Synchronisation of Goroutines by means of Channels

1. Go Memory Model
2. Synchronisation on:
    2.1. Buffered Channels
    2.2. Unbuffered Channels
    2.3. Channel close()
3. Example: http://play.golang.org/p/UeHAShi_dS

The Go memory model for channel synchronisations is seen as a very subtle subject and it has generated in the past a number of discussions in the go-nuts news group. This document aims at clarifying these subtleties by proposing a visualization method that represents the "happens-before" relationships described in the Go memory model.

1. Go Memory Model

Modern hardware and compilers implement a number of optimisation techniques that often break sequential expectations. The Go memory model reports several examples and here, we quote two: Case 1 to illustrate the possibility of operation reordering and Case 2 to illustrate a lack of synchronizations of two goroutines.

Case 1, listed below, could legally print 2 and then 0, the reason being that the Go memory model does not guarantee that operations in goroutine f() are seen by goroutine g() in the same order.

```
package main

var a, b int

func f() {
        a = 1
        b = 2
}

func g() {
        print(b)
        print(a)
}

func main() {
        go f()
        g()
}
```

In Case 2, the goroutine is started but never synchronised back to main(), and thus, the Go memory model does not guarantee main will ever see the change in done.

```
package main

var done bool

func main() {
        go func() {
                done = true
        }()

        for !done {
        }
}
```

The most fundamental guarantee in the Go memory model is that **"within a single goroutine, the happens-before order is the order expressed by the program"**. That is, the Go memory model ensures that any optimisations within a goroutine must reproduce the results obtained by running the goroutine sequentially.

And the most fundamental lack of guarantee in the Go memory model is that **"a pair of goroutines are not guaranteed to share results, unless they are synchronised at some point".** The point and the kind of synchronisation defines which results are shared between the pair goroutines. In this document, only synchronisation via channels will be discussed. The following section introduces a scheme to visualise what portions of a pair of goroutines are visible at synchronisation. The compiler is allowed to alter the ordering of those portions that are invisible.

2. Synchronisation on:

2.1. Buffered Channels

The Go memory model establishes that goroutines synchronise at channel receive and it provides the guarantee that **"a send on a channel happens before the corresponding receive from that channel completes"**. These guarantee not only defines the synchronisation point but also the visibility properties. Let us visualise what portions are visible with an easy example of a Go program that synchronises the goroutines main() and f().

```
package main

var ch = make(chan int, 1)
var s string

func f() {
        s = "hello"
        ch <- 0
}

func main() {
        go f()
        <-ch
```

```
        print(s)
}
```

The visualisation procedure goes as follows:

I) Highlight in red colour the synchronisation point, in this case the channel send and receive.

II) Highlight in yellow colour the statements that are guaranteed by the Go memory model to be visible. In this case, since the channel send happens-before the corresponding channel receive, this means, only the statements above the channel send and below the channel receive are visible.

III) Shadow in grey colour the remaining statements in both goroutines.

func f() {	**func main()** {
s = "hello"	go f()
ch <- 0	<-ch
}	print(s)
	}

The conclusion for this example is that the Go memory model guarantees the statement s=”hello” in goroutine f() is visible by the statement print(s) in goroutine main(). Hence, everything works as expected.

Let us now consider an example that suffers from a race condition. The colouring scheme follows the same procedure described above. In this case, both statements s=”hello” and print(s) happen in the invisible area (grey colour) and hence there is no guarantee that the compiler has not altered the ordering of those statements. That is, although for goroutine f() s=”hello” happens-before <-ch, goroutine main() could potentially see that s=”hello” has not happened before <-ch.

```
package main

var ch = make(chan int, 1)
var s string

func init() {
        ch <- 0
}

func f() {
        s = "hello"
        <-ch
}

func main() {
        go f()
        ch <- 0
        print(s)
}
```

func f() {	**func main()** {
s = "hello"	go f()
<-ch	ch <- 0
}	print(s)
	}

The visualization scheme helps identify the statements the programmer cannot trust that have been executed in the same order in both goroutines. In the following sections, the same colouring scheme will be applied.

2.2. Unbuffered Channels

The Go memory model for unbuffered channels guarantees not only that **"a send on a channel happens before the corresponding receive from that channel completes"**, but also that **"a receive from an unbuffered channel happens before the send on that channel completes"**. That is, the happens-before relationship is guaranteed in both directions, and hence the statement above and below the channel send and receive are reciprocally visible. Following is an example to demonstrate how this applies to the present colouring scheme:

```
package main

var ch = make(chan int)
var s string

func f() {
        s = "hello"
        ch <- 0
}

func main() {
        go f()
        <-ch
        print(s)
}
```

func f() {	**func main()** {
s = "hello"	go f()
ch <- 0	<-ch
}	print(s)
	}

2.3. Channel close()

Let us now consider a third case of channel synchronisation. The Go memory model states that **"the closing of a channel happens before a receive that returns a zero value because the channel is closed"**. In terms of the

present colouring scheme, this guarantee translates as:

I) Highlight in red colour the synchronisation point, in this case the **channel close** and **receive**.

II) Highlight in yellow colour the statements that are guaranteed by the Go memory model to be visible. In this case, since the **channel close** happens-before the corresponding **channel receive**, this means, only the statements above the **channel close** and below the **channel receive** are visible.

III) Shadow in grey colour the remaining statements in both goroutines.

The following example illustrates the colouring scheme with Go program that suffers from a race condition, as indicated by the fact that the statements s=”hello” and print(s) are now coloured in grey.

```
package main

var ch = make(chan int, 1)
var s string

func f() {
        s = "hello"
        <-c h
}

func main() {
        go f()
        close(c h)
        print(s)
}
```

func f() {	**func main()** {
s = "hello"	go f()
<-ch	close(ch)
}	print(s)
	}

3. Example: http://play.golang.org/p/UeHAShi_dS

This document concludes by applying the colouring scheme to an example that motivated the following discussion at the news group go-nuts:

https://groups.google.com/d/msg/golang-nuts/MDvnk1Ax7UQ/eQGkJJmOxc4J

After applying the colouring scheme to the example in **http://play.golang.org/p/UeHAShi_dS** it is possible to conclude that:

I) the example is guaranteed not to panic, on the grounds that n is only altered by goroutine sleeper() and the GO memory model guarantees that **“within a single goroutine, the happens-before order is the order**

expressed by the program”.

II) however, it is not guaranteed that main ever completes, because wg.Wait() in goroutine main() is in the invisible area (grey colour) that is not guaranteed to see any of the statements in goroutine sleeper().

```
package main

import (
	"sync"
	"sync/atomic"
	"time"
)

var wg sync.WaitGroup
var count int32
var limiter = make(chan bool, 1)

func main() {
	wg.Add(10)
	for i := 0; i < 10; i++ {
		limiter <- true
		go sleeper()
	}
	wg.Wait()
}

func sleeper() {
	if n := atomic.AddInt32(&count, 1); n != 1 {
		panic("unexpected!")
	}
	time.Sleep(time.Second)
	atomic.AddInt32(&count, -1)
	<-limiter
	wg.Done()
}
```

func sleeper() {	
if n := atomic.AddInt32(&count, 1); n != 1 {	
panic("unexpected!")	
}	**func main()** {
time.Sleep(time.Second)	wg.Add(10)
atomic.AddInt32(&count, -1)	for i := 0; i < 10; i++ {
<-limiter	limiter <- true
wg.Done()	go sleeper()
}	}
	wg.Wait()
	}