GO PO

15440 Fall 2021

Session 1: 7pm - 7:50pm
Session 2: 8pm - 8:50pm

Stolen from Fall'18 slides made by Samuel Kim and Chelsea Chen

Outline

- Overview
- Brief syntax overview
- Go Concurrency Examples
- P0
- Tips/Tricks
- Questions

Overview

- ➔ Purpose of this recitation:
  - ◆ Teach important GO concurrent models.
    - Will be crucial in p0 and p1.
    - Mutexes not allowed until p2.
  - ◆ Briefly go over p0.
  - ◆ Answer GO related questions.
- ➔ Strengthen basic knowledge of GO and its syntax.
  - ◆ Brief syntax overview.
  - ◆ **Please read and understand all of “A Tour of GO” before you start the project.**

How not to write Go

- Know another language,
- Think in terms of the other language,
- “Translate” statement by statement.

If you want to enjoy the power of Go...

...you need to embrace its idiosyncrasies

We kind of force you to do this anyway.

Brief Syntax Overview (1/5)

```
package main

import (
    "fmt"
    "math"
)

func test1(x, y int, s string) (int, string) {
    var a, b int = 1, 2
    var c float64
    c = math.Pi
    p := &x
    *p = 21

    return (a + b + int(c) + x + y), s

}

func main() {
    v1, _ := test1(-100000, 1, "p != np")
    fmt.Println(v1)

}
```

Always a part of a package

Library imports

Return types specified here

Implicit int pointer type

Type casting

Basics

Brief Syntax Overview (2/5)

```
func test2() {
    defer fmt.Println("this will print when the function exits")

    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }

    if newSum := sum * 100; newSum > 1000 {
        for sum < 1000 {
            sum += sum
        }
    } else {
        for {
        }
    }
}
```

One way to write a for loop

One-line declaration/conditional

Another way to write a for loop

Infinite for loop

Conditionals & For loops

Brief Syntax Overview (3/5)

```
type Vertex struct {
    X int
    Y int
}

func test3() {
    var a [2]string
    a[0] = "Hello"
    fmt.Println(a)

    v := Vertex{X: 5}
    fmt.Println(v)

    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("Morning")
    default:
        fmt.Println("Hi.")
    }
}
```

Definition of struct
(Note: methods/structs starting with capital letters are *public*)

Instantiation of array.

Instantiation of struct.
Unspecified struct fields will have default values.

Switch statements & Structs

Brief Syntax Overview (4/5)

```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func test4() {
    m := make(map[int]int)

    var b []int
    b = pow[:2]
    b[1] = -100

    for i, v := range pow {
        m[v] = i
    }
    fmt.Print(m)
}
```

Use the make built-in function to allocate dynamic data structures

Arrays/Slices and Maps

Brief Syntax Overview (5/5)

```
type Test5Interface interface {
    Something()
}
type RandomStruct struct {
    S string
}

func (t *RandomStruct) Something() {
    fmt.Println(t.S)
}
func main() {
    var i Test5Interface

    i = &RandomStruct{"Hello"}
    fmt.Printf("(%v, %T)\n", i, i)
    i.Something()
}
```

Interface just contains method signatures (method name, input/output types).

RandomStruct implicitly implements Test5Interface (by implementing all methods)

'i' is of type Test5Interface

Interfaces

Go Concurrency Example #1

- Problem: Consider the producer/consumer problem.
  - Producer generates data => puts into a buffer.
  - Consumer consumes data => removes from buffer.
  - Ensure producer won't add data into the buffer if buffer is full.
  - Ensure consumer won't remove data from an empty buffer.

