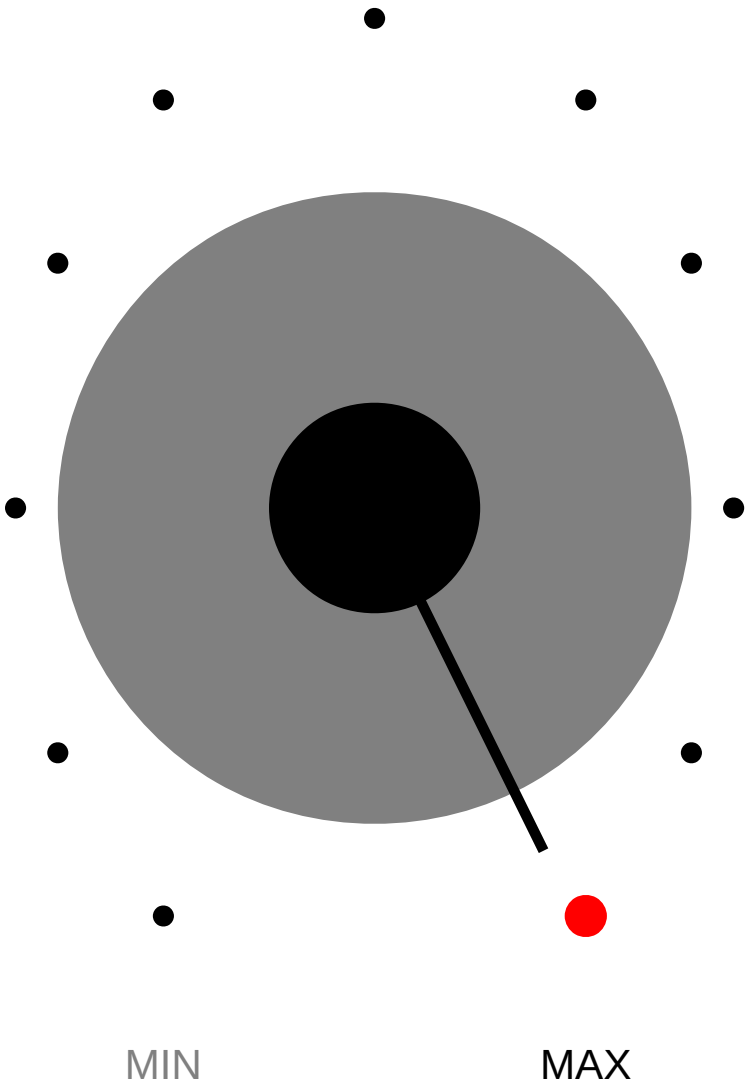
Why Choose Go?

