UCI Division of Continuing Education

Module 4: Synchronized Communication
Topic 1.1: Blocking on Channels

Iterating Through a Channel

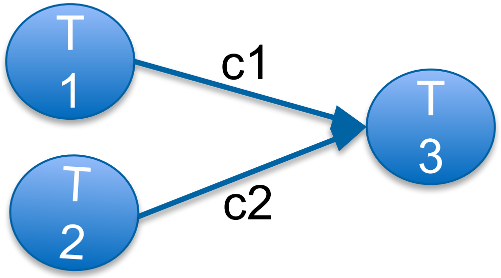
- Common to iteratively read from a channel

```
for i := range c {
    fmt.Println(i)
}
```

- Continues to read from channel c
- One iteration each time a new value is received
- i is assigned to the read value
- Iterates when sender calls `close(c)`

Receiving from Multiple Goroutines

- Multiple channels may be used to receive from multiple sources



- Data from both sources may be needed
- Read sequentially

```
a := <- c1
b := <- c2
fmt.Println(a*b)
```

Select Statement

- May have a choice of which data to use
  - i.e. First-come first-served
- Use the **select** statement to wait on the first data from a set of channels

```
select {
   case a = <- c1:
      fmt.Println(a)
   case b = <- c2:
      fnt.Println(b)
}
```

Module 4: Synchronized Communication
Topic 1.2: Select

Select Send or Receive

- May select either send or receive operations

```
select {
   case a = <- inchan:
      fmt.Println(“Received a”)
   case outchan <- b:
      fnt.Println(“sent b”)
}
```

Select with an Abort Channel

- May want to receive data until an **abort signal** is received
- Use select with a **separate abort channel**

```
for {
   select {
      case a <- c:
         fmt.Println(a)
      case <-abort:
         return
   }
}
```

Default Select

- May want a default operation to avoid blocking

```
select {
    case a = <- c1:
        fmt.Println(a)
    case b = <- c2:
        fmt.Println(b)
    default:
        fmt.Println(“nop”)

}
```

Module 4: Threads in Go
Topic 2.1: Mutual Exclusion

Goroutines Sharing Variables

- Sharing variables concurrently can cause problems
- Two goroutines writing to a shared variable can interfere with each other

Concurrency-Safe

- Function can be invoked concurrently without interfering with other goroutines

Variable Sharing Example

```
var i int = 0
var wg sync.WaitGroup
func inc() {
   i = i + 1
   wg.Done()}
func main() {
wg.Add(2)
   go inc()
   go inc()
   wg.Wait()
   fmt.Println(i)
}
```

- Two goroutine write to i
- i should equal 2

Possible Interleavings

- Seems like there is no problem

Task 1	Task 2	i
		0
i= i + 1		
		1
	i= i + 1	
		2

Task 1	Task 2	i
		0
	i= i + 1	
		1
i= i + 1		
		2

Granularity of Concurrency

- Concurrency is at the machine code level
- i = i + 1 might be three machine instructions

```
read i
increment
write i
```

- Interleaving machine instructions causes unexpected problems

Interleaving Machine Instructions

- Both tasks read 0 for i value

	Task 1	Task 2	i
			0
1:	**read i**		
2:		**read i**	
3:	inc		
4:	write i		
5:			1
6:		inc	
7:		write i	
8:			**1**

Module 4: Threads in Go
Topic 2.2: Mutex

Correct Sharing

- Don’t let 2 goroutines write to a shared variable at the same time!
- Need to restrict possible interleavings
- Access to shared variables cannot be interleaved

Mutual Exclusion

- Code segments in different goroutines which cannot execute concurrently
- Writing to shared variables should be mutually exclusive

Sync.Mutex

- A Mutex ensures mutual exclusion
- Uses a **binary semaphore**

- Flag up – shared variable is in use
- Flag down – shared variable is available

Module 4: Threads in Go
Topic 2.3: Mutex Methods

Sync.Mutex Methods

- `Lock()` method puts the flag up
  - Shared variable in use
- If lock is already taken by a goroutine, `Lock()` blocks until the flag is put down
- `Unlock()` method puts the flag down
  - Done using shared variable
- When `Unlock()` is called, a blocked `Lock()` can proceed

Using Sync.Mutex

- Increment operation is now mutually exclusive

```
var i int = 0
var mut sync.Mutex
func inc() {
   mut.Lock()
   i = i + 1
   mut.Unlock()
}
```

Module 4: Threads in Go
Topic 3.1: Once Synchronization

Synchronous Initialization

Initialization

  - must happen once
  - must happen before everything else

- How do you perform initialization with multiple goroutines?
- Could perform initialization before starting the goroutines

Sync.Once

- Has one method, **`once.Do(f)`**
- Function `f` is executed only one time
  - Even if it is called in multiple goroutines
- All calls to `once.Do()` block until the first returns
  - Ensures that initialization is executes first

Sync.Once Example

- Make two goroutines, initialization only once
- Each goroutine executes `dostuff()`

```
var wg sync.WaitGroup

func main() {
   wg.Add(2)
   go dostuff()
   go dostuff()
   wg.Wait()
}
```