Go Concurrency Example #1

- A Solution in C (docs.oracle)

```c
void producer(buffer_t *b, char item)
{
    pthread_mutex_lock(&b->mutex);

    while (b->occupied >= BSIZE)
        pthread_cond_wait(&b->less, &b->mutex);

    assert(b->occupied < BSIZE);

    b->buf[b->nextin++] = item;

    b->nextin %= BSIZE;
    b->occupied++;

    /* now: either b->occupied < BSIZE and b->nextin is the index
       of the next empty slot in the buffer, or
       b->occupied == BSIZE and b->nextin is the index of the
       next (occupied) slot that will be emptied by a consumer
       (such as b->nextin == b->nextout) */

    pthread_cond_signal(&b->more);

    pthread_mutex_unlock(&b->mutex);
}
```

```c
char consumer(buffer_t *b)
{
    char item;
    pthread_mutex_lock(&b->mutex);
    while(b->occupied <= 0)
        pthread_cond_wait(&b->more, &b->mutex);

    assert(b->occupied > 0);

    item = b->buf[b->nextout++];
    b->nextout %= BSIZE;
    b->occupied--;

    /* now: either b->occupied > 0 and b->nextout is the index
       of the next occupied slot in the buffer, or
       b->occupied == 0 and b->nextout is the index of the next
       (empty) slot that will be filled by a producer (such as
       b->nextout == b->nextin) */

    pthread_cond_signal(&b->less);
    pthread_mutex_unlock(&b->mutex);

    return(item);
}
```

Go Concurrency Example #1

```go
package main

var messageBuffer = make(chan int, 3)

func produce() {
    for i := 0; i < 1000; i++ {
        messageBuffer <- i
    }
}

func consume() {
    for {
        message := <-messageBuffer
        fmt.Println(message)
    }
}

func main() {
    go produce()
    go consume()
}
```

- A Solution in Go
  - A lot shorter and simpler!
  - It even achieves more than the C code by spawning its own “threads”
- Is there a problem with the code?

Go Concurrency Example #1

```go
package main

var finishedProducing = make(chan bool)
var finishedConsuming = make(chan bool)
var messageBuffer = make(chan int, 3)

func produce() {
    for i := 0; i < 1000; i++ {
        messageBuffer <- i
    }

    finishedProducing <- true
}

func consume() {
    for {
        select {
        case <-finishedProducing:

            finishedConsuming <- true
            return
        case message := <-messageBuffer:

            fmt.Println(message)
        }
    }
}
```

```go
func main() {
    go produce()
    go consume()
    <-finishedConsuming

    fmt.Print("ALL GO routines ended.")
}
```

- Proper termination
  - use channels to communicate between go channels that they are done

Go Concurrency Example #1

```go
package main

const maxBufSize = 3      // Comment
const numToProduce = 1000 // Comment

var finishedProducing = make(chan bool)        // Comment
var finishedConsuming = make(chan bool)        // Comment
var messageBuffer = make(chan int, maxBufSize) // Comment

// Comment
func produce() {
    for i := 0; i < numToProduce; i++ {
        messageBuffer <- i
    }

    finishedProducing <- true
}

// Comment
func consume() {
    for {
        select {
        case <-finishedProducing: // Comment
            finishedConsuming <- true
            return
        case message := <-messageBuffer: // Comment
            fmt.Println(message)
        }
    }
}
```

```go
func main() {
    go produce()
    go consume()
    <-finishedConsuming

    fmt.Print("ALL GO routines ended.")
}
```

- Proper style
  - ‘go fmt’ command is your friend
  - no magic numbers
  - comment constants, functions, cases
- Anyone see one last bug?

Go Concurrency Example #2

- Problem: Sharing a data structure across many threads

Go Concurrency Example #2

```go
type bankAccount struct {
    balance int
}

func newBankAccount() *bankAccount {
    return &bankAccount{
        balance: 1000,
    }
}

func (acc *bankAccount) withdraw(amount int) {
    acc.balance -= amount
}

func (acc *bankAccount) deposit(amount int) {
    acc.balance += amount
}

func (acc *bankAccount) checkBalance() int {
    return acc.balance
}
```

- Can you see the problem?

```go
func main() {
    tomAccount := newBankAccount()
    jerryAccount := newBankAccount()

    for i := 0; i < 1000; i++ {
        go tomAccount.withdraw(1) // Jerry takes Tom's money
        go jerryAccount.deposit(1)

        go jerryAccount.withdraw(1) // Tom takes Jerry's money
        go tomAccount.deposit(1)
    }

    time.Sleep(time.Second) // Let go-routines finish

    fmt.Printf("Tom's balance is: %d\n", tomAccount.checkBalance())
    fmt.Printf("Jerry's balance is: %d\n", jerryAccount.checkBalance())
}
```

