**Job Control**

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/q2n0VqQ5lkR#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 [#](https://www.educative.io/courses/master-the-bash-shell/q2n0VqQ5lkR#starting-jobs)

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

Type this in:

1

sleep 60 &





Type the above code into the terminal in this lesson.

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:

1

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:

1

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 [#](https://www.educative.io/courses/master-the-bash-shell/q2n0VqQ5lkR#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).

1

2

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:

1

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:

1

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:

1

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:

1

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 [#](https://www.educative.io/courses/master-the-bash-shell/q2n0VqQ5lkR#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.

1

wait





Type the above code into the terminal in this lesson.

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.

1

2

3

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:

1

echo $?





Type the above code into the terminal in this lesson

# Traps and Signals

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/7AlP9319AkQ#how-important-is-this-lesson)

**Traps** are an advanced concept. If you’re new to bash you might want to follow this lesson to be aware of it, and apply it as you get more knowledge of Linux or go deeper into bash scripting.

### Triggering signals [#](https://www.educative.io/courses/master-the-bash-shell/7AlP9319AkQ#triggering-signals)

Any easy way to trigger a **signal** is one you will likely already have used.

Follow the instructions here:

1

sleep 999





Type the above code into the terminal in this lesson.

Now hit the CTRL key down, hold it, and then hit the c key (CTRL-c). Then get the exit code:

1

echo $?





Type the above code into the terminal in this lesson.

You should have got the output of a number over 128. You will of course remember that $? is a special variable that gives the exit code of the last-run command.

What you are less likely to have remembered is that exit codes over 128 indicate that a signal triggered the exit, and that to determine the signal number you take 128 away from the number you saw.

Bonus points if you did remember!

Can you work out what the signal was that stopped the sleep command?

The signals are usually documented in the signal man page.

1

2

man signal

man 7 signal





Type the above code into the terminal in this lesson.

Show Hint

Note: man pages have different sections. man man will explain more if you’re interested, but to get a specific section, you put the section number in the middle, as above. Find out what section 7 is by reading man man. You might not have section 7 of the signal man page installed.

If the signals are not listed on the man pages on the lesson terminal, then google them!

Now figure out what the signal was, what the default action is for that signal and the signal name that is triggered when you hit CTRL-c.

1

sleep 999





Type the above code into the terminal in this lesson.

Now hit the CTRL key down, hold it, and then hit the z key (CTRL-z). Then get the exit code:

1

echo $?





Type the above code into the terminal in this lesson.

Challenge: which signal does CTRL-z trigger?

### The kill Command [#](https://www.educative.io/courses/master-the-bash-shell/7AlP9319AkQ#the-kill-command)

Another way to send a signal to a process is another one you have also likely come across: the kill command.

The kill command is misnamed, because it needn’t be used to terminate a process. By default, it sends the signal 15 (TERM), which (similar to 2) usually has the effect of terminating the program, but as the name suggests, is a stronger signal to terminate than INT (interrupt).

1

2

3

4

5

6

7

sleep 999 &

KILLPID=$(echo ${!})

echo ${KILLPID}

kill -2 ${KILLPID}

echo ${?}

wait ${KILLPID}

echo ${?}





Type the above code into the terminal in this lesson.

Note: The curly braces are required with the ${!} (which surprised me!). Bash interprets the ! as being a history command (try it!). I’m not sure why (it works fine outside the $()), but it is an indication that it’s perhaps wise to get into the habit of putting curly braces around your variable names in bash.

Can you explain why the echo after the kill outputs 0 and not 130?

Show Hint

Instead of -2 in the above listing, you can use the signal name. Either -INT or -SIGINT will work. Try them.

### Trapping Signals [#](https://www.educative.io/courses/master-the-bash-shell/7AlP9319AkQ#trapping-signals)

Type out this first:

1

while :; do sleep 5; done





Type the above code into the terminal in this lesson.

Now hit CTRL-c. The while loop will stop. Now create a similar-looking file with an extra line:

1

2

3

4

5

6

7

cat > trap\_exit.sh << END

#!/bin/bash

trap "echo trapped" INT

while :; do sleep 5; done

END

chmod +x trap\_exit.sh

./trap\_exit.sh # NOW HIT CTRL-c





Type the above code into the terminal in this lesson.

