Traps and Signals

In this lesson you will learn what a signal is, how the `kill` command can be used to send signals, and how signals can be trapped. In addition, you'll cover the 'wait' bash builtin, and what a 'process group' is.

WE'LL COVER THE FOLLOWING

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

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

Follow the instructions here:

```
sleep 999
```

Type the above code into the terminal in this lesson.

Terminal

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

```
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.

```
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`.

```
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:

```
echo $?
```

Type the above code into the terminal in this lesson.

Challenge: which signal does `CTRL-z` trigger?

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*).

```
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 `2` and not `130`?

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

Type out this first:

```
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:

```
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:

```
kill %1
```

Type the above code into the terminal in this lesson.

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:

```
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:

```
./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

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

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:

```
./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 `INT`errupt) 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!

Traps and Signals Quiz

1 What is `kill`'s default signal?

COMPLETED 0%

1 of 2

What You Learned

In this lesson you have learned:

- What a *signal* is
- What a *trap* is
- What the `kill` program does, and that it doesn't send `KILL` by default
- What an `INT` and `TERM` signal is
- How to *trap* exiting bash processes
- What a *process* group is, and its significance for *signals*

What Next?

Next we look at various methods used to debug bash scripts.

Exercises

1) Write a shell script that you can't escape from in the terminal provided above

2) Try and escape from the shell script you created in 1)

3) Ask everyone you know if they can escape the shell script

4) If no-one can escape it, send it to the author :)

5) Research the other 'special' signal traps. Use `man bash` for this.