Week 09 2

CSC209 Fall 2023

Dr. Demetres (dee) Kostas

November 21, 2023

Announcments

- A3 is over
  - unless you've got an ext.
- A4 released tomorrow
- Still processing some remarks
  - thanks for waiting

Signals

- Inter-process communication
  - as well as
    - process termination/control
    - notification from other
      - system components
      - e.g. CPU, timers

Two sides

- sending
  - using `kill()`
    - or CLI `kill [options] PID`

- receiving
  - using signal handlers
  - `sigaction` API

We started an example

- where we wrote a program
  - with a handler
  - and we tried to affect it
    - using signals
      - from the command line

Let's come back to it

`signals.pdf`

How would you implement this?

- another one of my
  - open questions
- follow up:
  - from what you've seen
  - how do *you think*
    - it was implemented?

Consider

- the number of standard signals
  - is 31
    - I said this is partially
      - because we don't use 0
- what does the number 32
  - bring to mind?

bit manipulation

- C's fundamental hallmark
  - is a strong HW understanding
- of course
  - individual bits can be manipulated
    - in some fashion
    - with any programming language
- but it is more rare
  - to naturally modify bits
  - for any and all data types

What is bit manipulation?

- consider an `int`
  - assume `sizeof(int) = 4`
    - 4 *bytes*
    - 8 bits in a byte
      - 32 bits!

11111111111111111111111111111111

Let's simplify to 4 bits

- Consider using 4 bits
  - to represent 4 *signals*
    - pending signals
- if bit number N is 1
  - then the signal with value N
    - is pending!
- what would this mean 0100?
  - (0'th on the right)

Review `AND` and `OR`

- these are operations
  - `&` for `AND`
  - `|` for `OR`
- What is 1010 `AND` 1000
  - 1000

bit versus logical

- since, in general
  - we also use `0` for `false`
    - and anything else as `true`
- there are separate **logical**
  - operations for these
  - as in, how to say
    - `true AND false is false`
    - without specifying bits

So consider

- `&` for bit-wise `AND`
- `&&` for *logical* `AND`
- suppose we had

```
int x = 4;
int y = 7;

int logical = x && y; // this is 1!
int bit = x & y; // this is 4!!
```

Consider signals again

- this time
  - imagining possible signals
    - as a *set* of 32 bits
  - They are present in the set
    - if the corresponding bit
      - is 1...

bitmasks

- ok, let's consider four signals
  - from right-to-left
  - if signal 0 and 2
    - were pending
      - then I could represent them
      - using bits
      - `char sig_pending = 0b0101;`

bitmasks

- what if I wanted
- to *ignore* signal 0?
- I could create **a mask**
  - `char mask = 0b1110;`
    - the type just needs to be
      - big *enough*
    - have enough bits
- What happens if I use
  - bit-wise `AND`?

masking bits

- `sig_pending & mask`
  - means, no matter what
    - value `sig_pending` has
      - for the 0 signal
    - it will be ignored
      - since it is `AND` with 0

How about setting bits?

- we can use bit-wise `OR`
  - using the single `|`
- with the bit we want to set on
- e.g. to set signal 0

```
int pending_signals = 0x0; // none pending
int mask = 0xFFFFFFFF; // no 0 bits, no masking

int new_signal = 0b0001;
pending_signals = pending_signals | new_signal;
```

(added after lecture)

- we avoid doing this manually
- using macros to empty and set
  - elements of normal sets
  - and elements of masks

Let's examine `greeting.c` solution