Introduction to Programming in Go

11. Concurrency in Go

Module Topics

1. Concurrency in Go
2. Goroutines
3. Channels
4. Select Statement
5. Race Conditions

Concurrency in Go

Concurrency in Go

1. Concurrency is not parallelism.

2. Concurrency is independently functioning computations that structure a real world task.

3. Based on Hoare's CSP where concurrency is implemented through channels where computations share information and coordinate tasks.

4. Concurrency is a fundamental part of the design of Go.

Concurrency in Go

Concurrency is implemented with four Go mechanisms

1. Concurrent function execution implemented with goroutines.
2. Synchronization and communication between goroutines (channels).
3. Multi-way concurrent control (select statement).
4. Specific Go concurrency idioms.

Goroutines

Goroutines

1. A goroutine is an independently executing function.
2. Goroutines are multiplexed into threads but goroutines are more lightweight than a thread.
3. Function become goroutines by use of the go operator.
4. By default, the main() function is a goroutine.
5. Goroutines are very efficient resourcewise.

Normal Synchronous Call

```go
// Example 11-01 Normal function call
// We have to ctrl-C to end the infinite loop

import (
    "fmt"
     "math/rand"
     "time" )

func service(message string) {
    for i := 0; ; i++ { // Infinite loop
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
    service("Message:")
}
```

```
[Module11]$ go run ex11-01.go
Message: 0
Message: 1
Error: process crashed or was terminated
while running.
```

Made into Goroutine

```go
// Example 11-02 Goroutine function call
// main() ends before the loop executes
...

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                   time.Millisecond)
    }
}

func main() {
   go service("Message:")
   fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-02.go
Done!
```

Made into Goroutine

```go
// Example 11-03 Goroutine function call
// goroutine executes until main() exits

func service(message string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}

func main() {
   go service("Message:")
   time.Sleep(2 * time.Second)
   fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-03.go
Message: 0
Message: 1
Message: 2
Message: 3
Message: 4
Message: 5
Done!
```

Multiple Goroutines

```go
// Example 11-04 Goroutine function call
...
func service(message string) {
    for i := 0; ; i++ { // Infinite loop
         fmt.Println(message, i)
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                    time.Millisecond)
    }
}
func main() {
   go service("Alpha:")
   go service("Beta:")
   time.Sleep(2 * time.Second)
   fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-04.go
Alpha: 0
Beta: 0
Alpha: 1
Beta: 1
... snipped for space
Done!
```

Anonymous Goroutine

```go
// Example 11-05 Anonymous goroutine
...

func main() {
     go func() {
        for i := 0; ; i++ {
            fmt.Println("Anon:", i)
            time.Sleep(time.Duration(rand.Intn(1e3)) *
                                     time.Millisecond)
         }
     }()
   time.Sleep(2 * time.Second)
   fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-05.go
Anon: 0
Anon: 1
Anon: 2
Anon: 3
Anon: 4
Anon: 5
Done!
```

