Signals

- Technique to send notifications between processes

- Equivalent of waving a flag of a certain color to someone across a street. You can recognize what signal it is, but no message is attached to it.

- In order to send a signal you can do so from a terminal with the kill command

i.e. kill -9 PID will send signal 9 (SIGKILL) to the process PID

(The book has a page with all the signals that exist and default behavior)

- In C you can send a signal with

```
int kill(pid_t pid, int sig);
```

For example, `kill(pid, SIGINT)` would be equivalent to pid's process receiving a Control-C.

- You can send a signal to yourself using

int raise(int sig);

For example `raise(SIGSEGV)` will make your current process turn into a Segmentation Fault. Note that you might not be accessing a bad memory address, but sending the signal will have the same result.

```
#include <signal.h>

int main()
{
   raise(SIGSEGV);
   return 1;
}
```

Is the fastest way to get a segmentation fault in a program, and all without messing up with memory! This will still create a .core file.

- Many signals are sent automatically (like SIGSEGV above), others can be send through a keyboard (like SIGINT).

- Some signals can be caught and handled any way the user wants. Others cannot be caught, so that there is always some way to terminate a process.

- To handle a signal use the method:

sighandler_t signal(int signum, sighandler_t handler)

sighandler_t is a *function pointer* for a function taking an int (the signal received), and returning void.

When the signal is received, signal will suspend the current execution and go to the signal handler).

For example:

```
#include <signal.h>

void hello(int sig)
{
     my_str("You pressed control-C, good
for you!\n");
}

int main()
{
    signal(SIGINT, hello);
    while (1);
    return 1;
}
```

This would be a horrible program that goes on an infinite loop that cannot be exited with control-C. Each control-C causes a SIGINT which will print our little message and go back where execution was.

(you can still kill it by looking for it's process id and using kill -9 pid since SIGKILL cannot be caught)

- if you know that you are expecting to receive a signal, you can pause your process with the method

```
int pause();
```

- This will hold your process until a signal (any signal) is received. Now we could change our previous bad program into something slightly better:

```
int main()
{
   signal(SIGINT, hello);
   pause();
   return 1;
}
```

This way the program will wait for a signal, if SIGINT is the signal received, we will get the message and end the program. Any other signal will still un-pause your process, but the handler will not be called.

- Since pause suspends the process until a signal is received, you could end up with a process sitting there too long. You can prevent that with alarm:

```
unsigned int alarm(unsigned int seconds)
```

When called, alarm will wait for the amount of seconds specified. After that time has passed a `raise(SIGALRM)` will automatically happen.

To prevent an alarm from "ringing" if you already received a different signal, use alarm(0) which will shut off the current alarm.

Binary operations basics

- You can write hexadecimal numbers as 0x80 (for example)

0x80 would be the binary 1000 0000

If you use that number as a mask with the binary AND (&) with your characters for example, you will get the leftmost bit in your character.

```
if (character & 0x80)
   // you had a 1
else
  // you had a 0
```

How about moving bits around? The easiest way is to shift your character. For example:

character = character << 1;

That shifts the entire byte by one bit. Bits “fall” off the end and are truncated.

To “apply” a bit you can OR a byte with a bitmask. For example, to take a byte, shift it left by 1, and set the LSB to 1: `c = (c << 1) | 1;`