# Module 3: Threads in Go
# Topic 1.1: Goroutines

# Creating a Goroutine

- One goroutine is created automatically to execute the `main()`
- Other goroutines are created using the **go** keyword

```
a = 1
foo()
a = 2
```

- Main goroutine blocks on call to foo()

```
a = 1
go foo()
a = 2
```

- New goroutine created for foo()
- Main goroutine does not block

# Exiting a Goroutine

- A goroutine exits **when its code is complete**
- **When the main goroutine is complete**, all other goroutines exit
- A goroutine may not complete its execution because main completes early

# Module 3: Threads in Go
# Topic 1.2: Exiting Goroutines

# Early Exit

```
func main() {
   go fmt.Printf("New routine")
   fmt.Printf("Main routine")
}
```

- Only “**Main routine**” is printed
- Main finished before the new goroutine started

# Delayed Exit

```
func main() {
    go fmt.Printf("New routine")
    time.Sleep(100 * time.Millisecond)
    fmt.Printf("Main routine")
}
```

- Add a delay in the main routine to give the new routine a chance to complete
- "**New RoutineMain Routine**" is now printed

# Timing with Goroutines

- Adding a delay to wait for a goroutine is **bad**!
- Timing assumptions may be wrong
  - Assumption: delay of 100 ms will ensure that goroutine has time to execute
  - Maybe the OS schedules another thread
  - Maybe the Go Runtime schedules another goroutine
- Timing is nondeterministic
- Need formal **synchronization** constructs

# Module 3: Threads in Go
# Topic 2.1: Basic Synchronization

# Synchronization

- Using **global events** whose execution is viewed by all threads, simultaneously

| | |
|---|---|
| 1: `x = 1` | |
| | 1: `print x` |
| 2: `x = x + 1` | |

| | |
|---|---|
| 1: `x = 1` | |
| 2: `x = x + 1` | |
| | 1: `print x` |

- Want print to occur after update of x

# Synchronization Example

**Task 1**

```
x = 1
x = x + 1
GLOBAL EVENT
```

**Task 2**

```
if GLOBAL EVENT
    print x
```

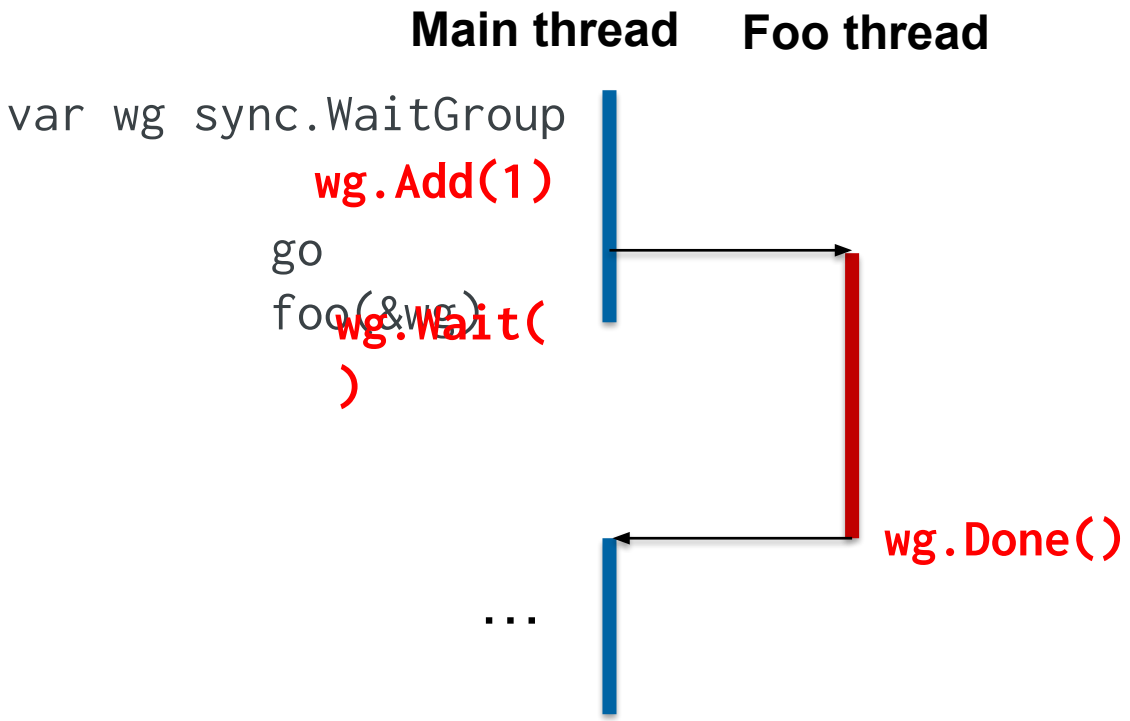
- **GLOBAL EVENT** is viewed by all tasks at the same time
- Print must occur after update of x
- Synchronization is used to restrict bad interleavings

