Supplement: UNIX Signals

Leonard König, April 12, 2021

Inter-Process Communication (IPC)

One of the most crucial tasks of an OS kernel is to separate running programs into isolated processes that cannot interfere with each others execution. Ideally, the processes do not even know whether other processes are running.

However, in reality, we want to have processes communicate and cooperate with each other—but confined to small, controlled channels of communication. Inter-Process Communication is a broad categery of techniques our OS provides to do just that. For example, a process *A* may ask the OS to deliver a message to process *B*. Likewise, the process *B* might answer with another message.

In this exercise, we look at a specific IPC technology called *signals*. There is a fixed number of these signals which can be sent from one process to another to, e.g., signal the request for process termination (`SIGTERM`) or interruption (`SIGINT`).[1]

The receiving process can catch those signals. It can comply with the request (often process termination in the above two cases), ignore them, or do something wholly different. We will now write a program called `evil` which runs endlessly doing nothing and but will catch both of these signals and chooses to ignore them.

Afterwards, we write a small program called `kill`, modeled after UNIX `kill(1)` utility, which can send arbitrary signals to a given process. We can observe, that it's not able to terminate our 'evil' process using the mentioned signals.

[1] Ever wondered what the 'red x' symbol in the upper right corner of the window border did? It sends `SIGTERM` to the process running in that window!

An evil plan

Catching a signal consists of two steps: Writing a function that is to be executed when the signal is received, and actually registering this function with our operating system.

Our function has to have the signature `void foo(int)` as this is the fixed signature for *every* signal handler. Let's write one:

```
void my_signal_handler(int signal_number)
{
        (void)signal_number;
}
```

Right now, this function does precisely nothing. The only 'statement' it contains serves the only use to 'silence' the compiler complaining about the variable being unused.

In order to now register this function with our system, we need to call the `sigaction(2)` function. Unfortunately, in order for Linux to provide us with this function, we need to type

```
#define _POSIX_C_SOURCE 199309L
```

at the start of our C file *before* the header includes, as this is a feature of the POSIX version from the 9th of 1993.

The `sigaction` function takes three arguments, first, the signal to be caught, then a 'structure' defining which function is to be called (i.e., `my_signal_handler`), and finally an optional argument where we simply pass `NULL`.

Structures are the C way to build custom types by combining other types. Simply for reference, the structure that is passed to the `sigaction` function looks like:

```
// only for reference:
struct sigaction {
        void (*sa_handler) (int);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_sigaction)(int, siginfo_t *, void *);
}
```

It contains foru members of which we will ignore all but the first, the others will be set to zero or unset. The first member has type '*pointer to* a function taking `int` and returning `void`)—which matches our function definition of our signal catcher above!

Now, let's write our `main` function to register the signal handler:

```
#define _POSIX_C_SOURCE 199309L

// ...

int main(void)
{
        struct sigaction sa_register = {
                .sa_handler = &my_signal_handler,
                .sa_flags = 0,
        };
        sigemptyset(&sa_register.sa_mask);

        //TODO
}
```

We first define a new variable `sa_register` of type `struct sigaction` (the `struct` keyword is required!) and we immediately assign values to its members. As we don't have any special requirements, we can set `sa_flags` to 0 and we *must* leave `sa_sigaction` *unset*.[2] However, we must set `sa_mask` to empty via a special function, which takes the address of the member `sa_mask` of this variable `sa_register` and sets it to some implementation-defined 'empty' value.

We now are prepared to register our function for `SIGINT` (and `SIGTERM`):

```
int err = sigaction(SIGINT, &sa_register, NULL);
// error handling recommended but ommitted here.
```

Finally, we simply do an endless loop, thus not terminating the program at any point.

In the actual exercise you are also asked to print the process ID (PID) at startup, a timed message in the loop, as well as some text within the signal handler—but we leave that for you to figure out.

[2] In this specific notation and case, we could actually omit the line setting `sa_flags` (but not `sa_mask`). Furthermore, not setting `sa_sigaction` is required since it actually shares memory storage with `sa_handler`.
But this is a pecularity of C which you should only use if you are more acquainted to the language.

Killing all evil

The `kill` program takes an option with a number like `-2`, as well as the PID of the target process. That is, if our `evil` program has PID 1234, we could run

```
$ ./kill -2 1234
```

To send the signal `2` or `SIGINT` (as documented in `kill(1)`) to said program, triggering the signal handler. Alternatively, we can type `Ctrl+C` in the running `evil` program, which has the same effect.

So our program asks for two arguments, the first which is in `argv[1]` can be parsed using `sscanf`:

```
int n = sscanf(argv[1], "-%d", &sig);
```

Similarly, our PID in `argv[2]` can be parsed, however a PID is not an `int` but a `long int`, thus:

```
int m = sscanf(argv[2], "%ld", &pid);
```

The final step is to call the equally named *function* `kill(2)`):

```
int err = kill(pid, sig);
```

As with `sigaction` we need to request the appropriate POSIX standard before our headers.

With all this done, are you able to still kill your 'evil' program? Using which signal?