# Job control

## Overview

**Teaching:** 5 min
**Exercises:** 0 min
**Questions**

- How do keep track of the process running on my machine?
- Can I run more than one program/script from within a shell?

**Objectives**

- Learn how to use `ps` to get information about the state of processes
- Learn how to control, ie., "stop/pause/background/foreground" processes

The shell-novice lesson explained how we run programs or scripts from the shell's command line.

We'll now take a look at how to control programs *once they're running*. This is called job control ({ page.root }}/reference/#job-control), and while it's less important today than it was back in the Dark Ages, it is coming back into its own as more people begin to leverage the power of computer networks.

When we talk about controlling programs, what we really mean is controlling *processes*. As we said earlier, a process is just a program that's in memory and executing. Some of the processes on your computer are yours: they're running programs you explicitly asked for, like your web browser. Many others belong to the operating system that manages your computer for you, or, if you're on a shared machine, to other users.

## The `ps` command

You can use the `ps` command to list processes, just as you use `ls` to list files and directories.

### Behaviour of the `ps` command

The `ps` command has a swathe of option flags that control its behaviour and, what's more, the sets of flags and default behaviour vary across different platforms.

A bare invocation of `ps` only shows you basic information about *your*, *active* processes.

After that, this is a command that it is worth reading the '`man` page' for.

```
$ ps
```

```
  PID TTY          TIME CMD
12767 pts/0    00:00:00 bash
15283 pts/0    00:00:00 ps
```

At the time you ran the `ps` command, you had two active processes, your ( `bash` ) shell and the ( `ps` ) command you had invoked in it.

Chances are that you were aware of that information, without needing to run a command to tell you it, so let's try and put some flesh on that bare bones information.

```
$ ps -f
```

```
UID        PID  PPID  C STIME TTY          TIME CMD
vlad     12396 25397  0 14:28 pts/0    00:00:00 ps -f
vlad     25397 25396  0 12:49 pts/0    00:01:39 bash
```

In case you haven't had time to do a `man ps` yet, be aware that the `-f` flag doesn't stand for "flesh on the bones" but for "Do full-format listing", although even then, there are "fuller" versions of the `ps` output.

But what are we being told here?

Every process has a unique process id (PID). Remember, this is a property of the process, not of the program that process is executing: if you are running three instances of your browser at once, each will have its own process ID.

The third column in this listing, PPID, shows the ID of each process's parent. Every process on a computer is spawned by another, which is its parent (except, of course, for the bootstrap process that runs automatically when the computer starts up).

Clearly, the `ps -f` that was run is a child process of the ( `bash` ) shell it was invoked in.

Column 1 shows the username of the user the processes are being run by. This is the username the computer uses when checking permissions: each process is allowed to access exactly the same things as the user running it, no more, no less.

Column 5, STIME, shows when the process started running, whilst Column 7, TIME, shows you how much time process has used, whilst Column 8, CMD, shows what program the process is executing.

Column 6, TTY, shows the ID of the terminal this process is running in. Once upon a time, this really would have been a terminal connected to a central timeshared computer. It isn't as important these days, except that if a process is a system service, such as a network monitor, `ps` will display a question mark for its terminal, since it doesn't actually have one.

The fourth column, C, is an indication of the perCentage of processor utilization.

Your version of `ps` may show more or fewer columns, or may show them in a different order, but the same information is generally available everywhere, and the column headers are generally consistent.

## Stopping, pausing, resuming, and backgrounding, processes

The shell provides several commands for stopping, pausing, and resuming processes. To see them in action, let's run our `analyze` program on our latest data files. After a few minutes go by, we realize that this is going to take a while to finish. Being impatient, we kill the process by typing Control-C. This stops the currently-executing program right away. Any results it had calculated, but not written to disk, are lost.

```
$ ./analyze results*.dat
```

```
...a few minutes pass...
^C
```

Let's run that same command again, with an ampersand `&` at the end of the line to tell the shell we want it to run in the background (../reference/#background):

```
$ ./analyze results*.dat &
```

When we do this, the shell launches the program as before. Instead of leaving our keyboard and screen connected to the program's standard input and output, though, the shell hangs onto them. This means the shell can give us a fresh command prompt, and start running other commands, right away. Here, for example, we're putting some parameters for the next run of the program in a file:

```
$ cat > params.txt
density: 22.0
viscosity: 0.75
^D
```

(Remember, \^D is the shell's way of showing Control-D, which means "end of input".) Now let's run the `jobs` command, which tells us what processes are currently running in the background:

```
$ jobs
```

```
[1] ./analyze results01.dat results02.dat results03.dat
```

Since we're about to go and get coffee, we might as well use the foreground command, `fg`, to bring our background job into the foreground:

```
$ fg
```

```
...a few minutes pass...
```

When `analyze` finishes running, the shell gives us a fresh prompt as usual. If we had several jobs running in the background, we could control which one we brought to the foreground using `fg %1`, `fg %2`, and so on. The IDs are *not* the process IDs. Instead, they are the job IDs displayed by the `jobs` command.

The shell gives us one more tool for job control: if a process is already running in the foreground, Control-Z will pause it and return control to the shell. We can then use `fg` to resume it in the foreground, or `bg` to resume it as a background job. For example, let's run `analyze` again, and then type Control-Z. The shell immediately tells us that our program has been stopped, and gives us its job number:

```
$ ./analyze results01.dat
^Z
```

```
[1]  Stopped   ./analyze results01.dat
```

If we type `bg %1`, the shell starts the process running again, but in the background. We can check that it's running using `jobs`, and kill it while it's still in the background using `kill` and the job number. This has the same effect as bringing it to the foreground and then typing Control-C:

```
$ bg %1

$ jobs
```

```
[1] ./analyze results01.dat
```

```
$ kill %1
```

Job control was important when users only had one terminal window at a time. It's less important now: if we want to run another program, it's easy enough to open another window and run it there. However, these ideas and tools are making a comeback, as they're often the easiest way to run and control programs on remote computers elsewhere on the network. This lesson's ssh episode (../02-ssh/) has more to say about that.

## Key Points

- When we talk of 'job control', we really mean 'process control'
- A running process can be stopped, paused, and/or made to run in the background
- A process can be started so as to immediately run in the background
- Paused or backgrounded processes can be brought back into the foreground
- Process information can be inspected with `ps`

(../05-directory-structure/)

(../07-aliases/)