Bug 1

Problem Deadlock is caused since all go routines are asleep.

Solution Create a new go routine and make sure that the channel is closed once data is set.

Bug1.go

```go
package main

import "fmt"

// I want this program to print "Hello world!", but it doesn't work.
func main() {
	ch := make(chan string)
	go func() {
		ch <- "Hello world!"
		close(ch)
	}()

	fmt.Println(<-ch)

}
```

Bug 2

Problem We are not guaranteed that the Go routine will be able to go through the entire channel before the for statement is entirely executed. In GO, send occurs before receive.

Solution Once information was sent through the channel, we wait for a reply by the Go routine *Print*. When the text is printed on the screen, it sends a 1 through the channel. We know that send will occur before receive, hence the behaviour of the methods. In this way, we achieve synchronization.

```go
package main

import "fmt"

// This program should go to 11, but sometimes it only prints 1 to 10.
func main() {
	ch := make(chan int)
	//  c := make(chan bool)
	go Print(ch)
	for i := 1; i <= 11; i++ {
		ch <- i
		//c <- true // a send occurs before a receive.
		<-ch
	}

	close(ch)

}

// Print prints all numbers sent on the channel.
// The function returns when the channel is closed.
func Print(ch chan int) {
	for n := range ch { // reads from channel until it's closed
```

```
            fmt.Println(n)
            ch <- 1
            //<-c // receive occurs after send.
        }

}
```

Many Receivers

- Swapping `wgp.Wait()` with `close(ch)` will not allow us to send/receive any information through the channel before the go routines are complete. This will throw an error.
- Moving `close(ch)` out from the `main()` method to `Produce()` will throw an error (high probability) sometime during execution. Just because one GO routine completes does not imply that all are going to be complete at that instance in time.
- Nothing will happen if we remove `close(ch)` entirely as it will be garbage collected if not in use. It might be important to close if the receiver requires that.[1]
- By increasing the number of consumers from 2 to 4, the program will execute $200 - 300ms$ faster. That's because we have more consumers that can consume the data.
- No, we cannot take it for granted that all strings are going to be printed. We can be confident that all strings are going to be printed. That's because the program is going to wait while the produces produce data and this is going to be simultaneously processed by the consumers.

```
// Stefan Nilsson 2013-03-13

// This is a testbed to help you understand channels better.
package main

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

var count int = 0

func main() {
    // Use different random numbers each time this program is executed.
    rand.Seed(time.Now().Unix())

    const strings = 32
    const producers = 4
    const consumers = 2

    before := time.Now()
    ch := make(chan string)
    wgp := new(sync.WaitGroup)
    wgp.Add(producers)

```

[1]`http://stackoverflow.com/a/8593986`, last used 2015.03.29

```go
    wgc := new(sync.WaitGroup)
    wgc.Add(consumers)

    for i := 0; i < producers; i++ {
        go Produce("p"+strconv.Itoa(i), strings/producers, ch, wgp)
    }
    for i := 0; i < consumers; i++ {
        go Consume("c"+strconv.Itoa(i), ch, wgc)
    }
    wgp.Wait() // Wait for all producers to finish.
//  wgc.Wait()
    close(ch)
    wgc.Wait() // channel closed, process consumers.
    fmt.Println("time:", time.Now().Sub(before))
    fmt.Println("printed: ", count)
}

// Produce sends n different strings on the channel and notifies wg when
    done.
func Produce(id string, n int, ch chan<- string, wg *sync.WaitGroup) {
    for i := 0; i < n; i++ {
        RandomSleep(100) // Simulate time to produce data.
        ch <- id + ":" + strconv.Itoa(i)
    }
    wg.Done()
}

// Consume prints strings received from the channel until the channel is
    closed.
func Consume(id string, ch <-chan string, wg *sync.WaitGroup) {
    for s := range ch {
        fmt.Println(id, "received", s)
        count++
        RandomSleep(100) // Simulate time to consume data.
    }

    wg.Done()

    // it's strange that we cannot put wg.Done() here instead
    // and letting the wgc be the no. of consumers.
    //  wg.Done()
}

// RandomSleep waits for x ms, where x is a random number, 0 < x < n,
// and then returns.
func RandomSleep(n int) {
    time.Sleep(time.Duration(rand.Intn(n)) * time.Millisecond)
}
```

Oracle

```go
// Stefan Nilsson 2013-03-13

// This program implements an ELIZA-like oracle
    (en.wikipedia.org/wiki/ELIZA).
package main

import (
    "bufio"
    "fmt"
    "math/rand"
    "os"
    "strings"
    "time"
)

const (
    star   = "Pythia"
    venue  = "Delphi"
    prompt = "> "
)

func main() {
    fmt.Printf("Welcome to %s, the oracle at %s.\n", star, venue)
    fmt.Println("Your questions will be answered in due time.")

    oracle := Oracle()
    reader := bufio.NewReader(os.Stdin)
    for {
        fmt.Print(prompt)
        line, _ := reader.ReadString('\n')
        line = strings.TrimSpace(line)
        if line == "" {
            continue
        }
        fmt.Printf("%s heard: %s\n", star, line)
        oracle <- line // The channel doesn't block.
    }
}

// Oracle returns a channel on which you can send your questions to the
    oracle.
// You may send as many questions as you like on this channel, it never
    blocks.
// The answers arrive on stdout, but only when the oracle so decides.
// The oracle also prints sporadic prophecies to stdout even without being
    asked.
func Oracle() chan<- string {
    questions := make(chan string, 1)
    // TODO: Answer questions.
    // TODO: Make prophecies.
    // TODO: Print answers.

    answer := make(chan string, 1)

    go func() {
        for {
            prophecy(<-questions, answer)
        }
    }()

    go func() {
        for {
            prophecy("", answer)
        }
    }()

    go func() {
        for {
            fmt.Println(<-answer)
        }
    }()

    return questions

}

// This is the oracle's secret algorithm.
// It waits for a while and then sends a message on the answer channel.
// TODO: make it better.
func prophecy(question string, answer chan<- string) {
    // Keep them waiting. Pythia, the original oracle at Delphi,
    // only gave prophecies on the seventh day of each month.
    time.Sleep(time.Duration(20+rand.Intn(10)) * time.Second)

    // Find the longest word.
    longestWord := ""
    words := strings.Fields(question) // Fields extracts the words into a slice.
    for _, w := range words {
        if len(w) > len(longestWord) {
            longestWord = w
        }
    }

    // Cook up some pointless nonsense.
    nonsense := []string{
        "The moon is dark.",
        "The sun is bright.",
    }
    answer <- longestWord + "... " + nonsense[rand.Intn(len(nonsense))]
}

func init() { // Functions called "init" are executed before the main function.
    // Use new pseudo random numbers every time.
    rand.Seed(time.Now().Unix())
}
```