COS 316
Precept: Concurrency
Part 2
Precept Objectives

- Review Go concurrency concepts (needed for “connection pool” assignment)
- Gain more practice with Go and concurrency concepts
  - RWMutex
  - Condition Variables:
    - `sync.L.Lock` and `sync.L.Unlock`
    - `sync.Cond` and `Signal, Wait, Broadcast`
- Understand the Dining Philosophers problem
Review Mutexes

- Consider the following example

https://play.golang.org/p/LAfTM5gO-Ej
RWMutex

- An **RWMutex** - a reader+writer mutual exclusion lock.
- For an addressable **RWMutex** value `mu (mu sync.RWMutex)`
  - data writers
    - acquire the write lock of `mu` through `mu.Lock()` method calls
    - release the write lock of `mu` through `mu.Unlock`
  - data readers
    - acquire the read lock of `mu` through `mu.RLock()` method calls
    - release the read lock of `mu` through `mu.RUnlock`
- Why do we want different types of locks for writing vs reading?
- Modify the example (from previous slide) to use RWMutex
Notifications

- Use `sync.Mutex` and `sync.RWMutex` values can also be used to make notifications
  - Note - not recommended - for illustrative purposes only!
- What gets printed first? Why?

- [https://play.golang.org/p/cw_os3bQfAG](https://play.golang.org/p/cw_os3bQfAG)

```go
cfunc main() {
    var mu sync.Mutex
    mu.Lock()
go func() {
        time.Sleep(time.Second)
        fmt.Println("COS")
        mu.Unlock()
    }()
    mu.Lock()
    fmt.Println("316")
}
```
Condition Variables - sync.Cond

- **sync.Cond** type - provides an efficient way to send notifications among goroutines

- sync.Cond value holds a **sync.Locker** field with name L - field value is of type *sync.Mutex* or *sync.RWMutex*

  - E.g.:
    - cond := sync.NewCond(&sync.Mutex{})
    - cond.L.Lock()
    - cond.L.Unlock()

- sync.Cond value holds a FIFO queue of waiting goroutines
Condition Variables - L.Lock(), L.Unlock(), Wait(), Broadcast(), Signal()

- cond := sync.NewCond(&sync.Mutex{})
- cond.L.Lock()
- cond.Wait()
- cond.Broadcast()
- cond.Signal()

- Call L.Lock() before Wait()
- Insert calling goroutine in queue and block (wait)
- Calls L.Unlock()

- Blocked routines go back to running state
- Invokes cond.L.Lock() (in the resumed cond.Wait() call) to try to acquire and hold the lock cond.L again
- cond.Wait() call exits after the cond.L.Lock() call returns
Condition Variables - Example

● Review the following example

● https://play.golang.org/p/m4kWYFYv5gD
Dining Philosophers

- Classic problem that illustrates issues related to synchronization
- Models concept of multiple processes competing for limited resources
- Formulated by E.W. Dijkstra
- Framework:
  - Five philosophers seated at a table
  - Infinite cycle of thinking and eating
  - Philosopher must pick up both forks in order to eat
  - Determine policy / algorithm so that each philosopher gets to eat and does not starve
Dining Philosophers Policy

- The philosophers require a shared policy that can be applied concurrently
- The philosophers are hungry! The policy should let everyone eat (eventually)
- The philosophers are utterly dedicated to the proposition of equality: the policy should be totally fair
- Discuss - what can go wrong?
type Philosopher struct {
    name  string // name of philosopher
    left  int    // fork number on the left
    right int    // fork number on the right
}
func (p *Philosopher) Dine(table []sync.Mutex) {
    for {
        p.Think()
        table[p.left].Lock()
        table[p.right].Lock()
        p.Eat()
        table[p.right].Unlock()
        table[p.left].Unlock()
    }
}

func main() {
    philosophers := make([]*Philosopher, len(philosophers))
    philosophers[0] = &Philosopher{
        name:  "Michelle",
        left:  0,
        right: 1,
    }
    philosophers[1] = &Philosopher{
        name:  "Bill",
        left:  1,
        right: 2,
    }
    philosophers[2] = &Philosopher{
        name:  "Sonia",
        left:  2,
        right: 3,
    }
    philosophers[3] = &Philosopher{
        name:  "Brooke",
        left:  3,
        right: 4,
    }
    philosophers[4] = &Philosopher{
        name:  "Eric",
        left:  4,
        right: 0,
    }
    table := make([]sync.Mutex, len(philosophers))
    for _, philosopher := range philosophers {
        go func(p *Philosopher) {
            p.Dine(table)
        }(philosopher)
    }
}
Solution 1 - Demonstration

● Run the program:
  ○ [https://play.golang.org/p/bV0JhIhN9It](https://play.golang.org/p/bV0JhIhN9It)

● Notes
  ○ Math.rand does not produce random numbers on the playground
  ○ Try running locally (copy and paste)
Solution to Problem

➢ Dijkstra
  ○ Number the resources (forks) from 0 to 4
  ○ Process (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork

➢ Are there any problems with this approach?
References

https://go101.org/article/concurrent-synchronization-more.html