Concurrency with shared memory

nprenet@bsu.edu

CS431, Spring 2022

Contents

1. Variable confinement

Variable confinement is one way to make a program concurrency-safe: ensure that only one routine owns the variable at any given time.

1.1 Channels (reminder)

Last week's lecture showed how we could use channels to make so that a single routine is responsible for making read and write operations on the variable:

```
var balances = make(chan int)
var deposits = make(chan int)

func Deposit(amount int) { deposits <- amount }

func Balance() int { return <-balances }

func teller() {
    var balance int
    for {
        select {
        case amount := <-deposits:
            balance += amount
        case balances <- balance:
        }
    }
```

```
}

func init() {
    go teller()
}
```

In this case, the `teller` function acts as a gateway to the variable. Client functions `Deposit` and `Balance` communicate with the routine through channels.

1.2 Serial confinement (pipeline)

If more than one routine need to access the variable during its lifetime, ensure that the ownership of the variable is *transferred* from one routine to another. In the pipeline below, all accesses to a given `cake` value are sequential:

```
type Cake struct { state string }

func baker(cooked chan<- *Cake) {
    for {
        cake := new(Cake)
        cake.state = "cooked"
        cooked <- cake // baker never touches this cake again
    }
}


func icer(iced chan<- *Cake, cooked <-chan *Cake) {
    for {
        cake := range cooked {
        cake.state = "iced"
        iced <- cake // icer never touches this cake again
    }
}
```

2. Mutual exclusion

The third way to avoid a data race is to allow many goroutines to access the variable, but only one at a time.

2.1 Binary semaphore (with buffered channel)

```
var (
    sema    = make(chan struct{}, 1)
    balance int
)

func Deposit(amount int) {
    sema <- struct{}{}
    balance += amount
    <-sema
}

func Balance() int {
    sema <- struct{}{}
    b := balance
    <-sema
```

```
    return b
}
```

2.2 The sync.Mutex type

Basic usage

```
import "sync"

var (
    mu      sync.Mutex
    balance int
)

func Deposit(amount int) {
    mu.Lock()
    defer mu.Unlock()
    balance += amount
}

func Balance() int {
    mu.Lock()
    defer mu.Unlock()
    return balance
}
```

Considerations:

- Same functionality as the binary semaphore, in a convenient type.
- Use `defer ...` for longer functions (added benefit: it executes *after* the return statement).

Example: making a data structure concurrency-safe

You can combine encapsulation and locks to create concurrency-safe types:

```
type Counter struct {
    mu sync.Mutex
    value int
}

func (c *Counter) Increment {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.value++
}
```

Locks are not re-entrant!

The function of a lock:

- Obvious: prevent other routines from accessing the variable
- Less obvious: guarantee that a shared variable is an consistent state when passed to another function that has acquired the lock.

The second condition is *not* compatible with the ability for a callee to acquire a mutex that has already been acquired by the caller (what we call a **re-entrant** lock).

```
var (
    mu      sync.Mutex
    balance int
)

func Deposit(amount int) {
    mu.Lock()
    defer mu.Unlock()
    deposit(amount)
}

func Balance() int {
    mu.Lock()
    defer mu.Unlock()
    return balance
}

func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    // Deposit(-amount)  // incorrect: the lock is not re-entrant
    deposit(-amount)
    if balance < 0 {
        deposit(amount)
        fmt.Println("Insufficient funds")
        return false // insufficient funds
    }
    return true
}

// This function requires that the lock be held
func deposit(amount int) {
    balance += amount
}
```

Considerations:

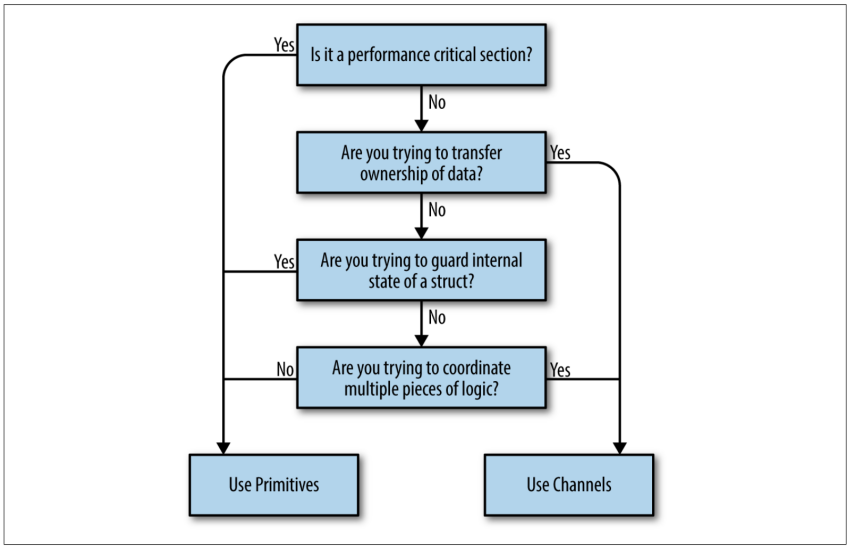
- The `Deposit()` function is meant to be called as a self-contained function: therefore, it must acquire and release the lock.
- The `deposit()` function is meant to be called (possibly twice) within the more complex `Withdraw()` routine, which already has the lock.

2.3 Also in the sync package: WaitGroup

```
var wg sync.WaitGroup

wg.Add(1)
go func() {
    defer wg.Done()
    fmt.Println("1st goroutine sleeping...")
    time.Sleep(1)
}()

wg.Add(1)
go func() {
```

```
    defer wg.Done()
    fmt.Println("2nd goroutine sleeping...")
    time.Sleep(2)
}()

wg.Wait()
fmt.Println("All goroutines complete").
```

Conclusion: locks or channels ?



Katherine Cox-Buday, *Concurrency in Go*, 2017, p. 35.