Concurrency in Go

nigeltao@golang.org

Concurrency… in Go

- Concurrency is about dealing with lots of things at once.
  - The opposite of sequential, or dealing with one thing at a time.
  - A web server deals with concurrent incoming and outgoing work whose lifetimes overlap.
- The Go programming language was open sourced in 2009.
  - In the Algol family (C, C++, Python, Java, JavaScript, C#). Example code:
    - ```
      func printTheSquare(x int) {
        logger.Println(x, "squared is", x*x)
      }
      ```
  - Emphasis on software engineering in practice.
  - One practical concern is concurrency, and this talk is about that aspect of Go.
      ```

Concurrency

- Concurrency is the *composition* of independently executing things.
  - Concurrency is about *dealing* with lots of things at once.
  - Think of a web server handling overlapping requests.
- Parallelism is the simultaneous *execution* of (possibly related) things.
  - Parallelism is about *doing* lots of things at once.
  - Think of vector math or GPUs, doing the same operation on multiple elements.
- Related, but separate ideas.
  - https://blog.golang.org/concurrency-is-not-parallelism
  - This talk is about concurrency.
- For example, Unix pipes connect processes.
  - find /usr/local/go/src/image | grep _test.go$ | xargs wc -l
  - On a single-core CPU, this will be concurrent without being parallel.

Concurrency

- Two popular approaches to concurrency.
  - Equally expressive; duals of one another.
  - Dan Kegel, "The C10k problem", 1999, surveys handling 10,000 network connections.
- Threads are procedure oriented.
  - Synchronous.
  - 'Obvious' extension of sequential, imperative programming: multiple flows of control.
  - Think of Unix processes... that share memory.
- Events are message oriented.
  - Asynchronous.
  - Think of typical GUIs (Graphical User Interfaces).

Threads

- Threads are like sub-processes that share memory (the address space).
  - Context switches are cheaper than between processes.
  - Can share more state, e.g. caches, intermediate work.
- One problem: race conditions on that memory.
  - "`numRequests++`" is not atomic.
  - One solution: mutexes enforce mutual exclusion.
  - Another solution: only share immutable things.
- Another problem: fault isolation.
  - The Google Chrome web browser deliberately uses multiple (sandboxed) processes.
  - Not covered in this talk.
- Larger design question: put concurrency in the language or the libraries?
  - Not covered in this talk: functional languages, immutability.

Mutexes

- Protects shared, mutable state.
  - A necessary evil. (Or is it?)
- One problem: holding the lock for too short a time.
  - TOCTTOU (Time Of Check To Time Of Use).
  - Or, more commonly, simply forgetting to take the lock when needed.
- Another problem: holding the lock for too long a time.
  - ```
    mutex.Lock()
    x++
    logger.Println("x is", x)    // I/O can take arbitrarily long.
    mutex.Unlock()
    ```
- More problems: deadlock (e.g. dining philosophers), livelock, starvation.
  - How coarse or fine should the locks be?
    ```

Events

- Instead of actively doing this and that, loop:
    - Select what you're interested in, and then
    - React to things that happen.
- Commonly seen in GUI programming and in JavaScript.
    - Mouse, keyboard, paint events are all interleaved in the one event loop.
    - In web browser JavaScript, XMLHttpRequest events are also interleaved.
    - For web servers (e.g. Node.JS), incoming requests and back-end responses are interleaved.
- Asynchronous APIs.
    - When reading from a file or socket, don't wait for the result, tell me later.
- Often only one OS-level thread.
    - No need to create, schedule or mutually exclude multiple threads.

Events

- Events have their own problems.
    - Callbacks, callbacks everywhere.
    - Context lost as a single conceptual operation broken into multiple handlers:
        - I drag the mouse from A to B, but "A's location" isn't part of:
            - B's mouse event,
            - a local variable in the mouse event handler or
            - the call stack.
        - Somewhat easier with closures (e.g. JavaScript), but not ideal (e.g. nesting).
- Asynchronous APIs are viral.
    - http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

Go

- Concurrency in Go is built on three things:
  - Goroutines, a concurrent computation model,
  - Channels, a concurrent communication model,
  - Select, a concurrent control structure.
- These build on old concepts that were half-forgotten.
  - Tony Hoare, "Communicating Sequential Processes" (CSP), 1978.
- Threads or Events? Both!
  - The Go programmer uses 'threads', which is simpler conceptually.
  - The Go runtime is implemented as 'events', which is more efficient.
- No silver bullet. You can still have race conditions and deadlocks in Go.
  - But the programming model is higher level and it's easier to avoid bugs a priori.
  - Primary benefit is clarity, not efficiency per se, but clarity can lead to efficiency.

Goroutines

- Unit of execution: PC (program counter) and the stack.
  - Many goroutines are multiplexed onto few OS-level threads.
  - Cheaper than threads to create and switch between.
    - Goroutine stacks start small (2 kilobytes) and grow on demand. On Linux, by default, threads take 2 megabytes up front.
    - Scheduling doesn't require going through the kernel and back.
  - Language makes it easy and idiomatic to create new goroutines.
  - Feasible for a web server process to have a million goroutines.
- This requires compiler and runtime support.
  - We already require a runtime for garbage collection.
  - Other, leaner languages choose no mandatory runtime. That's OK too.
    - Programming language design is about trade-offs. There's more than one 'right' answer.

Goroutines

- Do foo and bar sequentially (similar to ; in a Unix shell):
  - ```
    foo(42)
    bar()
    ```
- Do foo and bar concurrently (similar to & in a Unix shell), where foo runs in a new goroutine *while* bar runs in this one:
  - ```
    go foo(42)
    bar()
    ```
- The order matters. This does not do foo concurrently with bar, as bar runs in a new goroutine *after* foo finishes:
  - ```
    foo(42)
    go bar()
    ```
    ```