Channels

Channels

1. Channels are modeled after a socket or a pipe in Unix.

2. Channels are first class objects, like functions.

3. Channels are bidirectional and goroutines communicate by reading from and writing to channels.

4. Channels have types and are created with make(). In the code below "c" is a channel that passes ints.

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

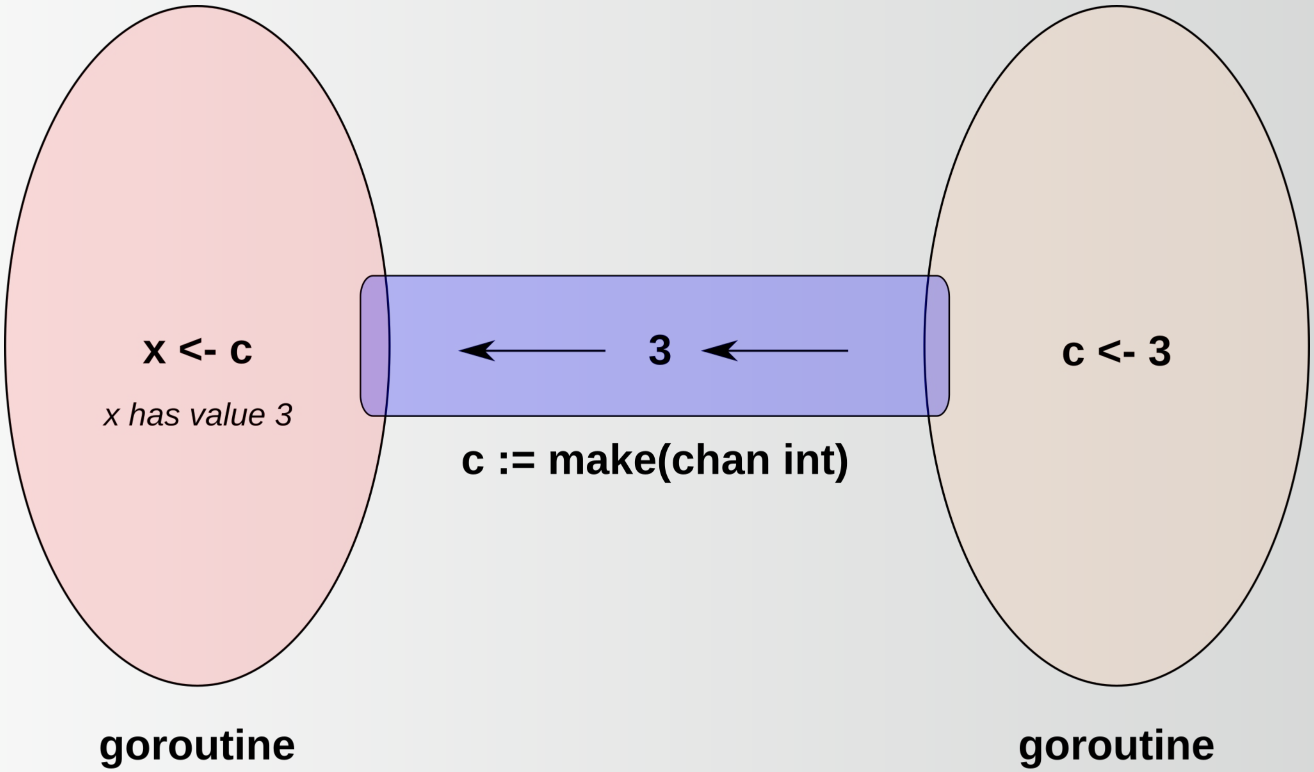
5. To write to the channel c we use the syntax

   ```
   c <-  2
   ```

   and to read from a channel c we use the syntax

   ```
   myvariable <- c
   ```

Communicating with Channels



Communicating with Channels

1. The value 3 is written onto the channel by the goroutine on the right.
2. The goroutine on the left reads it from the channel and assigns it to the variable x.
3. Channels will block on either side if the channel transmission cannot be completed.
4. The goroutine on the left blocks until there is something to read.
5. The goroutine on the right blocks until someone reads from the channel it just wrote to.
6. The channel "synchronizes" the activity of the two goroutines.

Basic Channel

```
// Example 11-06 Basic Channel
...
func service(message string, c chan string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
        c <- fmt.Sprintf("%s %d", message, i)
    }
}
func main() {
   c := make(chan string)
   go service("Message:", c)
   for i := 0; i < 5; i++ {
      fmt.Printf("main %d Got back: %q\n", i, <-c)
      time.Sleep(time.Second)
    }
   fmt.Println("Done")
}
```

Output for Example 11-06

```
[Module11]$ go run ex11-06.go
service:  0
service:  1
main 0 Got back: "Message:  0"
main 1 Got back: "Message:  1"
service:  2
main 2 Got back: "Message:  2"
service:  3
main 3 Got back: "Message:  3"
service:  4
main 4 Got back: "Message:  4"
service:  5
Done!
```

Deadlocks

1. Because channels block, we can deadlock.

2. Deadlocks usually occur when both sides are waiting for the other goroutine participating in a channel to do something.