Concurrency

Ease of deployment

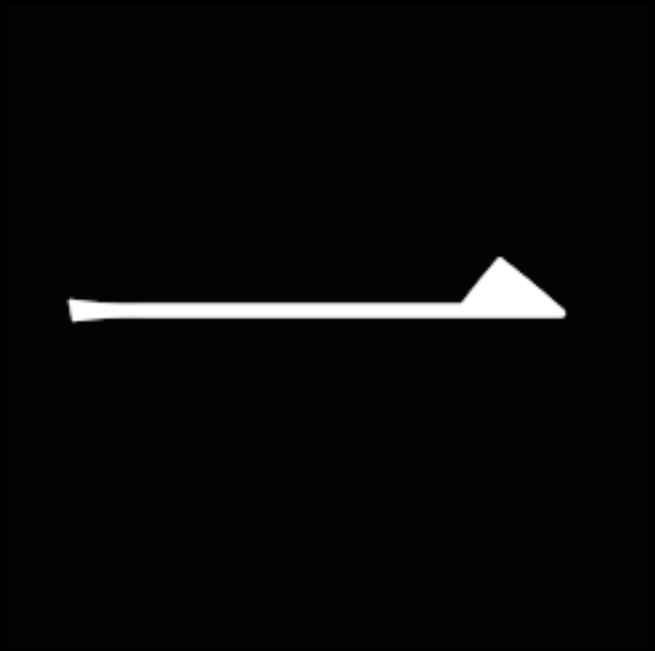
Performance



Performance

一から五

Five things that make Go fast



Values

Go

```
var gocon int32 = 2014
```

Python

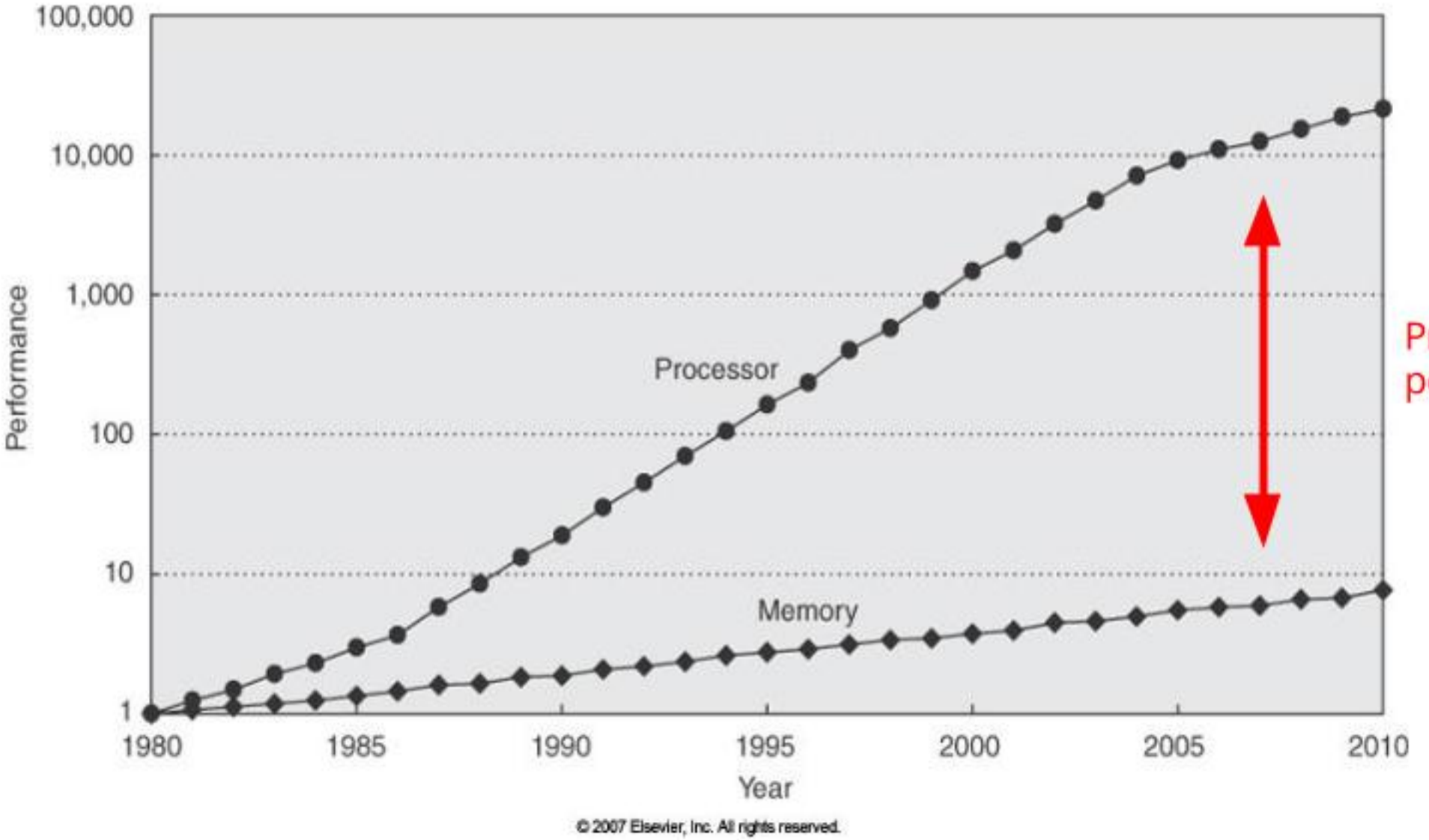
```
% python
>>> from sys import getsizeof
>>> gocon = 2014
>>> getsizeof(gocon)
24
```

