Project Infant Shell

Due: November 5, 2014

1   Introduction

You've now created your very own shell. As great as Newborn Shell is, its capabilities are limited; for example, your shell can run only one program at a time since it runs programs in the *foreground*, occupying the entire shell. This isn't especially useful: most of the time, you want to be able to run multiple processes concurrently. Additionally, if that process were to hang or become unresponsive, it would be impossible to end it without also killing the shell.

2   Assignment

In this assignment, you will be expanding your Newborn Shell with a basic *job control* system: your shell will be able to handle multiple processes by running some in the *background*, as well as manage a foreground process with signal forwarding.

2.1   Stencil

There is some stencil code for this assignment. You can install it in your home directory by running

`cs033_install shell_2`

However, most of the support code for this project is your Newborn Shell. Make sure to copy all of your Newborn Shell source to your Infant Shell directory. You can use the Makefile from Newborn Shell to run Infant Shell, but you will want to compile with the `-D_GNU_SOURCE` or `-std=gnu99` flag to avoid potential headaches.

2.2   Jobs vs. Processes

Suppose you execute your shell within your shell. Then the inner shell itself spawns its own child processes in response to the commands it receives.

In this situation, the shell is not running just a single foreground process, since the command it has executed now comprises multiple processes: one for the inner shell, and additional processes spawned by the inner shell.

Still, from the point of view of the outer shell, the inner shell and its child processes should be considered a single unit. This unit is called a *job*. A *job* is a process or group of processes created by your shell from a single command.

2.3   Foreground vs. Background

The notion of *foreground* and *background* jobs is crucial to a job control system.

A *foreground* job is the group of processes with which the user interacts. When the current version of your shell executes a command, that process receives all commands; the shell itself must wait for that process to terminate before it can resume receiving user input.

A *background* job is the opposite. Rather than take over the shell's interface, background processes run behind it. The shell still receives commands while the background processes execute without interrupting the shell.

Because background processes do not take over the shell's interface, it is possible for a shell which runs processes in the background to execute many jobs at once.

2.4   Specification

In addition to maintaining its current behavior, your shell should now exhibit the following additional behaviors:

- If `CTRL-C`, `CTRL-Z`, or `CTRL-\` is typed into your shell, the resulting signal (`SIGINT`, `SIGTSTP`, or `SIGQUIT`, respectively) should be sent to the currently-running foreground job (if one exists) instead of to your shell. If no foreground job is running, then nothing should happen.

- If a command ends with the character `&`, that command should run in the background. Otherwise, that command should be run in the foreground.

- When starting a job in the background, a message indicating its job and process IDs should be printed to standard output. In Infant Shell, this is formatted as:

  `[<job id>] (<process id>)`

- Jobs started by your shell that have terminated should be *reaped* (see section 5).

- Whenever a job is terminated by a signal, your shell should print a brief message indicating the cause of termination, which can be determined by the status set by `waitpid()`. The message should be formatted as follows:

  `[<jid>] (<pid>) terminated by signal <signal number>`

- Whenever a job is stopped by a signal, you should print a similar message formatted as follows:

  `[<jid>] (<pid>) suspended by signal <signal number>`

- Whenever a background job exits normally, print yet another similar message as follows:

  `[<jid>] (<pid>) terminated with exit status <status>`

- Whenever a job is resumed via a `SIGCONT` signal, print another similar message with the following formatting:

  `[<jid>] (<pid>) resumed`

- The following additional built-in commands should be supported, offering the same functionality and syntax as `bash` and most other shells:

- – `jobs` lists all the current background jobs, listing each job's job ID, state (running or suspended), and command used to execute it. We have implemented this printing for you in the support code. You must call `jobs()` to print all the jobs in your job list. See Section 7 for more information on the support code. For simplicity's sake, assign each job its ID and add it to the job list when it is created (even if it is a foreground job). The first job should have ID 1, the second ID 2, and so on.
- – `bg %<job>` resumes `<job>` (if it is suspended) and runs it in the background, where `<job>` is a job ID.
- – `fg %<job>` resumes `<job>` (if it is suspended) and runs it in the foreground. `<job>` is a job ID, just as in the `bg` command.

The % above simply tells your shell that you are referencing a job ID. To maximize compatibility with the provided testing program (described in Section 9), your shell should print no other output unless it is compiled with `PROMPT` defined (as in Newborn Shell).

The following sections provide information that you will find invaluable in producing the above behaviors. We recommend completing this project in a linear manner, fully finishing each part of a given section before moving on to the next section.

3   Part I: Signals

In Newborn Shell, when a job is launched, the shell can do nothing but wait until that job finishes. But what if the shell needs to perform some other task, or that job will never finish? There needs to be some way to interrupt execution of that job.

Signals are delivered asynchronously by the operating system to do precisely that. When you type characters on the keyboard, those characters are delivered to the current terminal. The terminal processes those characters and then performs some action upon its *foreground process group.* Most characters are simply passed along to be read by that process, but some, such as `CTRL-C`, generate a signal which is sent to every process in the terminal's foreground process group instead.