3. Go detects deadlocks and issues a panic when one occurs.

Deadlocks

```go
// Example 11-07 Deadlocks
// Both goroutines are waiting to read

func service(message string, c chan string) {
      c <-"Hello"   //write
      fmt.Println(<-c) //read
    }
}

func main() {
   c := make(chan string)
   go service("Message:", c)
   c <-"Bye"   //write
   fmt.Println(<-c) //read
}
```

```
[Module11]$ go run ex11-07.go
fatal error: all goroutines are asleep - deadlock!
```

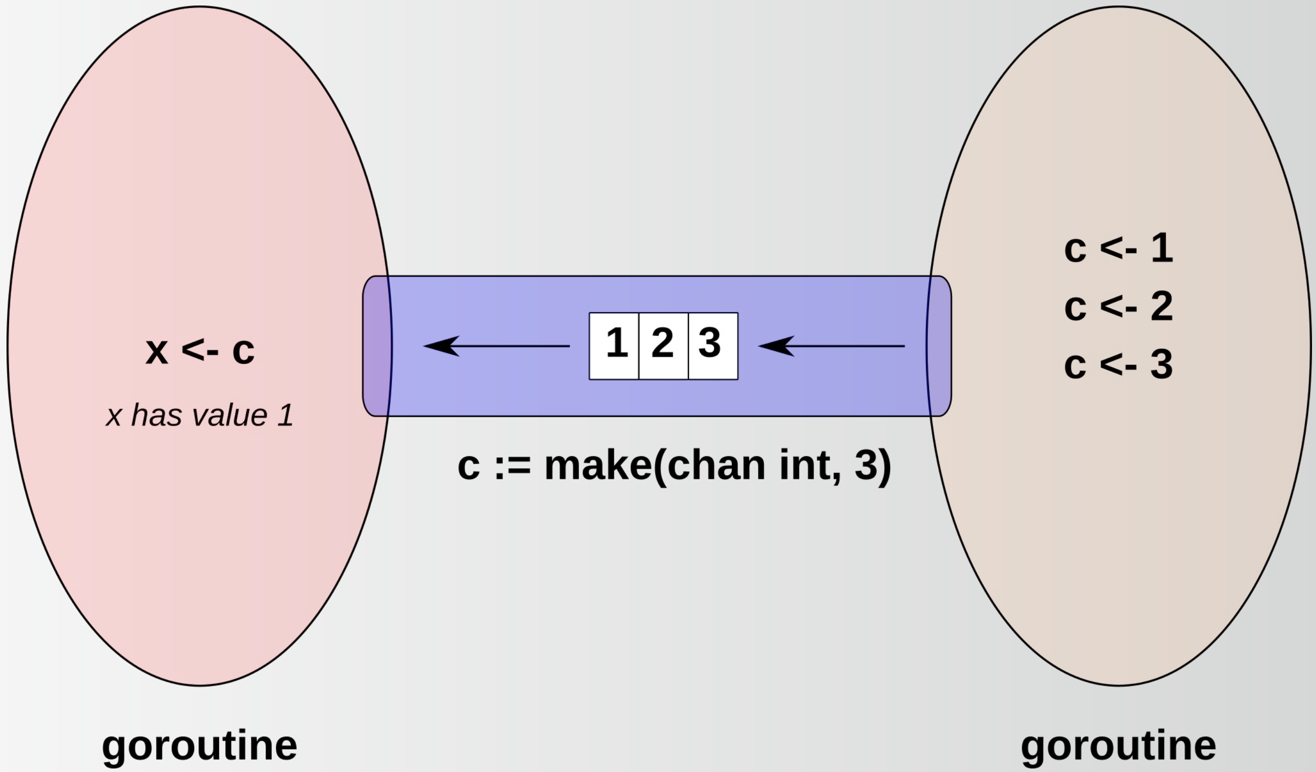
Buffered Channels

1. Buffers can be attached to channels.
2. Useful when there is a mismatch between the rate of reading and writing between the two goroutines.
3. Buffered channels are made with an initial capacity:

