ENCE360 Lab 3: Signals and Sockets

Objectives

The overall goal of this lab is to introduce you to two key methods of inter-process communication: signals and sockets. Both are used constantly as you interact with the operating system.

The aims of this lab are for you to understand and be able to:

1. Send signals
2. Configure signal handlers for non-default behaviour
3. Set up a simple socket interface

Preparation

As well as this handout, `lab3Signals.zip` should contain the files `waiting.c`, `server.c`, and `client.c`.

Once you've completed the tasks below, make sure you also complete the lab quiz.

Program `waiting.c`

Signals are the base level of inter-process communication provided by the operating system. They occur asynchronously (with no specified timing or sequencing). A list of all the signals you can send can be found with the shell command `man 7 signal`. These are defined in the header file `signal.h`.

Compile and run `waiting.c`, and observe the output generated.

- Can you follow the order of execution? Draw a timeline to help!
- Is there any other order this program could execute in?

Now make a copy of `waiting.c` named `asyncWaiting.c`. Modify `asyncWaiting.c` so that instead of the parent immediately waiting for the child process after sending the `SIGQUIT`, the waiting is delayed until the SIGCHLD signal is caught by the parent process. This will involve:

- Defining a signal handler, e.g. `void waitChild(int sigNum)`, which will call `wait()` when the parent receives the `SIGCHLD` signal.
- Having the parent registers this signal handler using the system call `signal()` before the child is created.

Observe and understand the differences in the output produced from `waiting.c` and `asyncWaiting.c`.

Program `sigaction.c`

If you type `man 2 signal` into a terminal you will see a warning to **Avoid the use** of the `signal()` function, and to use `sigaction()` instead. `sigaction()`

is more complex, but is a good introduction to struct-based configuration that's common when dealing with the operating system. It's also the only function of the two guaranteed to work, so we'll implement something with it.

Have a look at the man page with `man sigaction`, particularly the start of the "Description" (up to and including the table), and the example at the bottom (some computers have their example in a different page, like `man 2 mprotect`). Note that we first initialise a configuration `struct`, then pass it along with the function pointer to `sigaction()`.

Now, make a copy of `waiting.c` called `sigaction.c`. Modify it so it uses `sigaction()` instead of `signal()`.

Configuration via `struct` like this is very common in low-level programming, and it's best to get used to it. Also note the practice of passing `NULL` when we don't care about an argument; this is an advantage of using pointers.

Programs `server.c` and `client.c`

Sockets are an OS-supplied inter-process communication tool. They are similar to pipes from last lab, and are interacted with like files once opened. Sockets are designed to be connected to by multiple clients, and so are used much more than named pipes. While sockets can be used over the internet, we're just talking about local OS sockets.

Notably, the windowing system for Linux, X11, is actually a local server, and all programs wanting to display a window connect to it as clients. This approach works well for central OS services that many processes will want access to.

First, compile both `server.c` and `client.c`. Run each in a different terminal window, starting the server first. You should see two-way communication once the second process begins.

Now have a look through the code and see if you can follow the flow. In particular, see where the file descriptors are created in each program:

- There is only one in `client.c`, `sockfd`, which we `read()`/`write()` to.
- There are two in `server.c`. The first, `sock`, is what we use to `accept()` incoming connections. Doing so creates a second, `sockfd`, which we then `read()`/`write()` to.

Once you understand this, try modifying the server to `fork()` a child process each time a new connection is accepted. The end goal is to be able to have multiple clients connected to the server at once. This is a tricky one; feel free to ask for help!

Note: depending on how you `fork()` your children and exit your processes, you may need to delete the created file `mySocket` between runs.