Java

```
int gocon = 2014;
```

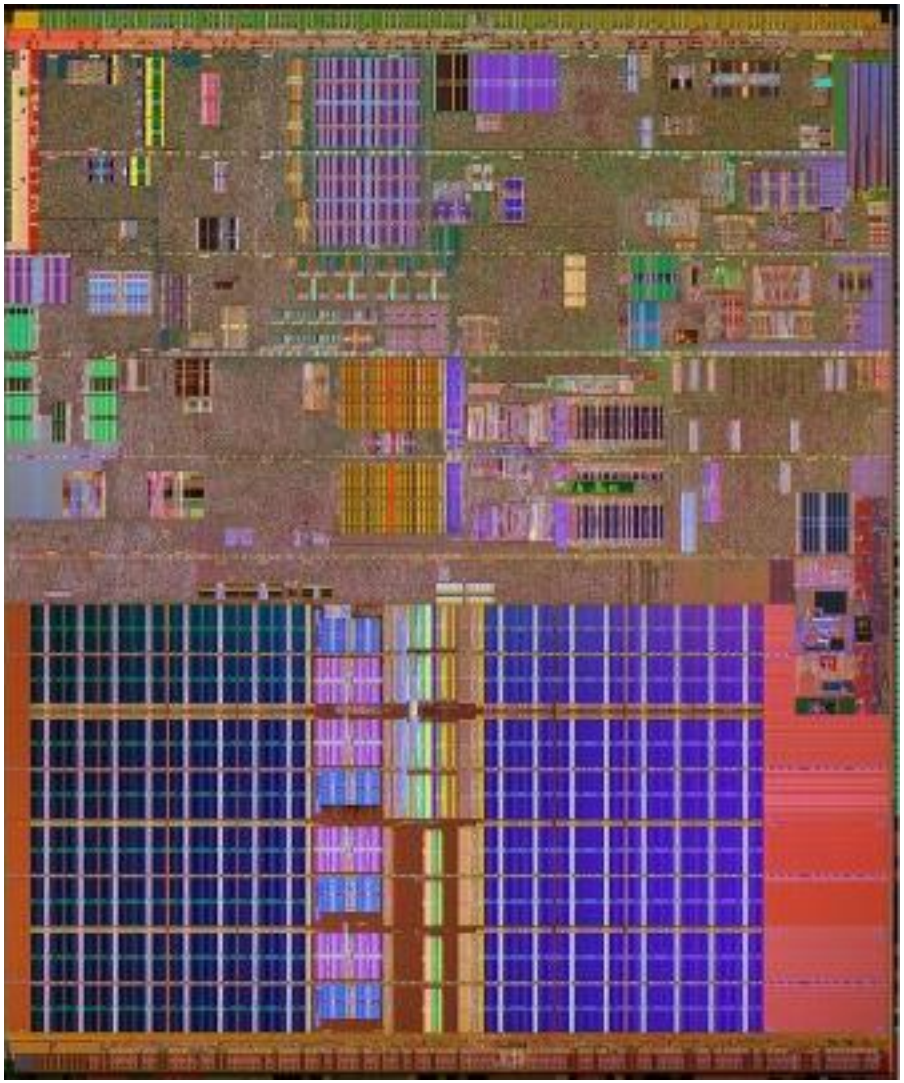
Java

```
// 16 bytes on 32 bit JVM
// 24 bytes on 64 bit JVM
Integer gocon = new Integer(2014);
```



