Project Shell Revisited

Due: November 9, 2012

1 Introduction

So you've now created your very own shell. As great as that shell is, it is unfortunately somewhat limited - for example, your shell can run only one program at a time 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. In this assignment you will be expanding your previous 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 Assignment

For this assignment, you will be augmenting your shell with a basic job control system, allowing it to run and handle multiple processes simultaneously.

2.1 Jobs vs. Processes

Suppose you execute your shell within your shell. The inner shell itself can then spawn its own child processes as it executes commands passed to it. Consequently, it would not be correct to say that the outer shell is running a single foreground process, since the command it has executed is no longer contained within just one process.

In spite of this, it is clear that from the point of view of the outer shell, the inner shell and its child process 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.

2.2 Foreground vs. Background

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

A *foreground* job is the process with which the user interfaces. 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, a background process runs behind it. The shell still receives commands while the background process executes 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.

2.3 Specification

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

- If a command ends with the character `&`, that command should be started and run in the background. Otherwise, that command should be run in the foreground.
- If a `SIGINT` signal is sent to your shell, that signal should be caught and forwarded to the currently-running foreground process, if one exists.
- If a `SIGTSTP` signal is sent to your shell, that signal should be caught and forwarded to the currently-running foreground process, if one exists; control should then be returned to the shell.
- The following additional built-in commands should be supported:
  - `jobs` lists all the current background jobs, listing each job's job ID, parent process 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 8: Support for more information on the support code.
  - `bg <job>` restarts (if it is stopped) `<job>` and runs it in the background. `<job>` can be either a process ID or a job ID. A `%` character before the ID indicates a job ID, e.g. `%1`.
  - `fg <job>` restarts (if it is stopped) `<job>` and runs it in the foreground. `<job>` can be either a process ID or a job ID. A `%` character before the ID indicates a job 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:

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

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

3 Part I: Signals

Presently, your shell can launch a single job and wait for it to return normally. The only way to stop that job mid-execution is with an interrupt or stop signal. However, you cannot send a signal to that job without passing it through your shell, which would kill your shell in addition to killing that job. Therefore, you must catch and forward these signals to each process associated with that job.

To do this, you'll need to write signal handling functions and install them correctly. Your signal handling function(s) should forward a signal to each process associated with the current foreground job. You should do this for the following signals:

- `SIGINT`, which is sent with CTRL-C;
- `SIGTSTP`, which is sent with CTRL-Z; and
- `SIGQUIT`, which is sent with CTRL-\.

3.1 Forwarding a Signal

- `int kill(pid_t pid, int sig)`

To forward a signal, you will have to make use of the `kill()` system call, which sends the signal `sig` to the process identified by `pid`. However, sending `sig` to *only* the process indicated by `pid` won't accomplish what is needed; it is necessary to send `sig` to the indicated process and each of its child processes.

Fortunately, it is easy to locate the child processes of a parent process. Each process has a *process group id* (pgid), which it inherits from its parent process.

Even more fortunately, `kill()` provides an easy mechanism for sending a signal to each process in a process group: if `pid` is negative, then `kill()` sends the indicated signal to each process in the process group indicated by `-pid`. To use `kill`, you may need to compile your program with the `gcc` flag `-D_GNU_SOURCE` or include the following line at the top of your file (before any `#include`s):

```
#define _GNU_SOURCE
```

There is one problem, though: each job run by your shell is a child of the shell, and will therefore share its process group id (pgid). Unless your shell changes the process group id of each of its jobs, forwarding a signal to the entire process group of a job will re-send that signal to your shell.

3.1.1 Changing a Process' Process Group

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

The `setpgid()` system call provides a way to do exactly that. 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()`.

Once your shell changes the process group ids of its child processes, it will no longer be able to execute jobs which attempt to read from `stdin`. If two different processes from distinct process groups attempt to read from the same `stdin` of the same terminal, only one process can succeed; the other is sent a `SIGTTIN` signal and is stopped immediately. While this is unfortunate in terms of the practicality of the shell, it is fortunate for testing; this signal causes a change in state which should be handled by your `SIGCHLD` handler (which you will implement in Part III; see section 5) without much extra maneuvering on your part.