Goroutines

- I/O APIs are synchronous. The runtime makes it efficient.
    - `nBytesRead, err := conn.Read(buffer)`
- Synchronous is simpler.
    - Go decoders (e.g. decompressors, image codecs) can just Read and block.
    - C/C++ decoders have to spill their intermediate state (local variables) and resume later.
- In Go, the basic web server model is simple, concurrent and efficient.
    -
      ```
      for {
        conn, err := socket.Accept()
        if err != nil {
          return err
        }
        go serve(conn)
      }
      ```

Channels

- The "go" keyword starts a new, concurrent goroutine.
  - ```go
    func main() {
      go expensiveComputation(x, y, z)
      anotherExpensiveComputation(a, b, c)
    }
    ```
- This story is incomplete.
  - How do we know when the two computations are done?
  - What are their values?
    ```

Channels

- Goroutines communicate with other goroutines via channels:
  - ```go
    func computeAndSend(ch chan int, x, y, z int) {
      ch <- expensiveComputation(x, y, z)
    }

    func main() {
      ch := make(chan int)
      go computeAndSend(ch, x, y, z)
      v2 := anotherExpensiveComputation(a, b, c)
      v1 := <-ch
      fmt.Println(v1, v2)
    }
    ```
    ```

Channels

- Communication (the <- operator) is sharing and synchronization.
  - Sharing is often giving.
  - Receive blocks until there is a sender.
  - Send blocks until there is a receiver.
  - Multiple readers and multiple writers are OK.
- Channels can also be buffered, so that sends don't always block.
  - Not covered in this talk.

Channels

- Do not communicate by sharing memory; instead, share memory by communicating.
  - https://blog.golang.org/share-memory-by-communicating
- Threads and locks are concurrency primitives; CSP is a concurrency model:
  - Analogy: Edgar Dijkstra, "Go To Statement Considered Harmful", 1968.
  - goto is a control flow primitive; structured programming (if statements, for loops, function calls) is a control flow model.

Channels

- Let's distribute some work over a pool of workers.
- Compare *communicating by sharing memory*…
    - ```
      type Work struct {
        x, y, z  int
        assigned bool
      }

      type WorkSet struct {
        mu   sync.Mutex
        work []*Work
      }
      ```
      ```

Channels

- ...with *sharing memory by communicating*.
- Each worker receives and sends pieces of work:
  - ```
    type Work struct { x, y, z int }

    func worker(in, out chan *Work) {
      for {
        work := <-in
        work.z = work.x + work.y
        out <- work
      }
    }
    ```
    ```

Channels

- The manager connects the workers:
  -
```
func main() {
  in  := make(chan *Work)
  out := make(chan *Work)
  for i := 0; i < nWorkers; i++ {
    go worker(in, out)
  }
  go workProducer(in)
  workConsumer(out)
}
```

Select

- Select chooses 1 of N communications. It blocks until one can proceed:
  - ```
    for {
      select {
      case work := <-in:
        doSome(work)
        out <- work
      case <-done:
        return
      }
    }
    ```
- "Done" could be a timeout, an explicit cancellation, etc.
  - https://blog.golang.org/context
    ```

Select

- Let's make a "chat roulette" server.
  - https://blog.golang.org/two-recent-go-talks
- Recall the basic web server model:
  - ```
    for {
      conn, err := socket.Accept()
      if err != nil {
        return err
      }
      go serve(conn)
    }
    ```
    ```

Select

- Matching two connections can select on the same channel:
  - ```
    var matcher = make(chan net.Conn)

    func serve(conn net.Conn) {
      select {
      case matcher <- conn: // Sending myself means please match with me.
        // Nothing else to do. We're now handled by the other goroutine.
      case partnerConn := <-matcher: // Receiving means I met my partner.
        chat(partnerConn, conn)
      }
    }
    ```
- Demo!
    ```

Summary

- Threads+locks and events can work, but Go offers a different approach:
  - Goroutines, a concurrent computation model.
  - Channels, a concurrent communication model.
  - Select, a concurrent control structure.
- Do not communicate by sharing memory; instead, share memory by communicating.
- Not a law, just a different way to think about concurrency.
  - Decompose problems into small, simple, self-contained things, working in concert.
  - Concurrency patterns: http://talks.golang.org/2012/concurrency.slide
  - Don't overdo it. Sometimes all you need is a mutex (and Go provides them).
- http://golang.org/ has tutorials, downloads, and more!