What’s going on? In the second listing you used the trap builtin to inhibit the default response of the trap\_exit process in the bash process and replace it with another response. In this case, the first argument to the trap builtin is evaluated and run as a command (echo trapped).

So how to get out of it and kill off the process?

First, hit CTRL-z, and then type:

1

kill %1





Type the above code into the terminal in this lesson.

### Trap Exit [#](https://www.educative.io/courses/master-the-bash-shell/7AlP9319AkQ#trap-exit)

In addition to the normal signal name traps listed in the man 7 signal file, there are some special ones.

Type this out:

1

2

3

4

5

6

7

8

9

cat > trap\_exit.sh << END

#!/bin/bash

trap "echo trapped" EXIT

sleep 999

END

chmod +x trap\_exit.sh

./trap\_exit.sh &

TRAP\_EXIT\_PID=${!}

kill -15 ${TRAP\_EXIT\_PID}





Type the above code into the terminal in this lesson.

* **Line 1-6** uses a here doc to create the trap\_exit.sh script and then make it executable
* **Line 7** runs the script in the background
* **Line 8** uses the ${!} variable to retrieve the backgrounded process identifier
* **Line 9** sends the TERM signal to the trap\_exit.sh process

Which signal did we use there?

The EXIT trap catches the termination of the script and runs. Try it with -2 as well.

Now run this:

1

2

3

./trap\_exit.sh &

TRAP\_EXIT\_PID=${!}

kill -9 ${TRAP\_EXIT\_PID}





Type the above code into the terminal in this lesson.

Some of the signals are not trap-able! Why do you think this is?

Show Hint

Experiment with some other signals to determine how EXIT handles them.

What is the name of the -9 signal? Is this the default that the kill command uses?

Show Hint

### A Note About Process Groups [#](https://www.educative.io/courses/master-the-bash-shell/7AlP9319AkQ#a-note-about-process-groups)

You may have noticed that in the above script you used the wait command after putting the process in the background.

The wait command is a bash builtin that returns when the child processes of the bash process completes.

This illustrates a subtle point about signals. They act on the currently running process, and not on their children.

Repeat the above exercise, but rather than having:

    sleep 999 &  
    wait

type:

    sleep 999

in your script.

What do you notice about the behaviour of the EXIT and INT signals?

How do you explain the fact that running this:

1

./trap\_exit.sh





Type the above code into the terminal in this lesson.

and then hitting CTRL-c works to kill the sleep process and output ‘trapped’, where sending the signal -2 before did not?

The answer is that foregrounded processes are treated differently - they form part of a ‘process group’ that gets any signals received on the terminal.

If this seems complicated, just remember: CTRL-c kills all the processes ’going on’ in the foreground of the terminal with the 2 (or INTerrupt) signal, while kill sends a message to a specific process, which may or may not be running at the time.

If this seems complicated, just remember: signals can get complicated!

**Debugging**

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/qAvJmXAkWo7#how-important-is-this-lesson)

If you write, use or maintain bash of any complexity you’ll want to know how to debug it!

### Syntax Checking Options [#](https://www.educative.io/courses/master-the-bash-shell/qAvJmXAkWo7#syntax-checking-options)

Start by creating this simple (but broken) script:

1

2

3

4

5

6

cat > debug\_script.sh << 'END'

#!/bin/bash

A=some value

echo "${A}

echo "${B}"

END





Type the above code into the terminal in this lesson.

Now run it with the -n flag. This flag only parses the script, rather than running it. It’s useful for detecting basic syntax errors.

1

bash -n debug\_script.sh





Type the above code into the terminal in this lesson.

You can see it’s broken. Fix it. Then run it:

1

bash debug\_script.sh





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 [#](https://www.educative.io/courses/master-the-bash-shell/qAvJmXAkWo7#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).

1

bash -v debug\_script.sh





Type the above code into the terminal in this lesson.

Try tracing to see more details about what’s going on. Each statement gets a new line.

1

bash -x debug\_script.sh





Type the above code into the terminal in this lesson.

Using these flags together can help debug scripts where there is an elementary error, or even just working out what’s going on when a script runs. I used it only yesterday to figure out why a systemctl service wasn’t running or logging.