4 Part II: Multiple Jobs

Now that your shell can forward signals to 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.

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.

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 8: Support.

4.1 Process State

At any time, a process can be in one of several states: it may be running, stopped, or terminated[1]. 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` terminates a process, and `SIGTSTP` stops a process. `SIGCONT` can be sent to resume execution of a stopped process.

5 Part III: Reaping

Now that your shell is capable of maintaining background jobs, it is time to make them practical. At present, your shell uses the `wait()` system call to wait for the most-recently-executed job to finish before waiting for another command. Doing so is clearly not conducive to executing background jobs. It is thus vital to remove this call from your program. However, this creates two problems:

- the distinction between foreground and background jobs no longer exists; and
- your shell has no way of knowing when any background job has changed state or terminated, which means the operating system will never be able to free resources associated with those jobs.

The first problem is fixed by using a "busy loop" - while a foreground job is running, do nothing.

The second problem is conveniently resolved with another signal, called `SIGCHLD`. This signal is sent to a process when any of that process' children changes state.

Since `SIGCHLD` does not specify which child has changed state, or what the new state of that child is, you will have to write a signal handler for this signal so that your shell checks over and explicitly cleans up each child process which has changed state.

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

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.

This system call, by default, only waits for children which have terminated. However, by providing the `options` bit vector with the correct flags, you can instruct the call to wait for other changes.

- `WNOHANG` instructs the function to return immediately without hanging if no child process has terminated;
- `WUNTRACED` instructs the function to return if a child process has been stopped, even if no child process has terminated;
- `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 `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 1 if the process terminated normally and 0 otherwise;
- `WEXITSTATUS(status)`, which returns the exit status of a normally-terminated process;
- `WIFSIGNALED(status)`, which returns 1 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 1 if the process was resumed by `SIGCONT` and 0 otherwise.

A change in state of the foreground job also triggers a `SIGCHLD`, so make sure your shell reacts appropriately.

- if that child has terminated, then it should be reaped, and control should be returned to the shell.
- if that child has been stopped, it should be moved to the background, and control should be returned to the shell.

6 Part IV: Race Conditions

It is impossible to determine exactly when a signal will be sent to (or received by) your program. This results in something called a *race condition*, which is a flaw in a process where the output or result is dependent on the sequence of other events. Right now, it is entirely possible that a child process will start and terminate before it can be added to the list of jobs. To prevent this from happening, your shell must *mask off* `SIGCHLD` before it `fork()`s, removing the mask after the child is added to the job list (the child can remove the mask immediately).

6.1 Masking Off a Signal

- `int sigemptyset(sigset_t *set)`
- `int sigaddset(sigset_t *set, int signum)`
- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`

A *signal mask* temporarily prevents the signals blocked by the mask from reaching the target process until after the mask is removed[2] The `sigset_t` type is used to set up a signal mask. `sigemptyset()` initializes a `sigset_t`; `sigaddset()` adds the signal `signum` to the given set. Once a `sigset_t` has been set up, `sigprocmask()` performs the action `how` using that set, storing the previous signal mask in `oldset` if it is not `NULL`.

`how` is one of three actions:

- `SIG_BLOCK` blocks each of the signals in `set`.
- `SIG_UNBLOCK` unblocks each already-blocked signal in `set`.
- `SIG_SETMASK` sets the set of blocked signals to exclusively those in `set`.

Each of the above functions returns 0 upon success and -1 on failure.

7 Part V: `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.

[2] At most one signal of each type will reach the target process once the mask is removed.

8 Support

You are provided with support code which manages the list of jobs for you. 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.

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.

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 Shell 1 to include *jobs.c* in the list of files to compile (update "`SRC = sh.c`" to "`SRC = sh.c jobs.c`").

9 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.