Using Sync.Once

- **setup()** should execute only once
- “hello” should not print until **setup()** returns

```
var on sync.Once
func setup() {
   fmt.Println("Init")
}
func dostuff() {
   on.Do(setup)
   fmt.Println("hello")
   wg.Done()
}
```

Execution Result

```
Init
Hello
hello
```

Result of `setup()`

Result of one goroutine

Result of the other goroutine

- **`Init`** appears only once
- **`Init`** appears before **`hello`** is printed

Module 4: Threads in Go
Topic 3.2: Deadlock

Synchronization Dependencies

- Synchronization causes the execution of different goroutines to depend on each other

G1	G2
`ch <- 1`	`x := <- ch`
`mut.Unlock()`	`mut.Lock()`

- G2 cannot continue until G1 does something

Deadlock

- **Circular dependencies** cause all involved goroutines to block
  - G1 waits for G2
  - G2 waits for G1
- Can be caused by waiting on channels

Deadlock Example

```
func dostuff(c1 chan int,
                c2 chan int) {
   <- c1
   c2 <- 1
   wg.Done()
}
```

- Read from first channel
  - Wait for write onto first channel
- Write to second channel
  - Wait for read from second channel

Deadlock Example cont.

```
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    wg.Add(2)
    go dostuff(ch1, ch2)
    go dostuff(ch2, ch1)
    wg.Wait()
}
```

- `dostuff()` argument order is swapped
- Each goroutine blocked on channel read

Deadlock Detection

- Golang runtime automatically detects when all goroutines are deadlocked

```
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0x173e2c, 0x1042ff98)
…
```

- Cannot detect when a subset of goroutines are deadlocked

Module 4: Threads in Go
Topic 3.3: Dining Philosophers

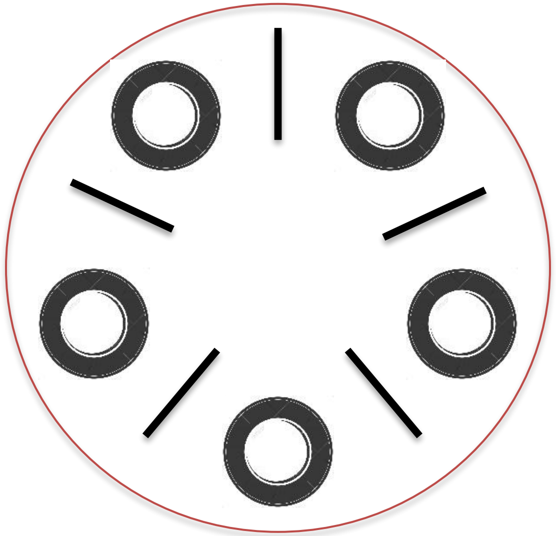
Dining Philosophers Problem

- Classic problem involving concurrency and synchronization

Problem

- 5 philosophers sitting at a round table
- 1 chopstick is placed between each adjacent pair
- Want to eat rice from their plate, but needs two chopsticks
- Only one philosopher can hold a chopstick at a time
- Not enough chopsticks for everyone to eat at once

Dining Philosopher Issues



- Each chopstick is a mutex
- Each philosopher is associated with a goroutine and two chopsticks

UCI Division of Continuing Education

Chopsticks and Philosophers

```
type ChopS struct{ sync.Mutex }

type Philo struct {
   leftCS, rightCS *ChopS
}
```

Philosopher Eat Method

```
func (p Philo) eat() {
   for {
      p.leftCS.Lock()
      p.rightCS.Lock()

      fmt.Println(“eating”)

      p.rightCS.Unlock()
      p.leftCS.Unlock()
   }
}
```

Initialization in Main

```
CSticks := make([]*ChopS, 5)
for i := 0; i < 5; i++ {
   CSticks[i] = new(ChopS)
}
philos := make([]*Philo, 5)
for i := 0; i < 5; i++ {
   philos[i] = &Philo{Csticks[i],
                   Csticks[(i+1)%5]}
}
```

- Initialize chopsticks and philosophers
- Notice `(i+1)%5`

Start the Dining in Main

```
for i := 0; i < 5; i++ {
   go philos[i].eat()
}
```

- Start each philosopher eating
- Would also need to Wait in the main

Deadlock Problem

- All philosophers might lock their left chopsticks concurrently
- All chopsticks would be locked
- Noone can lock their right chopsticks

```
p.leftCS.Lock()
p.rightCS.Lock()
fmt.Println(“eating”)
p.rightCS.Unlock()
p.leftCS.Unlock()
```

Deadlock Solution

- Each philosopher **picks up lowest numbered chopstick first**

```
philos[i] = &Philo{Csticks[i],
                 Csticks[(i+1)%5]}
```

- Philosopher 4 picks up chopstick 0 before chopstick 4
- Philosopher 4 blocks allowing philosopher 3 to eat
- No deadlock, but Philosopher 4 may starve