   ```
   c: = make(chan int, 5)
   ```

4. When the buffer is full, the channel blocks just like an unbuffered channel.
5. Buffers make channels less synchronized which can introduce difficulties when coordinating goroutines.
6. The capacity of the buffer is `cap(buffer)` and the number of items in the buffer is `len(buffer)`

Buffered Channels



Communicating with Channels

1. The values 1,2,3 are written onto the channel by the goroutine on the right..

2. The goroutine on the right blocks since the buffer is full.

3. The goroutine on the left reads 1 from the buffer.

4. The goroutine on the right can now write a new value.

Basic Buffered Channel

```go
// Example 11-08 Buffered Channels
...
func service(message string, c chan string) {
    for i := 0; ; i++ {
         fmt.Println(message,i,"buffered items ",len(c))
        c <- fmt.Sprintf("%s %d", message, i)
    }
}
func main() {
   c := make(chan string,3)
   go service("Message:", c)
   for i := 0; i < 5; i++ {
      fmt.Printf("main %d Got back: %q\n", i, <-c)
      time.Sleep(time.Second)
    }
   fmt.Println("Done")
}
```

Output for Example 11-08

```
[Module11]$ go run ex11-08.go
service:  0 buffered items  0
service:  1 buffered items  0
service:  2 buffered items  1
service:  3 buffered items  2
service:  4 buffered items  3
main 0 Got back: "Message:  0"
main 1 Got back: "Message:  1"
service:  5 buffered items  3
main 2 Got back: "Message:  2"
service:  6 buffered items  3
service:  7 buffered items  3
main 3 Got back: "Message:  3"
main 4 Got back: "Message:  4"
service:  8 buffered items  3
Done!
```

Unidirectional Channels

1. Channels are bidirectional because someone has to write to it and someone has to read from it.

2. When a channel is passed to a goroutine it may be specified as an input only or output only channel.

3. Specifying a direction only specifies how to use the channel inside that groutine - it is not property of the channel itself.

4. Trying to read on an output channel or write on an input channel is an error.

5. Direction is indicated like this

   `chan<-` (output)

   `<-chan` (input)

Unidirectional Channel

```
// Example 11-09 Unidirectional Channel
...
func service(message string, c chan<- string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
        c <- fmt.Sprintf("%s %d", message, i)
    }
}
func main() {
   c := make(chan string)
   go service("Message:", c)
   for i := 0; i < 5; i++ {
      fmt.Printf("main %d Got back: %q\n", i, <-c)
      time.Sleep(time.Second)
    }
   fmt.Println("Done")
}
```

Closing Channels

1. A close() operation on a channel is performed by a sender to indicate that no more data will be sent.

2. Trying to write to a closed channel causes a panic.

3. It is an error to try and close a read only channel.

Closing a Channel

```go
// Example 11-10 Closing a Channel
...
func service(message string, c chan<- string) {
    for i := 0; ; i++ {
         fmt.Println(message, i)
         c <- fmt.Sprintf("%s %d", message, i)
         close(c)  // will cause a panic next iteration
    }
}
func main() {
   c := make(chan string)
   go service("Message:", c)
   for i := 0; i < 5; i++ {
      fmt.Printf("main %d Got back: %q\n", i, <-c)
      time.Sleep(time.Second)
    }
   fmt.Println("Done")
}
```

```
[Module11]$ go run ex11-10.go
service:  0
service:  1
panic: send on closed channel
```

Closing Channels as a Signal

1. The range function will read on a channel until it is closed.
2. When the sender is finished sending data they close the channel.
3. Closing a channel causes the range function reading from it to terminate.
4. This ensures that the receiver is not blocked waiting for input that will never come.

