Exercise 1: Signals in bash

In this exercise, we're gonna look at how bash handles and works with signals in its child processes. There's a handful of really useful signals that we'll look at; `SIGTERM`, `SIGKILL`, `SIGTSTP`, `SIGCONT`, `SIGHUP` and `SIGCHLD`. We'll discuss what they do, when they're used, and how to get around their effects (and when you can't).

Setup

For this exercise, you'll be using the `pementorship/w6_ex1` image. We'll need some special permissions to be able to run `strace` with the `-p` flag, so run:

```
sudo docker run -it --cap-add=SYS_PTRACE pementorship/w6_ex1
```

For this exercise, we'll be using tmux. If you're not familiar with it, we'll put the commands needed to navigate and move around here, but we strongly advise you take a look at this short tutorial first.

Following signals

First thing you need to do is run `make` from the home directory. This will build all the C code that we'll need to run in these exercises. The first one we'll run is the binary called `./loop`. Run it in your first window. Notice all it does is print out hello over and over again:

```
[me@cd33cd660544:~] $ ./loop
Hello: 0
Hello: 1
Hello: 2
Hello: 3
[...]
```

Let's try attaching strace to it. Press the keys `Ctrl+b %`. This will create a new vertical split in the window with a new terminal. You can move between then with `Ctrl+b ←` and `Ctrl+b →`. On the new terminal, run `ps axf`. This will print out a **process tree**. Basically, a tree-like representation of the processes running on the container. Each process has a parent, which is usually the process that created it. We'll talk more about this in a future module. For now, you should see something like this:

```
[me@276fea436a5a:~] $ ps axf
  PID TTY      STAT   TIME COMMAND
    1 pts/0    Ss+    0:00 /usr/bin/tmux
    7 ?        Ss     0:00 /usr/bin/tmux
    8 pts/1    Ss     0:00  \_ -bash
   57 pts/1    S+     0:00  |   \_ ./loop
   28 pts/2    Ss     0:00  \_ -bash
   58 pts/2    R+     0:00      \_ ps axf
```

Interestingly, you can see two `tmux` processes here. We're going to focus on the one with all the child processes. This is the instance of tmux you're seeing on your terminal. Notice there's two instances of bash (your shell), and each one's running a program, one has the `./loop`, and the other has the instance of ps that output the tree.

Copy or keep track of the PID of the program running `./loop` (here 57). Now, on the second terminal, we'll strace it by running `sudo strace -p 57`:

```
[me@cd33cd660544:~] $ sudo strace -p 57
strace: Process 57 attached
restart_syscall(<... resuming interrupted read ...>) = 0
write(1, "Hello: 17\n", 10)             = 10
nanosleep({tv_sec=1, tv_nsec=0}, 0x7ffd878626b0) = 0
write(1, "Hello: 18\n", 10)             = 10
nanosleep({tv_sec=1, tv_nsec=0}, 0x7ffd878626b0) = 0
[...]
```

As you can see, we're basically making two syscalls over and over again, `write()` and `nanosleep()`, which just tells us we're writing to the terminal, and sleeping for a second, over and over again. However, this is kind of noisy, we really only care about the signals, not the syscalls, so you can filter for only signals using the `-e signal` flag:

```
[me@cd33cd660544:~] $ sudo strace -e signal -p 57
strace: Process 57 attached
```

Stopping, pausing, and resuming programs

Bash provides you ways of sending useful signals to currently running processes. Before we look at those though, we need to look at a quick concept: foreground and background processes.

Bash lets you run programs in two different modes, foreground and background mode. When you run a program like we have done so far, it will default to running in foreground mode. What that means is that, while the program is running, the shell is locked up and you cannot run any other commands. Bash will also redirect anything you type in to the standard input of the foreground process.

On the flip side, if you add a & at the end of any commands you run in bash, it'll run them in the background. The process will still be a child of bash, but bash will let you run other programs, and won't redirect things you type to standard input. That said, you'll still see the output of the program in your shell.

