Concurrency in Go

COS 418: Distributed Systems
Precept 1

Daniel Suo
• Concurrency

• Communicating sequential processes (CSP)

• Concurrency with shared memory

• Advanced: Goroutines vs. threads

• Advanced: CSP and shared memory?!
What is concurrency?
Concurrency

“Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once” – Rob Pike
Concurrency: a review

- Want to correctly and efficiently manage shared resources accessed from multiple, concurrent clients

- What OS constructs could we use to implement a webserver?

- What if the webserver services requests that write to a shared database?
Concurrent in Go

- Supports two styles (why?):
  - Communicating sequential processes (CSP) use communication as synchronization primitive
  - Shared memory multithreading uses locks (and their ilk)

- Reason about concurrency via partial ordering (happens-before order). See [https://golang.org/ref/mem](https://golang.org/ref/mem)

- Use concurrency correctly, but not responsible for the minutiae of Go implementations
CSP: goroutines

- For now, assume goroutines = threads

- The `main` function runs in *main routine*
  
  ```go
  f()
  go f()
  ```

- When `main` returns, all goroutines terminate
Example: clock.go
CSP: goroutines (example)

```go
func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err) // e.g., connection aborted
            continue
        }
        handleConn(conn) // handle one connection at a time
    }
}
```
CSP: goroutines (example)

```go
func handleConn(c net.Conn) {
    defer c.Close()
    for {
        _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
        if err != nil {
            return // e.g., client disconnected
        }
        time.Sleep(1 * time.Second)
    }
}
```
CSP: channels

- *channels* let one goroutine send values to another
  
  ```go
  ch := make(chan int) // unbuffered channel
  ch := make(chan int, 0) // unbuffered channel
  ch := make(chan int, 3) // buffered channel with capacity 3
  ```

- **send**: `ch <- x` // send value x to ch

- **receive**: `x = <-ch` // assign value from ch to x

- **close**: `close(ch)`
  - Additional receives get zero value
  - Additional sends panic
CSP: unbuffered channels

- The sending goroutine blocks until another goroutine receives
- A goroutine that attempts to receive will block until another goroutine sends
- Unbuffered channels ‘synchronize’ sending and receiving goroutines
Example: synchronize.go
Goroutines are not guaranteed to happen before any event the program

An aggressive compiler might remove!!
CSP: unbuffered channels (example)

```go
package main

var c = make(chan int)
var a string

func main() {
    go func() {
        a = "hello, world\n"
        c <- 0
    }()

    <-c
    print(a)
}
```
CSP: pipelines and unidirectional channels

- Pipelines let us chain together several channels without special syntax; just do it

- Unidirectional buffers specify buffers as just senders
  - **Receive-only** \( \text{ch} := \text{make}(\langle \text{chan} \Rightarrow \text{int} ) \)
  - **Send-only** \( \text{ch} := \text{make}(\text{chan}\langle \Rightarrow \text{int} ) \)
Example: pipeline.go
CSP: pipelines and unidirectional channels

```go
func main() {
    naturals := make(chan int)
    squares := make(chan int)

    go func() {
        for x := 0; ; x++ {
            naturals <- x
        }
    }()

    go func() {
        for {
            x := <-naturals
            squares <- x * x
        }
    }()

    for {
        fmt.Println(<-squares)
    }
}
```
CSP: pipelines and unidirectional channels

• What if we only want to send a finite set of numbers?

```go
func() {
    for {  
        x, ok := <- naturals  
        if !ok {
            break
        }
        squares <- x * x  
    }  
    close(squares)
}()
```
CSP: pipelines and unidirectional channels

- Go extends the range loop syntax for this common case

```go
func() {
    for x := range naturals {
        squares <- x * x
    }
    close(squares)
}()
```
CSP: buffered channels

- Unbuffered channel is a special case

- If there are items in the buffer, neither sender nor receiver are blocked

- If the buffer is empty, the receiver is blocked; if the buffer is full, the sender is blocked

- Choosing buffer size takes some forethought! You can deadlock or force processes in a pipeline to wait
What will this code do?

```go
func main() {
    ch := make(chan int)
    <-ch
}
```
Example: deadlock.go
CSP: select

- `select` allows multiplexing so we can receive from multiple channels without blocking

```plaintext
select {
  case <-ch1: // discard ch1 data
    // ...
  case x := <-ch2: // assign ch2 data
    // ...
  default:
    // ...
}
```
Example: countdown.go
Concurrency with shared memory

- Although we can do everything with CSP, sometimes less convenient than shared memory.

- Won’t spend much time because you should be familiar.
  - `sync.Mutex`: mutual exclusion with lock / unlock.
  - `sync.RWMutex`: multiple read, single write.
  - `sync.Once`: initialize variables once.
Advanced topics

• Race detector is part of Go runtime/toolchain
  – Looks for one goroutine accessing shared variable recently written by another goroutine without mutex

• Go under the hood
  – Greenthreads with growable stacks multiplexed on OS threads (scheduled by Go runtime)
  – Locks wrapped in a threadsafe queue

• When should you use different concurrency models? Can you combine?
“Don't be clever.”
- Rob Pike