Memory speed lags behind CPU speed

CPU Cache

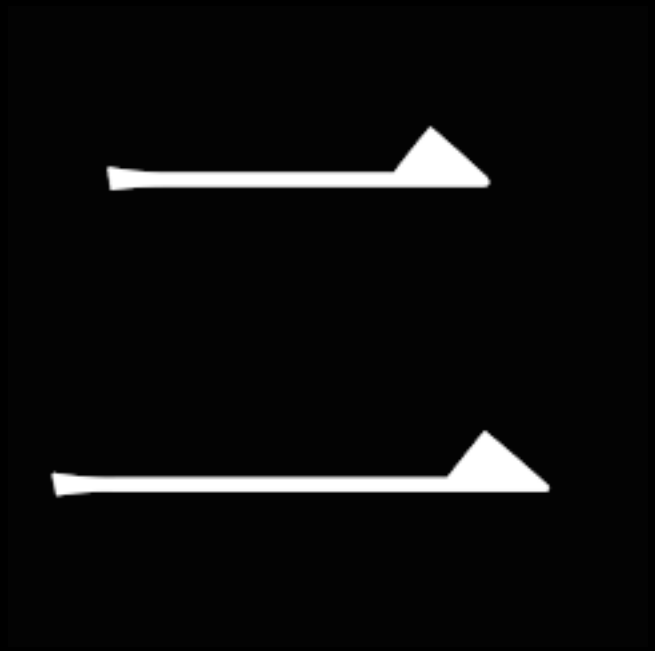


Values example

```
// Location is a point in a three dimensional space
type Location struct {
    // 8 bytes per float64
    // 24 bytes in total
    X, Y, Z float64
}

// Locations consumes 24 * 1000 bytes
var Locations [1000]Location
```

Values

Go lets you create compact data structures, avoiding unnecessary indirection, which use the cache better, leading to better performance.



Inlining

Function call procedure

1. Create new stack frame
2. Record the return address of the caller
3. Save registers that may be overwritten during the function call
4. Compute the function address
5. Branch to the computed address

Function calls have an unavoidable overhead

The Go compiler inlines a function by treating the body of the function as if it were part of the caller.

Inlining example

```
package util

// Max returns the larger of a or b.
func Max(a, b int) int {
    if a > b {
        return a
    }
    return b
}
```

```
package main

import "util"

// Double returns twice the value of the larger of a or b.
func Double(a, b int) int { return 2 * util.Max(a, b) }
```

After inlining

```
func Double(a, b int) {
    temp := b
    if a > b {
        temp = a
    }
    return 2 * temp
}
```

Contents of util.Max copied into Double

util.a

```
% strings ~/pkg/linux_amd64/util.a
...
package util
    import runtime "runtime"
    func @"".Max (@"".a
2 int , @"".b
3 int) (? int) { if @"".a
2 > @"".b
3 { return @"".a
2 }; return @"".b
...
```

Dead code elimination

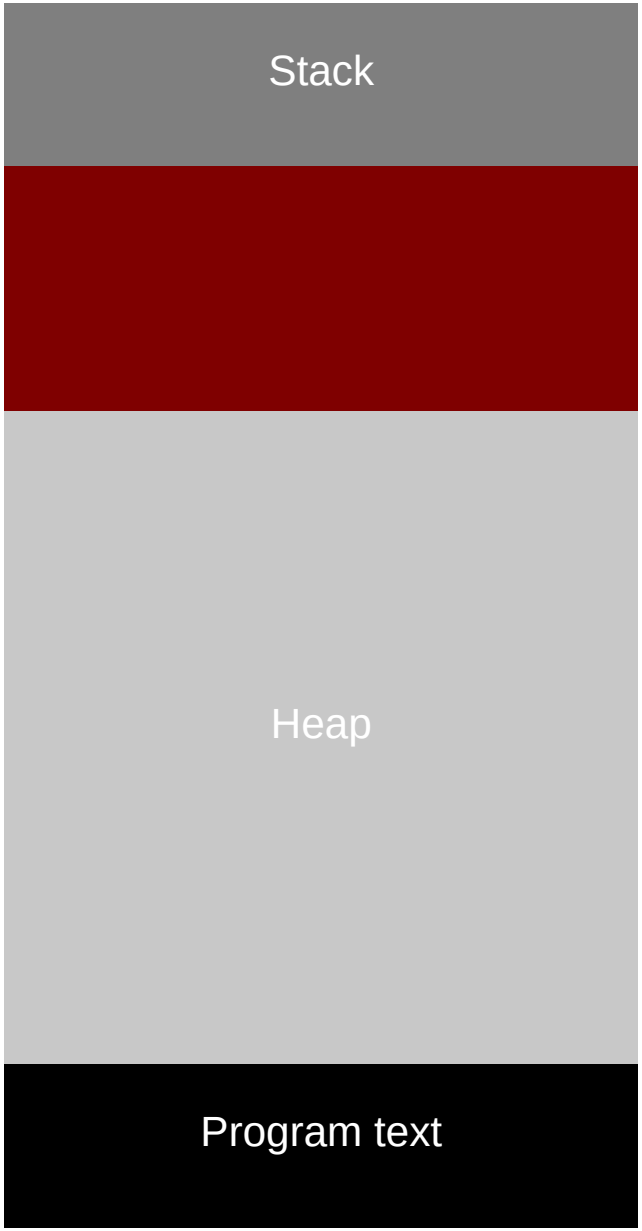
```
func Test() bool { return false }

func Expensive() {
    if Test() {
        // something expensive
    }
}
```