Therefore, in order to properly forward a signal to a job running in the foreground , your shell must set the process group ID of the terminal to be that of the foreground job. If there is no current foreground job, then the terminal's process group ID should instead be set to that of the shell.

3.1   Ignoring Signals

If you type `CTRL-C` (for example) when your shell has no foreground job, the controlling terminal will send `SIGINT` to your shell only. To avoid having the shell exit (or suspend execution), you must override its default behavior when it receives those signals so that it ignores the signal instead. You should **not** install a signal handler that does nothing. Instead, you should set the response to the signal to be `SIG_IGN`. You may do this with either the `sigaction()` system call or more concisely invoke the `signal()` system call.

3.2   Process Groups

- • `int setpgid(pid_t pid, pid_t pgid)`

- `int tcsetpgrp(int fd, pid_t pgrp)`

When you `fork()` off a new child process, that process begins execution as a copy of its parent, also inheriting its parent's *process group ID*. For signals to be properly sent to the new child but not also your shell, the process group ID of that child must be changed.[1]

The `setpgid()` system call fortunately exists to make this change. All that remains, then, is to ensure that each job has a unique process group ID. This is simple to do — individual process IDs are guaranteed to be unique, so all you need to do is set the process group ID of a job to its process ID. The child process must do this immediately after it returns from `fork()` before `execv()` is called.

A separate routine is necessary to make sure that the terminal sends signals to the right processes, however. `tcsetpgrp()` conveniently does this, changing the process group that the terminal sends typed characters and signals to. Use this system call to make sure that whenever you have a foreground job, the terminal's process group is the same as that job's process group, and the process group of your shell otherwise. The `fd` argument to `tcsetpgrp()` should be the file descriptor for `stdin`, which is a file descriptor associated with the controlling terminal. Make sure to reset the terminal process group to your shell after your foreground job has completed!

4   Part II: Multiple Jobs

Once your shell can manage signals and its foreground job, it is time to augment it so that it can run jobs in the background as well as the foreground. If your shell reads a '`&`' character at the end of a line of input, it should execute that command in the background. You can assume that '`&`' will only appear as the last non-whitespace character on a line.

Because a process running in the background immediately returns control of the program interface to the shell, you should be able to run a large number of background jobs (only limited by system resources). You should not wait for a background job to complete before launching another job.

Keep in mind that background jobs are run in seperate process groups. You will need to track each background job using a list. We have provided several functions that you can use to do this. For each job you are responsible for tracking its job ID, process ID, command, and state. See Section 7: Support.

4.1   Process State

At any time, a process can be in one of several states: it may be running, stopped, or terminated[2]. Stopped processes have been suspended, but are still active and can be resumed by the user at any time. Terminated processes, referred to as *zombies*, have stopped execution and can no longer be executed, but persist on the system, continuing to consume resources until they are explicitly *reaped*.

A process' initial state is running, and its state can be altered using signals. For example, `SIGINT`

---

[1]This will also matter later on when your shell begins to support multiple jobs; changing the process groups of those jobs is necessary so that signals will be sent to only the foreground job.

[2]There are other states, but your shell need not track them.

terminates a process, and SIGTSTP stops a process. SIGCONT can be sent to resume execution of a stopped process.

5   Part III: Reaping

Once your shell is capable of running background jobs, it is time to properly track them. At present, your shell uses the `wait()` or `waitpid()` system call to wait for its foreground process to finish before displaying another prompt and running another command. Once the process has finished, the `wait()` or `waitpid()` also *reaps* the process, freeing any resources that persist on the system. Waiting for a background job to finish defeats the purpose of the background job, however. What's needed is a means for knowing when any child process is terminated, then to getting them out of the zombie state to clean up the process table.

5.1   *Reaping* a Child Process

- `int waitpid(pid_t pid, int *status, int options)`

The `waitpid()` system call allows your shell to do exactly that: examine a child process which has changed state, and instruct the operating system to clean up any resources associated with that child if it terminated.

The `options` parameter may be set to any combination of the following flags, or to 0:

- `WNOHANG` instructs the function to return immediately without hanging if no child process has terminated, returning 0 and leaves the contents of the integer pointed to by `status` unchanged. If any child process indicated by the `pid` argument *has* changed state, `waitpid` returns the process id of that child and updates `status` as usual.

- `WUNTRACED` instructs the function to return if a child process has been stopped, even if no child process has terminated; and

- `WCONTINUED` instructs the function to return if a child has been restarted from a stopped state.

`waitpid()` stores information about what happened to a child process in the location pointed to by `status` (if `status` is not NULL). You'll want to access this information so your shell can print an informative message about what happened. You can use the following macros, all of which have a single integer `status` as an argument, to access this information:

- `WIFEXITED(status)`, which returns true if the process terminated normally and 0 otherwise;

- `WEXITSTATUS(status)`, which returns the exit status of a normally-terminated process;

- `WIFSIGNALED(status)`, which returns true if the process was terminated by a signal and 0 otherwise;

- `WTERMSIG(status)`, which returns the signal which terminated the process if it terminated by a signal;

- `WIFSTOPPED(status)`, which returns true if the process was stopped by a signal;

- WSTOPSIG(status), which returns the signal which stopped the process if it was stopped by a signal;

- WIFCONTINUED(status), which returns true if the process was resumed by SIGCONT and 0 otherwise.

