Job Control

In this lesson, you'll learn what job control is, and how to manage and run multiple jobs in your shell.

WE'LL COVER THE FOLLOWING

How Important is this Lesson?

Job control is a core feature of bash, and considered a central concept to understand if you are using bash every day.

Starting Jobs

You're going to look at running a simple job using the `sleep` command.

Type this in:

```
sleep 60 &
```

Type the above code into the terminal in this lesson.

Terminal

You typed a 'normal' command (`sleep 60`) and then added in another character, the ampersand (`&`). The ampersand will run the command 'in the background', which becomes a *job* in this bash session.

The *job* has two identifiers, which are immediately reported to you in the terminal.

```
[1] 39165
```

The first is the *job number*, which in this case is `1`, and the second is the *process identifier*, in this case `39165`, but this will be different for you.

If, before the time is up, you run any other commands:

```
pwd
```

Type the above code into the terminal in this lesson.

then they are not interfered with by this running job. You can just carry on as normal.

If you wait for the program to finish (60 seconds in this case), and then run any other command, then bash will report to you what happened to that job. For example, if you wait, and type:

```
true
```

Type the above code into the terminal in this lesson.

You'll see:

```
[1]+  Done                    sleep 60
```

in the terminal.

Again, it reports the job number, this time with the status (`Done`), and the command that was originally run (`sleep 60`).

Controlling Jobs

Just like starting jobs, you can control jobs by sending signals to them.

Here you're going to start two jobs, one to sleep for two minutes, and the next

for one second more (so we can distinguish between them).

```
sleep 120 &
sleep 121 &
```

Type the above code into the terminal in this lesson.

Now you have two jobs running in the background. You can find out what their status is using the `jobs` builtin:

```
jobs
```

Type the above code into the terminal in this lesson.

and you should see this:

```
[1]-  Running                 sleep 120 &
[2]+  Running                 sleep 121 &
```

Each job is identified by its number in square brackets, followed by a plus or a minus sign. The plus sign indicates which job will be brought to the foreground if you run the `fg` builtin.

Let's do that here:

```
fg
```

Type the above code into the terminal in this lesson.

This time you should see:

```
sleep 121
```

The `sleep 121` process is now running in the foreground. Bash lets you know this by outputting the foregrounded command to the terminal.

Now send the `STOP` signal to the foregrounded process by hitting `\C-z`.

```
[2]+  Stopped                 sleep 121
```

Again, you see the job number, the plus sign, and the status (`Stopped`). If we

want the process to continue to run, then you need to use the `bg` builtin:

```
bg
```

Type the above code into the terminal in this lesson.

Now you should see:

```
[2]+ sleep 121 &
```

The process has been continued from its stopped state. If you've taken over two minutes to do all this then you may get a report that the process has terminated with a message that they are '`Done`'. If so, start the two sleep commands again and get back to here.

Now that you have the two sleeps running in the background, you can send other signals to them. For example, you can kill them:

```
kill %1
```

Type the above code into the terminal in this lesson.

```
[1]+  Terminated: 15          sleep 120
```

The per cent sign followed by a number is called a *job specification* and is the way you can tell bash you want to operate on that job number. Of course, you can also send a signal by using the process identifier as well.

Waiting

One technique that can be useful in bash scripts is to start a number of background processes, and then wait for them to finish before continuing. This can be done with the `wait` builtin.

If your original sleep is still running, then you can run wait.

```
wait
```

Type the above code into the terminal in this lesson.

Eventually, the wait command returns with a '`Done`' status reported for that

Eventually, the wait command returns with a `Done` status reported for that process.

```
[1]+  Done                    sleep 120
```

Normally, wait always returns an exit code of zero. But if you add a job specification, wait returns the exit code of the last-completed job.

```
sleep 20 &
sleep 30 && false &
wait %1 %2
```

Type the above code into the terminal in this lesson.

So what exit code will that `wait` report? Check with:

```
echo $?
```

Type the above code into the terminal in this lesson.

Challenge: What if you swapped the sleep time numbers 20 and 30 above?

1 One of the outputs when backgrounding a process is

What You Learned

- *Job control* using signals

- *Job specifications*

- The `fg` and `bg` builtins
- The `wait` builtin

What Next?

Next we look at *traps* and *signals*.

Exercises

1) Write a script to run a series of useful housekeeping or reporting commands in the background, and then output a summary message at the end when they have all completed.