Dead code elimination

```
func Expensive() {
    if false {
        // something expensive is now unreachable
    }
}
```

Escape analysis

Process address space



`0x7fffffff`

`0x00000000`

Escape analysis

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

Escape analysis example

```
// Sum returns the sum of the numbers 1 to 100.
func Sum() int {
    numbers := make([]int, 100)
    for i := range numbers  {
        numbers[i] = i + 1
    }
    var sum int
    for _, i := range numbers {
        sum += i
    }
    return sum
}
```

numbers never escape Sum()

Escape analysis example

```
const Width, Height = 640,480

type Cursor struct {
    X, Y int
}

func Center(c *Cursor) {
    c.X += Width / 2
    c.Y += Height / 2
}

func CenterCursor() {
    c := new(Cursor)
    Center(c)
    fmt.Println(c.X, c.Y)
}
```

Center does not retain a reference to c

c created with new, not visible outside of CenterCursor, allocated on the stack

Escape analysis in action

```
% go build -gcflags=-m esc.go
# command-line-arguments
./esc.go:26: can inline Center
./esc.go:33: inlining call to Center
./esc.go:6: Sum make([]int, 100) does not escape
./esc.go:26: CenterCursor new(Cursor) does not escape
./esc.go:34: NewPoint ... argument does not escape
```

Show escape analysis and inlining info

Center() inlined into CenterCursor()

make([]int, 100)

func Center(c *Cursor)

c := new(Cursor)

fmt.Println(c.X, c.Y)

Goroutines

Process switching cost

- Saving and restoring all CPU registers
- Reconfiguring the memory management unit
- Switch into kernel space
- Scheduler overhead

Processor registers

amd64 (up to 64 bits each)

```
RAX, RBX, RCX, RDX, RSP, RBP,
RSI, RDI, R8, R9, R10, R11,
R12, R13, R14, R15, RIP
```

MMX (64 bits each)

```
MM0, MM1, MM2, MM3, MM4, MM5,
MM6, MM7
```

387 floating point (32 bits each)

```
F0, F1, F2, F3, F4, F5, F6, F7
```

SE{2,3,4} (128 bits each)

```
XMM1, XMM2, XMM3, XMM4, XMM5,
XMM6, XMM7, XMM8, XMM9, XMM10,
XMM11, XMM12, XMM13, XMM14,
XMM15
```

Threads

Many threads can share the same address space. Creation and switching are faster compared to indidivual processes.