# Module 3: Threads in Go
# Topic 2.2: Wait Groups

# Sync WaitGroup

- Sync package contains functions to synchronize between goroutines
- **sync.WaitGroup** forces a goroutine to wait for other goroutines
- Contains an internal counter
  - Increment counter for each goroutine to wait for
  - Decrement counter when each goroutine completes
  - Waiting goroutine cannot continue until counter is 0

# Using WaitGroup




- **`Add()`** increments the counter
- **`Done()`** decrements the counter
- **`Wait()`** blocks until counter == 0

# WaitGroup Example

```
func foo(wg *sync.WaitGroup) {
    fmt.Printf("New routine")
    wg.Done()
}
func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go foo(&wg)
    wg.Wait()
    fmt.Printf("Main routine")
}
```

# Module 3: Threads in Go
# Topic 3.1: Communication

# Goroutine Communication

- Goroutines usually work together to perform a bigger task
- Often need to send data to collaborate
- Example: Find the product of 4 integers
  - Make 2 goroutines, each multiplies a pair
  - Main goroutine multiplies the 2 results
- Need to send ints from main routine to the two sub-routines
- Need to send results from sub-routines back to main routine

# Channels

- Transfer data between goroutines
- Channels are typed
- Use `make()` to create a channel

```
c := make(chan int)
```

- Send and receive data using the `<-` operator
- Send data on a channel

```
c <- 3
```

- Receive data from a channel

```
x := <- c
```

# Channel Example

```
func prod(v1 int, v2 int, c chan int) {
   c <- v1 * v2}
func main() {
   c := make(chan int)
   go prod(1, 2, c)
   go prod(3, 4, c)
   a := <- c
   b := <- c
   fmt.Println(a*b)
}
```

# Module 3: Threads in Go
# Topic 3.2: Blocking on Channels

# Unbuffered Channel

- Unbuffered channels cannot hold data in transit
  - Default is unbuffered
- Sending blocks until data is received
- Receiving blocks until data is sent

**Task 1**

```
c <- 3
```

**Task 2**

One hour later …

```
x := <- c
```

# Blocking and Synchronization

- Channel communication is synchronous
- Blocking is the same as waiting for communication
- Receiving and ignoring the result is same as a Wait()

**Task 1**

```
c <- 3
```

**Task 2**

```
<- c
```

# Module 3: Threads in Go
# Topic 3.3: Buffered Channels

# Channel Capacity

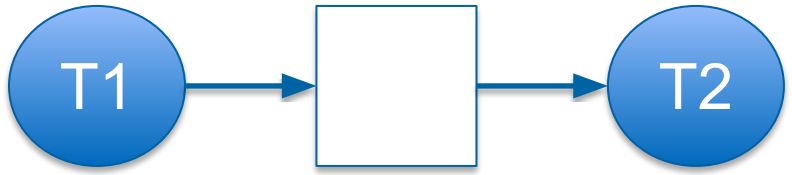
- Channels can contain a limited number of objects
  - Default size 0 (unbuffered)
- **Capacity** is the number of objects it can hold in transit
- Optional argument to `make()` defines channel capacity

```
c := make(chan int, 3)
```

- Sending only blocks if **buffer is full**
- Receiving only blocks if **buffer is empty**

# Channel Blocking, Receive

- Channel with capacity 1




```
c <- 3
```

```
a := <- c
b := <- c
```

- First receive blocks until send occurs
- Second receive blocks forever

# Channel Blocking, Send

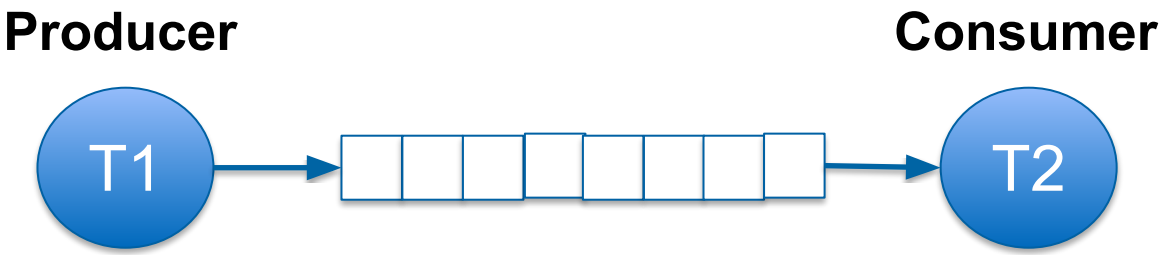

```
c <- 3
c <- 4
```

```
a := <- c
```

- Second send blocks until receive is done
- Receive can block until first send is done

# Use of Buffering

- Sender and receiver do not need to operate at exactly the same speed




- Speed mismatch is acceptable
- Average speeds must still match