Fix the error you see before continuing.

### Managing Variables [#](https://www.educative.io/courses/master-the-bash-shell/qAvJmXAkWo7#managing-variables)

Variables are a core part of most serious bash scripts (and even one-liners!), so managing them is another important way to reduce the possibility of your script breaking.

Change your script to add the ‘set’ line immediately after the first line and see what happens.

#!/bin/bash  
set -o nounset  
A="some value"  
echo "${A}"  
echo "${B}"

Now research what the nounset option does. Which set flag does this correspond to?

Show Hint

Without running this, try and figure out what this script will do. Will it run?

#!/bin/bash  
set -o nounset  
A="some value"  
B=  
echo "${A}"  
echo "${B}"

Try it and see.

I always set nounset on my scripts as a habit. It can catch many problems before they become serious.

### Profiling Bash Scripts [#](https://www.educative.io/courses/master-the-bash-shell/qAvJmXAkWo7#profiling-bash-scripts)

Returning to the xtrace (or set -x) flag, we can exploit its use of a PS variable to implement the profiling of a script:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

#!/bin/bash

set -o nounset

set -o xtrace

declare A="some value"

PS4='$(date "+%s.%N => ")'

B=

echo "${A}"

A="another value"

echo "${A}"

echo "${B}"

ls

pwd

curl -q bbc.co.uk

true





Type the above code into the terminal in this lesson.

* **Lines 2-3** set the nounset and xtrace options.
* **Line 4** declares a variable A and gives it a value.
* **Line 5** sets the PS4 variable, which runs before each command is processed in bash, and outputs the result prepended by a + sign. From this point in the script, each line shows the time it was run to nanosecond granularity, allowing you to see where the time running the script goes.
* **Lines 6-14** runs various simple commands that exercise the ability to see how long each line runs for.

Note: If you are on a Mac, then you might only get second-level granularity on the date!

### Shellcheck [#](https://www.educative.io/courses/master-the-bash-shell/qAvJmXAkWo7#shellcheck)

Finally, here is a very useful tip for understanding bash more deeply and improving any bash scripts you come across. Shellcheck is a website (<http://www.shellcheck.net/>) and a package that gives you advice to help fix and improve your shell scripts. Very often, its advice has prompted me to research more deeply and understand bash better.

Here is some example output from a script I found on my laptop:

$ shellcheck shrinkpdf.sh  
In shrinkpdf.sh line 44:  
          -dColorImageResolution=$3             \  
                                 ^-- SC2086: Double quote to prevent globbing and word splitting.  
   
In shrinkpdf.sh line 46:  
          -dGrayImageResolution=$3              \  
                                ^-- SC2086: Double quote to prevent globbing and word splitting.  
   
In shrinkpdf.sh line 48:  
          -dMonoImageResolution=$3              \  
                                ^-- SC2086: Double quote to prevent globbing and word splitting.  
   
In shrinkpdf.sh line 57:  
        if [ ! -f "$1" -o ! -f "$2" ]; then  
                       ^-- SC2166: Prefer [ p ] || [ q ] as [ p -o q ] is not well defined.  
   
In shrinkpdf.sh line 60:  
        ISIZE="$(echo $(wc -c "$1") | cut -f1 -d\ )"  
                      ^-- SC2046: Quote this to prevent word splitting.  
                      ^-- SC2005: Useless echo? Instead of 'echo $(cmd)', just use 'cmd'.  
   
In shrinkpdf.sh line 61:  
        OSIZE="$(echo $(wc -c "$2") | cut -f1 -d\ )"  
                      ^-- SC2046: Quote this to prevent word splitting.  
                      ^-- SC2005: Useless echo? Instead of 'echo $(cmd)', just use 'cmd'.

The most common reminders are regarding potential quoting issues, but you can see other useful tips in the above output, such as preferred arguments to the test construct, and advice on ‘useless’ echos.

### What You Learned [#](https://www.educative.io/courses/master-the-bash-shell/qAvJmXAkWo7#what-you-learned)

In this lesson, you learned:

* bash flags useful for debugging
* How to use traps and declare to trace the use of variables
* How to make your scripts more robust with nounset
* How to use shellcheck to help you reduce the risk of your scripts failing