Goroutines

Goroutines are cooperatively scheduled, with switching occuring only at well defined points. The compiler knows the registers in use, and saves them automatically

Goroutine scheduling points

- Channel send and receive
- go statement
- Blocking system call
- Garbage collection

Goroutine 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
    }
}
```

Goroutine 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
    }
}
```

Goroutine 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
    }
}
```

Goroutine 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
    }
}
```

Goroutine 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
    }
}
```

Goroutine 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
    }
}
```

Goroutine 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
    }
}
```

Goroutine example

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


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

    return buf[:n]
}
```

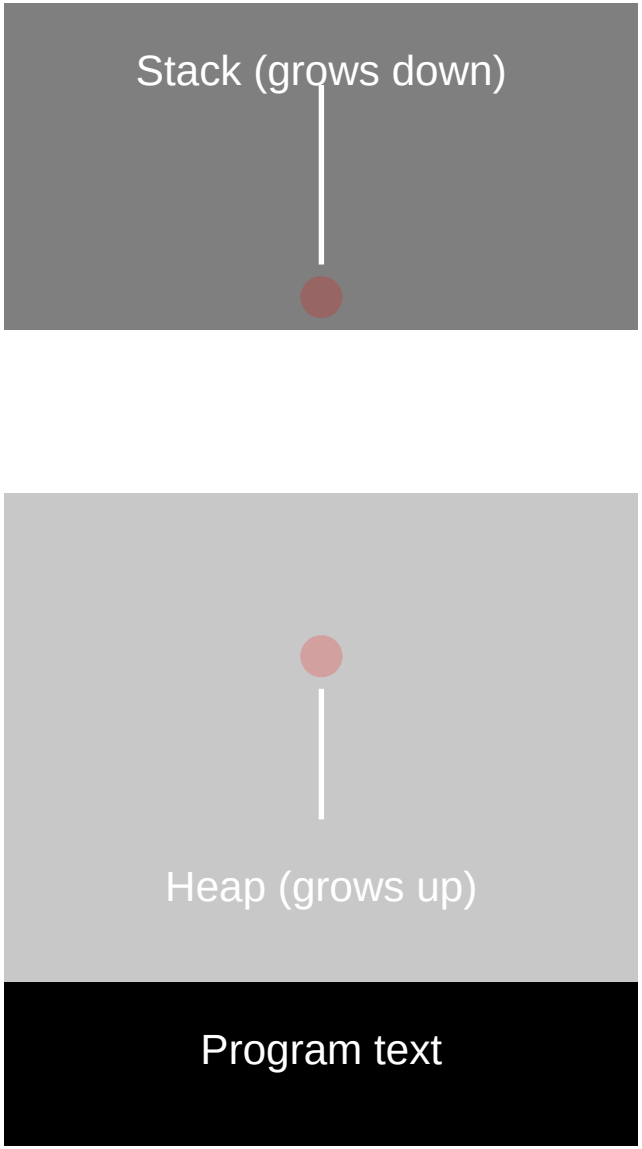
```
func Process(c chan int) {
    for {
        v := <- c
        c <- v + 1
    }
}
```