Closing a Channel as a Signal

```go
// Example 11-11 Closing a Channel
...
func sender(output chan<- int) {
    for i := 0; i < 3; i++ {
         output <- i
    }
    close(output)
}
func receiver(input <-chan int) {
    for j := range input {
         fmt.Println("Received ", j)
    }
    fmt.Println("I'm done ")
}
func main() {
   c := make(chan int)
   go receiver(c)
   go sender(c)
   time.Sleep(time.Second)
  }
```

```
[Module11]$ go run ex11-11.go
Received  0
Received  1
Received  2
I'm done
```

The Blocking Problem

1. The previous example required a close() on the channel to prevent blocking.

2. However it may happen that the sender exits before the channel is closed, then the receiver blocks.

3. A Go idiom is to start a deferred goroutine that closes the channel whenever the sender exits.

4. We can check explicitly to see if a channel is closed by using the comma ok idiom.

Using comma ok

```
// Example 11-12 Go idioms
...
func sender(output chan<- int) {
     defer close(output)  // ensures no blocking
    for i := 0; i < 3; i++ {
         output <- i
    }
}
func receiver(input <-chan int) {
    for  {
         j, ok := <-input
         fmt.Println(ok)
         if !ok {
             break
         }
         fmt.Println("Received ", j)
    }
     fmt.Println("I'm done ")
}
func main() {} //as before
```

Output for Example 11-12

```
[Module11]$ go run ex11-12.go
true
Received  0
true
Received  1
true
Received  2
false
I'm done
```

The Select Statement

The Select Statement

1. The select statement is a channel polling mechanism that works like a switch statement.

2. Each case in the select statement is an input channel.

3. Each iteration, input is read from a channel that is not blocked.

4. If more than one channel is not blocked, then one of the unblocked channels is chosen at random.

5. If all the channels are blocked, then the default case is executed.

Select Statement

```
// Example 11-13 Generator goroutines
...
func sourceEven(outchan chan<- int) {
    for i := 0; ; i++ {
         output <- i * 2
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                                            time.Millisecond)
    }
}
func sourceOdd(outchan chan<- int) {
    for i := 0; ; i++ {
         output <- (i * 2) + 1
         time.Sleep(time.Duration(rand.Intn(1e3)) *
                                            time.Millisecond)
    }
}
```

Select Statement

```go
// Example 11-13 Multiplexer goroutine
...
func counter(ceven <-chan int, codd <-chan int) {
   for {
      select {
      case e := <-ceven:
         fmt.Println("Even:", e)
      case o := <-codd:
         fmt.Println("Odd:", o)
      default:
         fmt.Println("no one is ready")
         time.Sleep(100 * time.Millisecond)
      }
   }
}
```

Select Statement

```go
// Example 11-13 Main goroutine
...

func main() {
   codd := make(chan int)
   ceven := make(chan int)
   go sourceOdd(codd)
   go sourceEven(ceven)
   go counter(ceven, codd)
   time.Sleep(2 * time.Second)
}
```

```
[Module11]$ go run ex11-13.go
Even: 0
Odd: 1
no one is ready
Odd: 3
no one is ready
no one is ready
no one is ready
no one is ready
Even: 2
no one is ready
Odd: 5
Even: 4
no one is ready
Even: 6
no one is ready
no one is ready
no one is ready
Odd: 7
no one is ready
no one is ready
Even: 8
no one is ready
```

Race Conditions

Race Conditions

1. Channels are concurrency safe.
2. Often concurrency requires two goroutines to access the same variable or location in memory.
3. Race conditions mean interleaved reads and writes to the shared variable create some form of value corruption.
4. Race conditions usually only occur when the system is in a certain state, loading or stress.
5. Race conditions are very difficult to detect in testing.

Go Solution to Race Conditions

1. A Go idiom to prevent race conditions is to have the shared variable only accessible by one goroutine.

2. Requests to read and write are send to that goroutine over channels.

3. This generally will prevent the sort of environment necessary for a race condition.

4. The net effect is to "wrap" the resoutce in a goroutine and allow access only via channels which are concurrency safe.

Go Locking

1. If the Go Idiom is not feasible, then the resource can be locked with a mutex.

2. Each go routine that needs access locks the variable, accesses it then unlocks it.

3. This is a more traditional approach in other programming languages.

Lab 11: Concurrency