CS 521: Systems Programming

Maps & Concurrency

Lecture 18

Maps

- Associative arrays, maps, dictionaries, hashmaps, etc. can be one of the most useful data structures!
  - With enough time and patience, all problems can be solved with a Map 😉
- You might be surprised to learn that POSIX C actually has a built-in hashtable
  - `hcreate()`, `hsearch()`, etc.
  - But we didn't learn about it because it's… not great for most use cases

Go Maps

On the other hand, Go maps are great!

Let's create one:

```
myMap := make(map[string]int)
/*                    |     |
                      |     \--> value's type
                      |
                      \--> key's type
*/
```

And put something in it: `myMap["test"] = 42`

Pre-Populating a Map

We can create a map and add entries to it at the same time:

```
myMap := map[string]int{
    "thing1":          1,
    "thing2":          2,
    "thing3":          45,
    "thing4":          99,
    "something else": 10000,
}
```

Don't forget the comma on the last line! (`,`)

Adding to a Map

```
ages := make(map[string]int)
ages["matthew"] = 45
ages["alice"] = 22
ages["joe"] = 99

...

ages["joe"] = 95 // The entry for 'joe' is updated w/ new value
```

The various print functions can handle maps automatically:

```
fmt.Println("Here's everyone's ages:", ages)
```

Deleting from a Map

The built-in `delete` function removes items from the map. `len` reports it size, same as arrays or slices:

```
ages["matthew"] = 99
fmt.Printf(">>> %d\n", len(ages))
>>> 3

delete(ages, "matthew")

fmt.Printf(">>> %d\n", len(ages))
>>> 2
```

Checking for Items

We can use a 2nd optional return value when retrieving from a map to determine whether the element is present or not:

```
lookup := "bill"
_, present := ages[lookup]
if present {
    fmt.Println("We have " + lookup + " !")
} else {
    fmt.Println("There is no " + lookup + " here :-(")
}
```

Default Values

Let's try accessing an item that doesn't exist in the map:

```
fmt.Println(ages["bobby"])
```

What happens? An error? Runtime exception? Panic?

```
0
```

The default value for the datatype (`int` in this case) is returned.

What Can be a Key?

- We can use anything that's comparable as a key. This includes:
  - boolean, numeric, string, pointer, and structs or arrays that contain only those types
  - With structs, all the members are used to evaluate equality
- We cannot use slices, maps, and functions as keys

Next Up: Concurrency

Options for Multithreading

- We've can run a goroutine with the `go` keyword
  - goroutine: lightweight thread
    - managed by the go runtime, not the OS
- You might wonder: *can we create a full-fledged OS thread? Sort of like pthreads?*
  - The answer: **no**.
    - One exception: dealing with C threads. But that's a special case
- With go, goroutines are your **go**-to way to deal with concurrency

Synchronization in Go

- Okay, what about synchronization?
- Say we have two goroutines and they both access a global variable… is that safe?
  - **no!**
  - We still need to protect the global, just like we would in C or Java
    - Additionally, many built-in data structures (like maps) are not thread-safe
      - Check the documentation (as usual 😄)
- Next question: do we have mutexes, then?

Mutexes

- The `sync` package provides mutexes and condition variables
  - As well as `RWMutex`, a synchronized `Map`, and others
- **HOWEVER**, using them is generally *not* recommended
  - They are intended for use by lower-level libraries where more control or performance is needed
- Instead, go has a different mechanism for dealing with concurrency: **channels**
  - Based on a paradigm called CSP (communicating sequential processes)

Communicating Sequential Processes

- CSP is designed to simplify concurrent programming
- Instead of coordinating access to shared memory, each **process** communicates via **message passing**
- The processes execute sequentially and communicate by sending messages to the other processes
  - Sharing by sending/receiving data rather than coordinating shared access
  - Once something has been shared, the receiving end holds a **copy**
    - But could send back a modified copy

CSP vs. Synchronization

- Performance with CSP is generally not *quite* as good as using traditional synchronization
  - Some (usually small) overhead from passing the messages and making copies
- However, the upside is sequential processes are **much** easier to reason about and program
- Go does this with **channels**
  - goroutines can send/receive data over the channel
  - synchronization can be achieved by *blocking* to wait for a message

Creating a Channel

```
var channel = make(chan datatype)
```

So, we could do:

```
var channel = make(chan string)
```

Or even (assuming `Message` is a struct):

```
var channel = make(chan Message)
```

Sending and Receiving Data

```
/* Let's send a message through the channel: */
channel <-"Hello"

/* Receive it somewhere else (probably in another goroutine): */
msg := <-channel
```

Note: the “arrow” you’re seeing here is a single operator.

You can’t “point” it in another direction, e.g., `->`

Rethinking Producer-Consumer

- Let's say we want to re-implement the producer-consumer paradigm in go
- Have the producer make the work:

```
channel <-Work{workStart, workEnd}
```

- Then the consumer will grab the job and start working on it: `work := <-channel`
- (we'll assume `channel` is a global variable)
- Done!
  - kinda.

Issues

- How do we shut the threads down?
  - `close(channel)`
  - `work, more := <-channel`
    - `more` will be `false` when the channel has been closed
- What if we want the producer to be able to make more than one piece of `Work` at a time?
  - Create a buffered channel:
  - `var channel = make(chan Work, 100)`

Dealing with Deadlock

- Let's say all the goroutines in your program are blocked (probably waiting on a channel for data)
- Uh oh, deadlock, right? I guess the the program will hang…
- …nope! The go scheduler can detect when all goroutines (including the main goroutine) are blocked and will terminate the program
  - ```
    fatal error: all goroutines are asleep - deadlock!
    ```
    ```

What About Race Conditions?

- If two goroutines try to modify a shared variable, we have a race condition
  - Not something we can easily detect with the scheduler…
- But amazingly, go has a built-in data race detector
  - `go run -race something.go`

Dealing with Multiple Channels

- As you can imagine, more complicated programs may use multiple channels
- However, reading from a channel is a blocking call… What happens if we want to grab data from *any* channel that's ready?
- We can fall back on an old trusty Unix friend that's found its way into the go runtime: `select`

Select

In the following example, the ordering of the prints depends on which channel is ready first:

```
select {
case msg1 := <-channel1:
    fmt.Println(msg1)
case msg2 := <-channel2:
    fmt.Println(msg2)
}
```

Often used inside an infinite loop (say, in a consumer) to grab data from any available channel.