Blocking syscalls

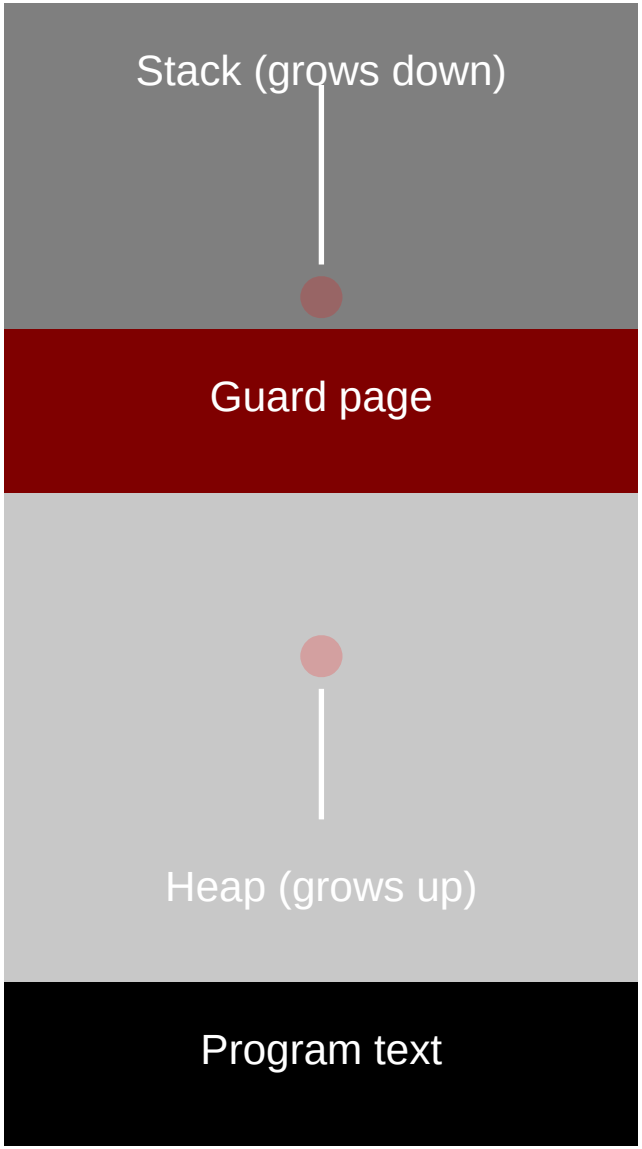
```
// package syscall
TEXT    ·Syscall(SB), NOSPLIT,$0-56
        CALL    runtime·entersyscall(DB)
        ...
        MOVQ    8(SP), AX    // syscall number
        SYSCALL
        ...
        CALL    runtime·exitsyscall(SB)
        RET
```

