Golang

2 August 2016

Apoorva M
Developer,
ThoughtWorks

Plan

- What's Golang?
- Why do people choose Go?
- What makes Go fast?
- Goroutine vs Threads
- Closing thoughts

Go Programming Langauge

- Open source project by Google
- Its an expressive, fast, statically typed language
- Known for its Concurrency mechanisms
- Compiled language
- Has Garbage Collection

Examples

Simple Static Webserver

```
package main

import (
    "log"
    "net/http"
)

func main() {
    // Simple static webserver
    log.Fatal(http.ListenAndServe(":8080", http.FileServer(http.Dir("/usr/share/doc"))))
}
```

Run

Anonymous functions - Closures

Go supports anonymous functions, which can form closures.

```
package main

import "fmt"

func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}
func main() {
    nextInt := intSeq()

    fmt.Println(nextInt())
    fmt.Println(nextInt())
    fmt.Println(nextInt())

    newInts := intSeq()
    fmt.Println(newInts())
}
```

Run

Goroutines

A goroutine is a lightweight thread of execution.

```
package main
import (
    "fmt"
    "time"
)
func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
        time.Sleep(time.Microsecond)
    }
}
func main() {
    f("direct")

    go f("goroutine1")
    go f("goroutine2")

    time.Sleep(time.Second * 2)
}
```

Run

Communicate between goroutines

Channels

```
func main() {
    var Ball int
    table := make(chan int)

    go player("Player 1", table)
    go player("Player 2", table)

    table <- Ball
    time.Sleep(1 * time.Second)
    <-table
}
func player(playerName string, table chan int) {
    for {
        ball := <-table
        fmt.Println(playerName)
        ball++
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

Run

Visualization (https://divan.github.io/demos/pingpong/)

Why do people use Go?

Few common reasons are

- Concurrency
- Ease of deployment
- Performance

What makes Go fast?

Source:

- Five things that make go fast (http://dave.cheney.net/2014/06/07/five-things-that-make-go-fast)

Values

```
var foo int32 = 1234 // consumes 4 bytes of memory
var bar int = 2016   // consumes 8 bytes of memory

~ python
>>> from sys import getsizeof
>>> foo = 1234
>>> getsizeof(foo)
24

Integer foo = new Integer(1234); // Java: consumes 16 or 24 bytes
```

- Go lets you create compact data structures, avoiding unnecessary redirection
- Compact data structures utilize cache better
- Better cache utilization leads to better performance

Inlining

- Function calls are not free. Optimization technique to reduce overhead is Inlining
- Dead code elimination