Shells commonly post changes in the status of jobs to the terminal as they are about to try and print another prompt. This would be a good place to check which jobs have changed state in your shell, as well.

6 Part IV: fg and bg

Now that you've added all of the other functionality, it is time to add the last piece of the puzzle: restarting a stopped job in the foreground or background using the commands fg and bg respectively, or moving a background job into the foreground with fg.

For both of these commands the first thing you should do is send a SIGCONT signal to the stopped job.

The rest is merely a matter of managing jobs in your shell.

Some programs that might help you test management of foreground and background processes include:

- /bin/sleep

- /usr/bin/yes item /usr/bin/find

Note that if you try to run any program that reads from standard input (such as /bin/cat) in the background, it will receive the signal SIGTTIN from the terminal and be suspended. You should be able to run (and restart) such processes in the foreground with no problems, however.

7 Support Code

You are provided with support code which will help you manage the job list. The *jobs.h* file contains declarations of the functions that you can use to this; those functions are defined in the *jobs.c* file. You are not responsible for understanding how the functions in *jobs.c* are implemented, although you are encouraged to read through it.

Here are the functions you are provided with:

- init_job_list(): initializes a job list (a job_list_t) and returns a pointer to it.

- cleanup_job_list(): cleans up the job list, de-allocating any system resources associated with it. Make sure you call this before your program exits!

- add_job(): adds a job to the job list. Each job in the job list has a job id, process id, state, and command (which is the command used to start that job).

- remove_job_jid() and remove_job_pid(): remove a job from the job list, indicated by the job's job id or process id respectively.

- update_job_jid() and update_job_pid(): update the process state of the job indicated by the given job id or process id respectively.

- get_job_pid(): gets the process id of a job indicated by a job id.

- get_job_jid(): gets the job id of a job indicated by a process id.

- get_next_pid(): returns the process id of the next job in the job list, or −1 if the end of the list has been reached. Future calls to this function will then resume from the start of the list.

- jobs(): prints out the contents of the job list. Useful for implementing the jobs builtin command.

You are also provided with two macros, _STATE_RUNNING and _STATE_STOPPED which correspond to process states. Use these macros whenever you need to initialize or update a process state.

To use these functions in your program, you must do two things. First, you must include *jobs.h* in any file in which you reference any of these functions. Second, you must make sure to compile your program with *jobs.c*. You can do so by updating the Makefile from Newborn Shell to include *jobs.c* in the list of files to compile (update "sh.c" to use "sh.c jobs.c" wherever applicable).

8   Library Functions

The same restriction on non-syscall functions from Newborn Shell applies, but you are also allowed to use the support code. Remember that atoi() and sprintf() are allowed functions; these functions should be sufficient to do any string manipulation or formatting that you might need.

9   Testing Your Shell

To help test your shell, we have provided you with a tester. To use the tester, run

cs033_shell_2_test -s <33noprompt>

Optional arguments to the tester include:

-h	Print help message
-v	Trace progress
-e	Like -v, but trace output only on error
-r	Compare against reference shell (see below)
-t <tracefile>	Check one tracefile (default: check all)
-s <sh>	Run tests on <shell>

If a test fails, the other tests will not be run, but you can specify which test you want to run using the -t option. The tests can be found in /course/cs033/pub/shell_2.

Note: Sometimes the driver may read in more than one line at a time, causing the test to fail. If you see this happening, you can re-run the tests. If you determine this to be the source of the error and when run again the test passses, you can assume your code is correct.

9.1 Format of the Tracefiles

Trace files contain a sequence of commands for your shell to run, interspersed with special commands that the driver interprets to mean special actions are to be taken.

TSTP	Send a SIGTSTP signal to the shell
INT	Send a SIGINT signal to the shell
QUIT	Send a SIGQUIT signal to the shell
KILL	Send a SIGKILL signal to the shell
CLOSE	Close writer (sends EOF to shell)
WAIT	Wait for shell to terminate
SLEEP <n>	Sleep for n seconds

9.2 Test Executables

The testing scripts utilize small executables that are located (with their source code) in the */course/cs033/pub/shell_2/programs* directory. It will be nearly impossible to understand what the tests are doing without understanding these programs, so we recommend examining the README contained in that directory as well as the source code for some of the programs.

9.3 Demo

When in doubt, follow the implementation provided in the demos. The demos can be invoked using **cs033_shell_2_demo** and **cs033_shell_2_demo_noprompt** respectively. The demos are produced from identical source code, but the first is compiled with the macro PROMPT defined, and consequently produces additional output. The provided testing infrastructure will optionally test your shell's output against output provided by the noprompt demo.

10 Handing In

To hand in the second part of your shell, run

cs033_handin shell_2

from your project working directory. You must at a minimum hand in all of your code, the Makefile used to compile your program (if you use one), and a README documenting the structure of your program, any bugs you have in your code, any extra features you added, and how to compile it (if you do not use a Makefile).

If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.