: Command not found\n", argv[0]);
        exit(1); // Exit if execve fails
      }
    } else if (pid > 0) { // Parent process
      addjob(jobs, pid, bg ? BG : FG, cmdline); // Add job to the job list
      if (!bg) {
        waitfg(pid); // Wait for the foreground job to complete
      } else {
        printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline); // Print background job info
      }
    }
  }
}
```

Fig. 2

Explanation

The eval function is the core component of the shell that processes and executes the command line input provided by the user. Initially, it uses the parseline function to break the input cmdline into an array of arguments argv and determine if the command should run in the background (bg). If the command is empty, the function simply returns. Otherwise, it checks if the command is a built-in command. If the command is not built-in, the function proceeds to create a new process using fork(). In the child process (pid == 0), it sets the process group ID using setpgid(0, 0) to manage job control signals and attempts to execute the specified command with execve. If the command execution fails, it prints an error message and exits the child process. In the parent process (pid > 0), it adds the new job to the job list with addjob. If the command is a foreground job (!bg), it waits for the job to complete using waitfg. For background jobs, it immediately prints the job ID and PID.

Trace Implications

- **Trace 01:** `eval` is used to detect EOF to allow the shell to exit gracefully.
- **Trace 03:** `eval` handles executing commands in the foreground by forking a new process and managing job completion.
- **Trace 04:** `eval` manages background execution by forking processes and adding them to the job list.
- **Trace 14:** `eval` processes command line input and provides error messages for unrecognized commands or incorrect arguments.
- **Trace 15:** `eval` integrates all functionalities to handle complex job management and command execution in both foreground and background.

Part 2-ii - Signaling Mechanism

Signaling Mechanism Code:

```c
void sigchld_handler(int sig)
{
    int old_errno = errno;
    pid_t pid;
    int status;

    // Reap child processes that have changed state
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        if (WIFEXITED(status)) {
            // Child process exited normally
            deletejob(jobs, pid); // Remove terminated job from job list
        } else if (WIFSIGNALED(status)) {
            // Child process was terminated by a signal
            int jid = pid2jid(pid); // Get job ID
            printf("Job [%d] (%d) terminated by signal %d\n", jid, pid, WTERMSIG(status));
            deletejob(jobs, pid); // Remove terminated job from job list
        } else if (WIFSTOPPED(status)) {
            // Child process was stopped by a signal
            struct job_t *job = getjobpid(jobs, pid);
            if (job) {
                job->state = ST; // Mark job as stopped
                int jid = pid2jid(pid); // Get job ID
                printf("Job [%d] (%d) stopped by signal %d\n", jid, pid, WSTOPSIG(status));
            }
        }
    }
    errno = old_errno; // Restore errno to its previous value
}
```

(a) sigchld

```c
/////////////////////////////////////////////////////////////////////////////
//
// sigint_handler - The kernel sends a SIGINT to the shell whenver the
//    user types ctrl-c at the keyboard.  Catch it and send it along
//    to the foreground job.
//
void sigint_handler(int sig)
{
    pid_t pid = fgpid(jobs); // Get the PID of the foreground job
    if (pid != 0)
        kill(-pid, SIGINT); // Send SIGINT to the entire foreground process group
}

/////////////////////////////////////////////////////////////////////////////
//
// sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
//     the user types ctrl-z at the keyboard. Catch it and suspend the
//     foreground job by sending it a SIGTSTP.
//
void sigtstp_handler(int sig)
{
    pid_t pid = fgpid(jobs); // Get the PID of the foreground job
    if (pid != 0)
        kill(-pid, SIGTSTP); // Send SIGTSTP to the entire foreground process group
}
```

(b) sigint and sigtstp

Fig. 3

Explanation

In a Unix shell, blocking and unblocking signals is important for managing process control and preventing race conditions. Blocking signals temporarily disables them during critical sections of code to ensure consistency, such as when a new process is being forked and added to the job list. This is done using functions like sigprocmask, which can block signals like SIGCHLD, SIGINT, and SIGTSTP by adding them to a signal mask. Unblocking signals is done by restoring the previous signal mask once the critical operation is complete. In the sigchldhandler, the shell handles SIGCHLD signals sent by the kernel when a child process changes state, such as terminating or stopping. The handler reaps zombie processes by calling waitpid with appropriate options and updates the job list by removing terminated jobs or marking jobs as stopped. The siginthandler forwards the SIGINT signal to the foreground job's process group when the user types Ctrl-C, causing the foreground job to terminate. Similarly, the sigtstphandler forwards the SIGTSTP signal to the foreground job when the user types Ctrl-Z, suspending the job and marking it as stopped. These handlers are crucial for correctly managing the job states in the shell.

Trace Implications

- **Trace 06:** `sigint_handler` captures Ctrl-C inputs and forwards the SIGINT signal to terminate the foreground job.
- **Trace 07:** `sigint_handler` ensures that SIGINT signals affect only the foreground job.
- **Trace 08:** `sigtstp_handler` intercepts Ctrl-Z inputs to suspend the foreground job.
- **Trace 11:** `sigint_handler` broadcasts SIGINT to all processes within the foreground process group.
- **Trace 12:** `sigtstp_handler` sends SIGTSTP to all processes in the foreground group to stop them.
- **Trace 16:** `sigint_handler` and `sigtstp_handler` handle external signals from other processes.

Part 2-iii - Analysis Insight

While implementing the shell lab, one of the significant challenges I faced was handling signal forwarding for foreground jobs, particularly with SIGINT and SIGTSTP. Initially, these signals did not affect the intended processes as expected, leading to unexpected behavior when attempting to interrupt or stop jobs. To address this, I verified that the foreground job was correctly identified using the fgpid function and ensured that signals were sent to the entire process group using kill(-pid, signal), where pid is the foreground job's process ID. This correctly targeted the process group, and debug print statements helped confirm that signals were reaching the appropriate PIDs. Another challenge was managing transitions between background and foreground jobs with the fg and bg commands. Some jobs did not resume correctly, and the shell sometimes failed to wait for foreground jobs. To solve this, I reviewed the dobgfg function to ensure proper parsing of job IDs and PIDs and updated job states to running or foreground as needed. Additionally, I used waitfg to block the shell until foreground jobs completed, testing various job states to verify expected behavior. As an aside, ensuring error messages matched the reference implementation exactly was crucial for passing trace tests, involving meticulous attention to formatting details like spaces and punctuation, but particularly newline characters. Throughout the entire lab I kept finding myself wondering which code was being ran. However, I found it much easier to default to temporary print statements rather than gdb because I could encode certain values into the print statements very quickly and modularly.