Pausing and resuming processes

Now that we have the concept out of the way, let's talk about how you would apply this. One think people commonly want to do in bash is turn a foreground process into a background one. To do this, bash provides the bg command which, if you provide a process, it will start running it in the background. However, to do this with a running process, we need to somehow pause it so we can type in the command. For this, we can run `Ctrl+Z`. This will pause execution of the foreground process (we'll see how in a sec), and print out the job ID of the process (think of the job ID as a process ID that identifies programs inside an instance of bash instead of across a whole system). Then we can run `bg %1`, where 1 here is the job ID we mentioned earlier:

```
[...]
Hello: 235
Hello: 236
Hello: 237
^Z
[1]+  Stopped                 ./loop
[me@276fea436a5a:~] $ bg %1
[1]+ ./loop &
Hello: 238
```

```
[me@276fea436a5a:~] $ Hello: 239
Hello: 240
Hello: 241
Hello: 242
Hello: 243
Hello: 244

[me@276fea436a5a:~] $ Hello: 245
Hello: 246
Hello: 247
```

Notice that after we resume the job, it'll continue printing into the terminal, but you now get a prompt you can write commands with!

Now, let's take a look at what happened in strace:

```
[me@276fea436a5a:~] $ sudo strace -e signal -p 57
strace: Process 57 attached
--- SIGTSTP {si_signo=SIGTSTP, si_code=SI_KERNEL} ---
--- stopped by SIGTSTP ---
--- SIGCONT {si_signo=SIGCONT, si_code=SI_USER, si_pid=8, si_uid=1000} ---
```

Notice two signals got sent. `SIGTSTP` and `SIGCONT`. The first was sent when you pressed `Ctrl+Z`, while the second got sent when we ran `fg`. `SIGTSTP` is a special signal that makes a processs stop running. It's one of the few signals that you cannot handle (in other words, you can't change the behaviour of receving the signal). It will always, unconditionally, pause the process. If you follow it up with a `SIGCONT` (which is what happened) it'll make the process resume execution normally.

Killing processes

This program is probably getting a bit annoying, so let's run `kill` to stop it. I'll open a new split with `Ctrl+b "` but you can do it on the same window as the annoying program is running. It is in the background after all:

```
[me@276fea436a5a:~] $ kill 57
[me@276fea436a5a:~] $
```

Not much seems to have happened here, but if you look at the place where we were running the loop program, you'll see it stopped printing entirely. Now, look at the window that was running `strace` on that program:

```
[...]
--- SIGTERM {si_signo=SIGTERM, si_code=SI_USER, si_pid=86, si_uid=1000} ---
+++ killed by SIGTERM +++
[me@276fea436a5a:~] $
```

So, running the `kill` command will send a `SIGTERM` signal to the process in question, and it will subsequently be killed. Interestingly , the `kill` command can be used to send any signal, not only `SIGTERM`. Confusingly, not every signal it sends will kill the target process. Read `man kill` to learn more about how to send those signals.

Now, to some more complex killing techniques. Let's start up `./loop` again and attach `strace` to it. This time, on the `./loop` window, hit `Ctrl+C`:

```
[me@276fea436a5a:~] $ ./loop
Hello: 0
Hello: 1
[...]
Hello: 30
Hello: 31
^C
[me@276fea436a5a:~] $
```

It'll kill the process. Let's go to the strace output to see how it did it:

```
[me@276fea436a5a:~] $ sudo strace -e signal -p 106
strace: Process 106 attached
--- SIGINT {si_signo=SIGINT, si_code=SI_KERNEL} ---
+++ killed by SIGINT +++
[me@276fea436a5a:~] $
```

This time, we see `SIGINT` instead. This is a different signal that can be used to terminate processes, and generally works the same way as `SIGTERM`, but is used for "interrupts from keyboard".

