Controlling Your Processes

To use a stage metaphor, all the processes you want to run on your machine are like actors, and you are the director. You control when and how they run. But, how can you do this? Well, let's look at the possibilities.

The first step is to run the executable. Normally, when you run a program, all the input and output is connected to the console. You see the output from the program and can type in input at the keyboard. If you add an & to the end of a program, this connection to the console is severed. Your program now runs in the background, and you can continue working on the command line. When you run an executable, the shell actually creates a child process and runs your executable in that structure. Sometimes, however, you don't want to do that. Let's say you've decided no shell out there is good enough, so you're going to write your own. When you're doing testing, you want to run it as your shell, but you probably don't want to have it as your login shell until you've hammered out all the bugs. You can run your new shell from the command line with the exec function:

```
exec myshell
```

This tells the shell to replace itself with your new shell program. To your new shell, it will look like it's your login shell—very cool. You also can use this to load menu programs in restricted systems. That way, if your users kill off the menu program, they will be logged out, just like killing off your login shell. This might be useful in some cases.

Now that your program is running, what can you do with it? If you need to pause your program temporarily (maybe to look up some other information or run some other program), you can do so by typing Ctrl-z (Ctrl and z at the same time). This pauses your program and places it in the background. You can do this over and over again, collecting a list of paused and "backgrounded" jobs. To find out what jobs are sitting in the background, use the `jobs` shell function. This prints out a list of all background jobs, with output that looks like this:

```
[1]+  Stopped                 man bash
```

If you also want to get the process IDs for those jobs, use the `-l` option:

```
[1]+  26711  Stopped       man bash
```

By default, `jobs` gives you both paused and running background processes. If you want to see only the paused jobs, use the `-s` option. If you want to see only the running background jobs, use the `-r` option. Once you've finished your sidebar of work, how do you get back to your paused and backgrounded program? The shell has a function called `fg` that lets you put a program back into the foreground. If you simply execute `fg`, the last process backgrounded is pulled into the foreground. If you want to pick a particular job to put in the foreground, use the `%` option. So if you want to foreground job number 1, execute `fg %1`. What if you want your backgrounded jobs to continue working? When you use Ctrl-z to put a job in the background, it also is paused. To get it to continue running in the background, use the `bg` shell function (on a job that already has been paused). This is equivalent to running your program with an & at the end of it. It will stay disconnected from the console but continue running while in the background.

Once a program is backgrounded and continues running, is there any way to communicate with it? Yes, there is—the signal system. You can send signals to your program with the `kill procid` command, where procid is the process ID of the program to which you are sending the signal. Your program can be written to intercept these signals and do things, depending on what signals have been sent. You can send a signal either by giving the signal number or a symbolic number. Some of the signals available are:

- 1: SIGHUP — terminal line hangup
- 3: SIGQUIT — quit program
- 9: SIGKILL — kill program
- 15: SIGTERM — software termination signal
- 30: SIGUSR1 — user-defined signal 1
- 31: SIGUSR2 — user-defined signal 2

If you simply execute `kill`, the default signal sent is a SIGTERM. This signal tells the program to shut down, as if you had quit the program. Sometimes your program may not want to quit, and sometimes programs simply will not go away. In those cases, use `kill -9 procid` or `kill -s SIGKILL procid` to send a kill signal. This usually kills the offending process (with extreme prejudice).

Now that you can control when and where your program runs, what's next? You may want to control the use of resources by your program. The shell has a function called `ulimit` that can be used to do this. This function changes the limits on certain resources available to the shell, as well as any programs started from the shell. The command `ulimit -a` prints out all the resources and their current limits. The resource limits you can change depend on your particular system. As an example (this crops up when trying to run larger Java programs), say you need to increase the stack size for your program to 10000KB. You would do this with the command `ulimit -s 10000`. You also can set limits for other resources like the amount of CPU time in seconds (-t), maximum amount of virtual memory in KB (-v), or the maximum size of a core file in 512-byte blocks (-c).

The last resource you may want to control is what proportion of the system your program uses. By default, all your programs are treated equally when it comes to deciding how often your programs are scheduled to run on the CPU. You can change this with the `nice` command. Regular users can use nice to alter the priority of their programs down from 0 to 19. So, if you're going to run some process in the background but don't want it to interfere with what you're running in the foreground, run it by executing the following:

```
nice -n 10 my_program
```

This runs your program with a priority of 10, rather than the default of 0. You also can change the priority of an already-running process with the `renice` program. If you have a background process that seems to be taking a lot of your CPU, you can change it with:

```
renice -n 19 -p 27666
```

This lowers the priority of process 27666 all the way down to 19. Regular users can use nice or renice only to lower the priority of processes. The root user can increase the priority, all the way up to –20. This is handy when you have processes that really need as much CPU time as possible. If you look at the output from `top`, you can see that something like pulseaudio might have a negative niceness value. You don't want your audio skipping when watching movies.

The other part of the system that needs to be scheduled is access to IO, especially the hard drives. You can do this with the `ionice` command. By default, programs are scheduled using the best-effort scheduling algorithm, with a priority equal to (niceness + 20) / 5. This priority for the best effort is a value between 0 and 7. If you are running some program in the background and don't want it to interfere with your foreground programs, set the scheduling algorithm to "idle" with:

```
ionice -c 3 my_program
```

If you want to change the IO niceness for a program that already is running, simply use the `-p procid` option. The highest possible priority is called real time, and it can be between 0 and 7. So if you have a process that needs to have first dibs on IO, run it with the command:

```
ionice -c 1 -n 0 my_command
```

Just like the negative values for the nice command, using this real-time scheduling algorithm is available only to the root user. The best a regular user can do is:

```
ionice -c 2 -n 0 my_command
```

That is the best-effort scheduling algorithm with a priority of 0.

Now that you know how to control how your programs use the resources on your machine, you can change how interactive your system feels.

—JOEY BERNARD