```
package main

func Test() bool {
    return false
}

func Expensive() {
    if Test() {
        // unreachable code
    }
}
```

Run

```
func Expensive() {
    if false {
        // unreachable code
    }
}
```

Escape Analysis

- Escape analysis determines whether any references to a value escape the function in which the value is declared.
- If no references escape, the value may be safely stored on the stack.
- Values stored on the stack do not need to be allocated or freed.

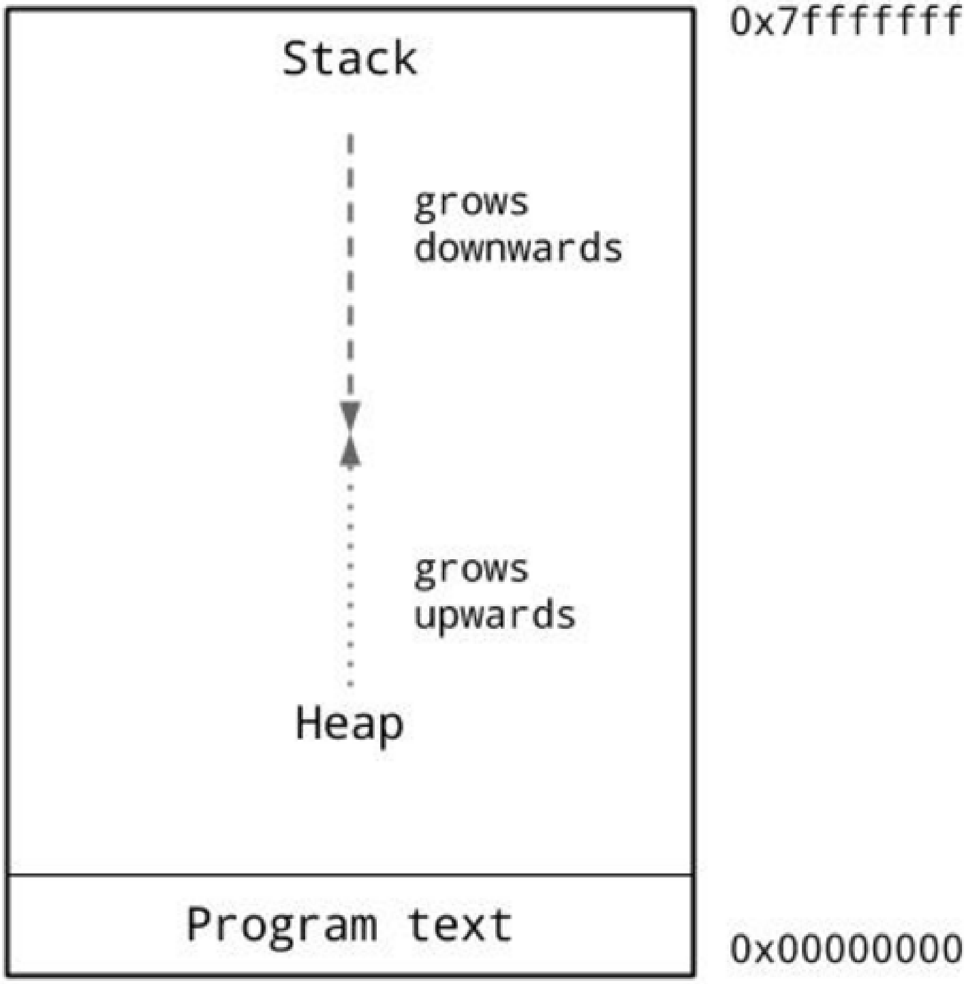
```
func print() {
   numbers := []int{1,2,3,4,5} // numbers never escape print()
   for _, num := range numbers {
       fmt.Println(num)
   }
}
```

Goroutines

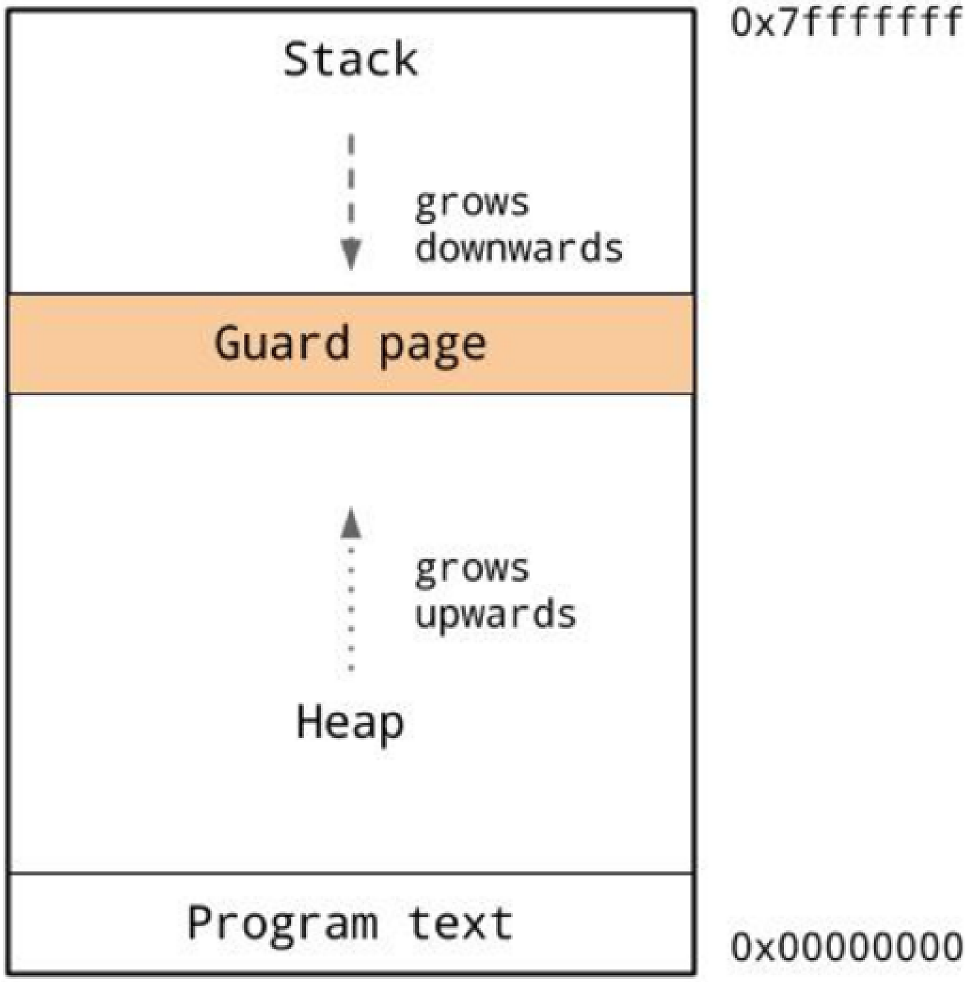
How's it different from Threads?

- Memory Consumption
- Setup and teardown
- Switching time

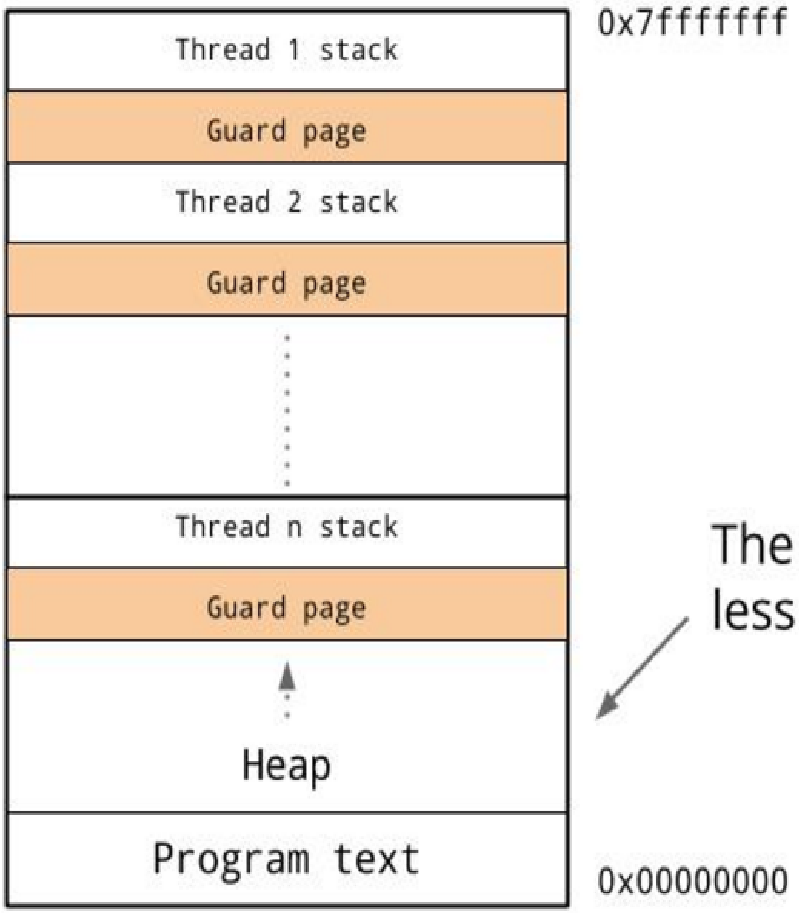
Process address space



Guard page


Stack
grows
downwards
Guard page
grows
upwards
Heap
Program text
0x7fffffff
0x00000000

Thread stacks and Guard pages



The more threads in your program, the less heap is available

Memory Consumption