Go Concurrency Example #2

```go
type bankAccount struct {
    balance        int
    depositChan    chan int
    withdrawChan   chan int
    balanceRequest chan bool
    balanceResult  chan int
}

func newBankAccount() *bankAccount {
    return &bankAccount{
        balance:        1000,
        depositChan:    make(chan int),
        withdrawChan:   make(chan int),
        balanceRequest: make(chan bool),
        balanceResult:  make(chan int),
    }
}

func (acc *bankAccount) withdraw(amount int) {
    acc.withdrawChan <- amount
}

func (acc *bankAccount) deposit(amount int) {
    acc.depositChan <- amount
}

func (acc *bankAccount) checkBalance() int {
    acc.balanceRequest <- true
    currBalance := <-acc.balanceResult
    return currBalance
}
```

```go
func (account *bankAccount) bankAccountRoutine() {
    for {
        select {
        case amount := <-account.depositChan:
            account.balance += amount
        case amount := <-account.withdrawChan:
            account.balance -= amount
        case <-account.balanceRequest:
            account.balanceResult <- account.balance
        }
    }
}

func main() {
    tomAccount := newBankAccount()
    jerryAccount := newBankAccount()

    go tomAccount.bankAccountRoutine()
    go jerryAccount.bankAccountRoutine()

    for i := 0; i < 1000; i++ {
        go tomAccount.withdraw(1) // Jerry takes Tom's money
        go jerryAccount.deposit(1)

        go jerryAccount.withdraw(1) // Tom takes Jerry's money
        go tomAccount.deposit(1)
    }

    time.Sleep(time.Second) // Let go-routines finish

    fmt.Printf("Tom's balance is: %d\n", tomAccount.checkBalance())
    fmt.Printf("Jerry's balance is: %d\n", jerryAccount.checkBalance())
}
```

- bankAccountRoutine manages data.
- Use channels to communicate requests/results.

P0: Introduction to Go Concurrency

- Released yesterday (September 7th, 2021)
- Due on **September 16th, 2021, 11:59 PM**
- **No grace days**, -10% penalty for each day late
- No submissions will be accepted after September 18th, 2021, 11:59 PM
- Please make sure you have checked out “A Tour of Go” https://tour.golang.org/welcome/1

P0 Part A: Concurrency

- A key value store.
  - Given an abstracted database, implement simple operations.
  - Multiple, concurrent clients.
  - Supports Put, Get, Delete, and Update.
- No mutexes (cannot use go's 'sync' package).
  - Also can't use channels as mutexes!
- You can only use 'bufio', 'io', 'bytes', 'fmt', 'net', and 'strconv'.
- All tests available to you.
  - This will not be true in future projects.

P0 Part B: Testing

- A Squarer: simple concurrent data structure.
- Well-defined API and **requirements**.
- We *provide* you with a working implementation.
- We have 2 *secret* buggy implementations.
- Can you write a (valid) test that catches the bug?
- Unlimited Gradescope submissions.
- Hint: read the code carefully + use your Go knowledge.

Good to know - GO

- `for-select` loop:
  - `select` will wait until a single case is ready.
- Know which calls are blocking.
  - unbuffered vs buffered channels.
  - TCP/UDP calls (e.g. Listen, Read, Write).
- Run '`go fmt`' before submitting to autolab!
- GoLand is an IDE by JetBrains (creators of IntelliJ).
  - Students can get for free: https://www.jetbrains.com/student/
- atom plugin go-plus is great.
- vsc has a Go extension.
- Make sure GOPATH/GOROOT is set properly.

Good to know - Miscellaneous

- Gradescope Issues.
  - No submission limits on P0, but there will be for future projects.
  - Many cores on autolab = more parallelism than your machine (run with -race).
  - Note many students will be submitting on deadline date.
  - Run code on AFS clusters before submitting.
- Installing GO.
  - Version 1.17!
  - https://golang.org/doc/install

Questions?