Exercise 2: Signal handlers

Something we touched on briefly in the last exercise was signal handlers. Most signals let you write a handler for them that executes a piece of code instead of whatever the default behaviour is for the signal is. A common use-case for signals is to communicate messages between processes. The signal is used to tell the other process that an event happened. Because of this, you'll often see signals reffered to as a form of *Inter-Process Communication*, or IPC. In this topic, it's important to note that there's two signals you'll never be able to handle: `SIGKILL` and `SIGSTOP`.

Setup

Much like before, you can find the image for thid exercise in w6_ex2, but you should run it in the following way to allow `strace` to run properly:

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

Clean-up on end

A common usecase for signal handlers is to override what happens before a process gets a signal that would usually result in it getting killed, and perform some cleanup (or, in some cases, ignore it completely). Let's take a look at `cleanup.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdbool.h>

bool running = true;
int counter = 0;

void cleanup_exit(int signal) {
    running = false;
}

int main() {
    // Setup signal handlers
    struct sigaction sig_action;
    sig_action.sa_handler = cleanup_exit;
    sigemptyset(&sig_action.sa_mask);
    sig_action.sa_flags = 0;

    sigaction(SIGINT, &sig_action, NULL);
    sigaction(SIGTERM, &sig_action, NULL);

    while(running) {
        printf("Working...\n");
        counter += 1;
        printf("Counter is %d\n", counter);
        sleep(1);
    }
    printf("Cleaning up...\n");
    printf("Writing result to \"results.txt\"\n", counter);
    FILE* f = fopen("results.txt", "w");
    fprintf(f, "Iterations: %d\n", counter);
    fclose(f);
    printf("Cleanup finished! Exiting\n", counter);

    return 0;
}
```

Take a bit of time to read through the code and try to understand what's going on. We'll do a breakdown below.

NOTE: Remember to run make

First, let's run it to see what it's doing:

```
[me@3a0ecc94475c:~] $ ls
Makefile  cleanup  cleanup.c  kill_ign  kill_ign.c  stop_ign  stop_ign.c
[me@3a0ecc94475c:~] $ ./cleanup
Working...
Counter is 1
Working...
Counter is 2
Working...
Counter is 3
Working...
Counter is 4
^CCleaning up...
Writing result to "results.txt"
Cleanup finished! Exiting
[me@3a0ecc94475c:~] $ ls
Makefile  cleanup  cleanup.c  kill_ign  kill_ign.c  results.txt  stop_ign  stop_ign.
[me@3a0ecc94475c:~] $ cat results.txt
Iterations: 4
```

As you can see, it counts up and when you press `Ctrl+C` to send a `SIGINT` (signaled by the `^C` before the `"Cleaning up..."`), it starts doing cleanup by writing the number of iterations that were acheieved to `results.txt` in the current directory.

Let's look again at how it's achieved:

```
// Setup signal handlers
struct sigaction sig_action;
sig_action.sa_handler = cleanup_exit;
sigemptyset(&sig_action.sa_mask);
sig_action.sa_flags = 0;

sigaction(SIGINT, &sig_action, NULL);
sigaction(SIGTERM, &sig_action, NULL);
```

Here, we setup the signal handlers. We do this making use of the `sigaction()`, which lets you specify what to do when a system call is received. In this case, we're simply setting a handler for the syscall, but you can set special parameters to make the signal be completely ignored, or to revert to the "default" handler for a signal.

Ignoring SIGSTOP or SIGKILL

I've created two programs, `stop_ign.c` and `kill_ign.c` (both of these got build into `stop_ign` and `kill_ign` when you ran make earlier). They make a call to `sigaction()` to ignore the SIGSTOP and SIGKILL signals respectively. This means that, if we can handle the signals, we should be able to ignore those signals and continue running normally:

```
[me@08679f3ca1bb:~] $ ./kill_ign
Working...
Counter is 1
Working...
Counter is 2
[...]
Working...
Counter is 17
Killed
[me@08679f3ca1bb:~] $
```

That killed happened as we sent a SIGKILL from a second tmux pane:

```
[me@08679f3ca1bb:~] $ kill -9 44
[me@08679f3ca1bb:~] $
```

Interesting... So our program did not ignore the signal. Let's see what happens with SIGSTOP in the other program:

```
[me@08679f3ca1bb:~] $ ./stop_ign
Working...
Counter is 1
Working...
Counter is 2
[...]
Working...
Counter is 11

[1]+  Stopped                 ./stop_ign
[me@08679f3ca1bb:~] $
```

With the other tmux pane looking like this:

```
[me@08679f3ca1bb:~] $ kill -SIGSTOP 84
[me@08679f3ca1bb:~] $
```

The lesson here is, no matter what you do, you cannot ignore these two signals. But wait, did the OS just let us execute these two `sigaction()` calls without telling us they're illegal? Well, not quite. If you read `man 2 sigaction` you will see the following near the bottom of the section on `sigaction()`:

```
RETURN VALUE
       sigaction() returns 0 on success; on error, -1 is returned, and errno is set
       indicate the error.
```

This is a common way of signaling error on syscalls: by placing it in the return value and using `errorno` to set the failure (you can take a look at `man errorno` for more information on how to figure out failures during syscalls, as well as a full listing of all error numbers, including some interesting notes on common pitfalls when using it).