- Threads start out at 1MB, along with Guard page
- The creation of a goroutine does not require much memory - only 2KB of stack space

Setup and Teardown costs

- Threads have significant setup and teardown costs because it has to request resources from the OS and return it once its done.
- Goroutines are created and destroyed by the runtime

Switching costs

- During a thread switch, the scheduler needs to save all registers
- Goroutines are cooperatively scheduled, rather than relying on the kernel to manage their time sharing.

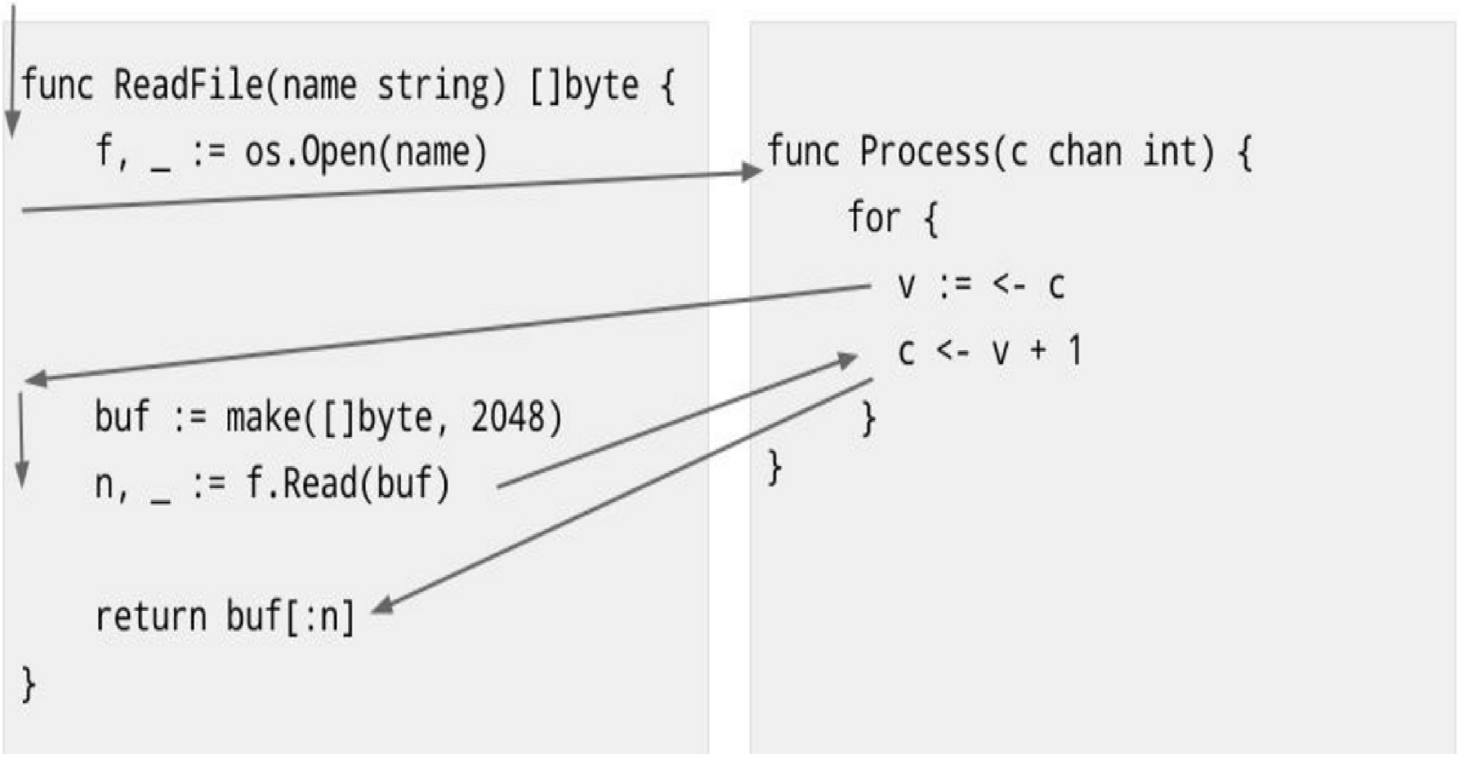
How goroutines are executed?

- Go runtime is allocated a few threads on which all the goroutines are multiplexed.
- At any point of time, each thread will be executing one goroutine.
- If that goroutine is blocked, then it will be swapped out for another goroutine that will execute on that thread instead.

When does switch between goroutines happen?

- Channel send and receive operations
- Blocking syscalls like file and network operations.
- Go statement
- Garbage collection

Example



```
func ReadFile(name string) []byte {
    f, _ := os.Open(name)



    buf := make([]byte, 2048)
    n, _ := f.Read(buf)

    return buf[:n]
}
```

```
func Process(c chan int) {
    for {
        v := <- c
        c <- v + 1
    }
}
```

This results in relatively few operating system threads per Go process, with the Go runtime taking care of assigning a runnable Goroutine to a free operating system thread.

Goroutine stacks

- No guard pages
- Check for available stack space is done as part of the function call
- Initial stack size is small
- Grows as needed

Closing thoughts

- I think Golang is a great langauge for building cross platform utilities.
- Values, Inlining, Escape Analysis, Goroutines, copying stacks etc are a few things which make Go fast.
- These features are powerful individually, they do not exist in isolation.
- As with other languages, it is important to prevent simultaneous access of shared resources by more than one goroutine.

Do not communicate by sharing memory; instead, share memory by communicating.

Thank you

Apoorva M
Developer,
ThoughtWorks
@ItsApoorvaHere (http://twitter.com/ItsApoorvaHere)