Forced kill

Let's take a look at another program, `block`. You should see it in the same folder as everything else. Let's run it and try to kill it with `Ctrl+C`:

```
[me@8813f29d6ee1:~] $ ./block
^CRecived SIGINT; Ignoring...
^CRecived SIGINT; Ignoring...
^CRecived SIGINT; Ignoring...
^CRecived SIGINT; Ignoring...
```

That's odd... It's ignoring you. Let's kill it by sending a `SIGTERM` from the other tmux split we had open using `kill`:

```
[me@8813f29d6ee1:~] $ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
[...]
me          39 88.4  0.0   4328   668 pts/1    R+   11:59   0:06 ./block
[...]
[me@8813f29d6ee1:~] $ kill 39
```

```
[me@8813f29d6ee1:~] $ ./block
[...]
Recived SIGTERM; Ignoring...
```

This brings us to an interesting fact about `SIGTERM` and `SIGINT`: they're "handleable" functions. That means you can override the default action of the signal to do whatever you want. These are usually called signal handlers, we'll talk about this a bit more in the next exercise.

Let's talk about how to kill the process when you find yourself in a situation like this. It's very common for programs to override the handlers for `SIGTERM` and `SIGINT` to do cleanup before killing the process, but often times, programs can get so stuck that even the cleanup ends up stuck and the program never terminates, so for this we need to sent `SIGKILL`. `SIGKILL` is another signal that cannot be interrupted or handled, and will always, unconditionally and immediately kill processes. This can be often dangerous (it might kill a process while it's doing critical work, or leave a file half-written and so currupted), but desperate times call for deparate measures.

You could send this signal, but let's send it from the same terminal, using that `SIGTSTP` trick we learned earlier (reminder, use `Ctrl+Z`):

```
[me@8813f29d6ee1:~] $ ./block
^CRecived SIGINT; Ignoring...
^CRecived SIGINT; Ignoring...
^CRecived SIGINT; Ignoring...
^CRecived SIGINT; Ignoring...
Recived SIGTERM; Ignoring...
^Z
[1]+  Stopped                 ./block
[me@8813f29d6ee1:~] $ kill -9 %1
[1]+  Killed                  ./block
[me@8813f29d6ee1:~] $
```

Notice we used the `-9` flag in `kill`. This will send a `SIGKILL` (since 9 is the code used for said signal). You could have also used `-SIGKILL`.

The death of bash

One interesting thing you'll notice is that every program you run in bash seems to be attached to bash in some way. That's where it's printing out to, job id's identify it as part of that bash instance... One of the most interesting aspects of this connection between bash and the processes you run inside of it is is what happens to the child processes when bash dies. Let's take a look at it. Start up `loop` again on one terminal and attach strace on the other:

```
[me@ac1f6644c90f:~] $ ./loop
Hello: 0
Hello: 1
Hello: 2
[...]
```

```
[me@ac1f6644c90f:~] $ sudo strace -p 61
strace: Process 61 attached
restart_syscall(<... resuming interrupted read ...>) = 0
write(1, "Hello: 48\n", 10)             = 10
nanosleep({tv_sec=1, tv_nsec=0}, 0x7ffec1d06a50) = 0
write(1, "Hello: 49\n", 10)             = 10
[...]
```

Now, go back to the window running `loop` and press `Ctrl+b x`. This will close the window and kill the bash process. You'll need to confirm by typing `y`. If you look at the `strace` output, you should see something like this:

```
[me@ac1f6644c90f:~] $ sudo strace -p 61
[...]
--- SIGHUP {si_signo=SIGHUP, si_code=SI_USER, si_pid=8, si_uid=1000} ---
+++ killed by SIGHUP +++
[me@ac1f6644c90f:~] $
```

Now we see yet another signal: `SIGHUP`. This one's used to signal when a terminal "hangs up" (this probably tells you everything you need to know about how old some of these things are), and is used to tell child processes of a terminal that their parent died, and they should die too. This is the sole mechanism via which terminals handle killing child processes on termination.