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`:

```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;
}
```