Segmenting and copying stacks

Process address space



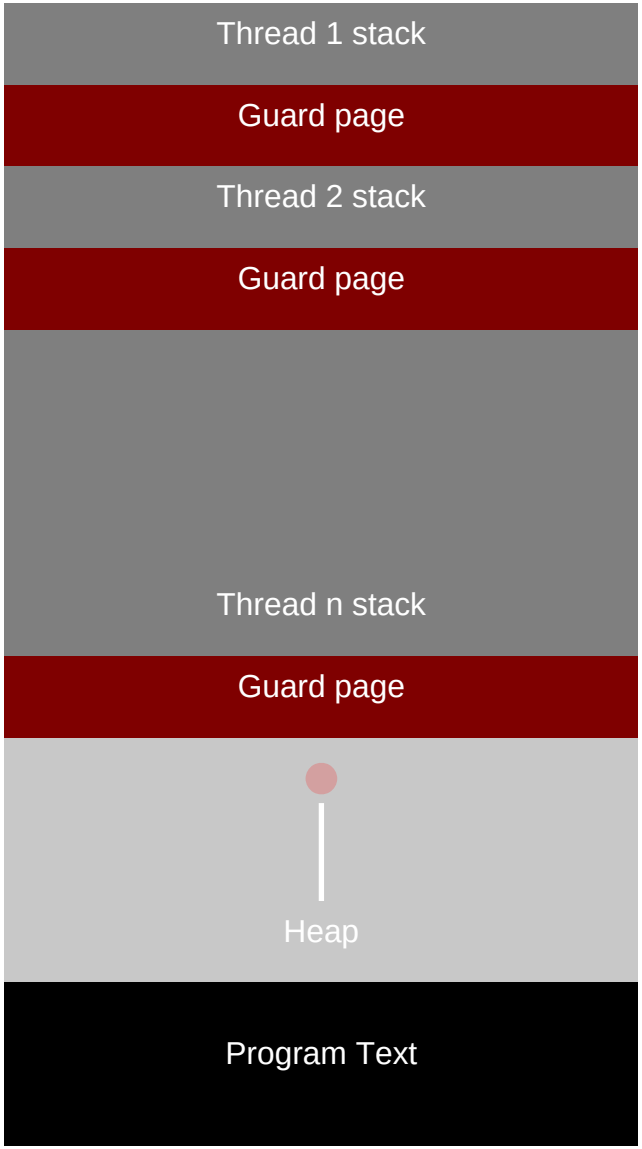
Guard page



0x7fffffff

0x00000000

Thread stacks and guard pages



`0x7fffffff`

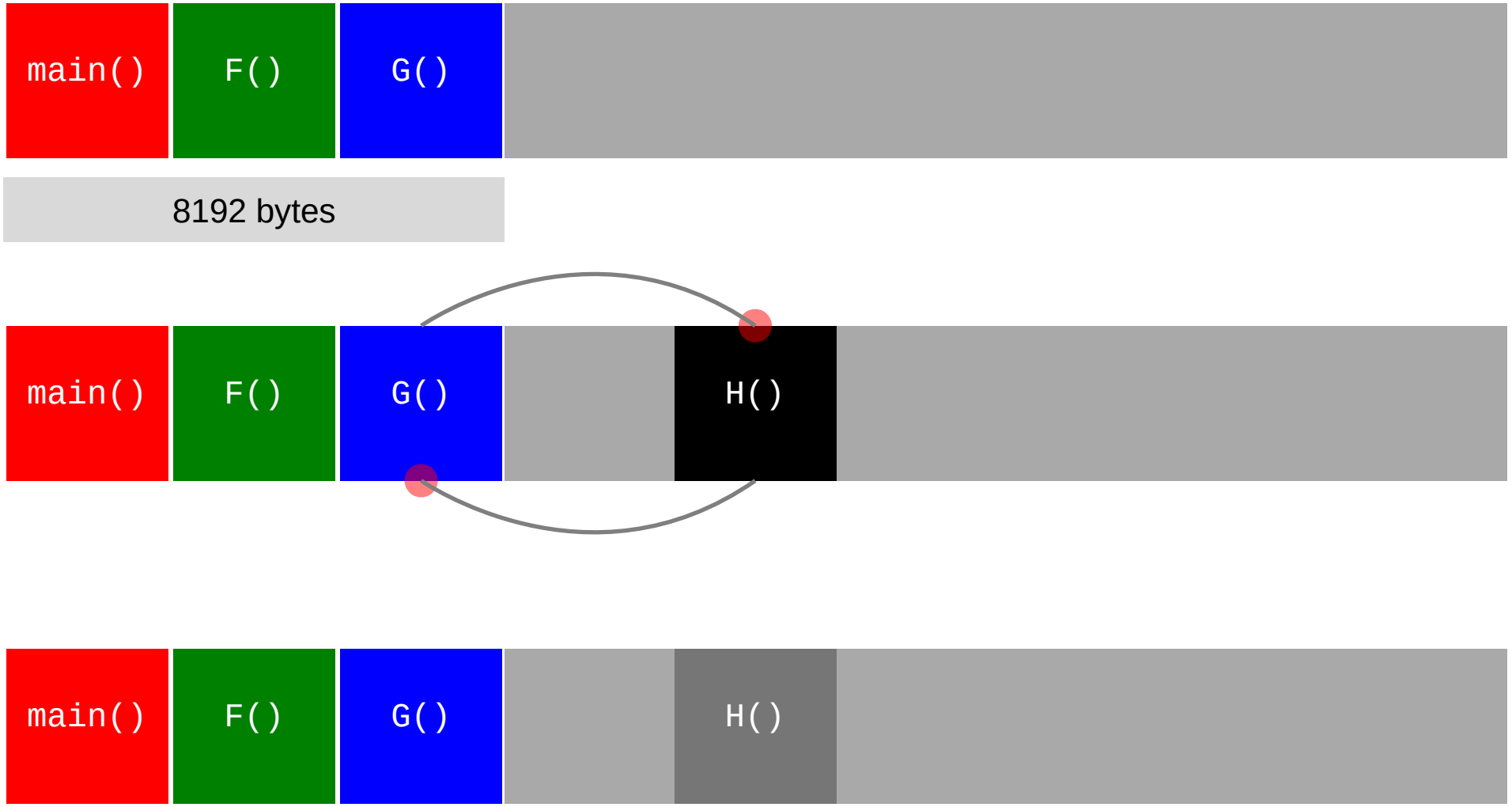
`0x00000000`

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

Goroutine stacks

- No guard pages
- Check for available space as part of the function call
- The initial stack is very small, currently 8kb
- Grow as needed

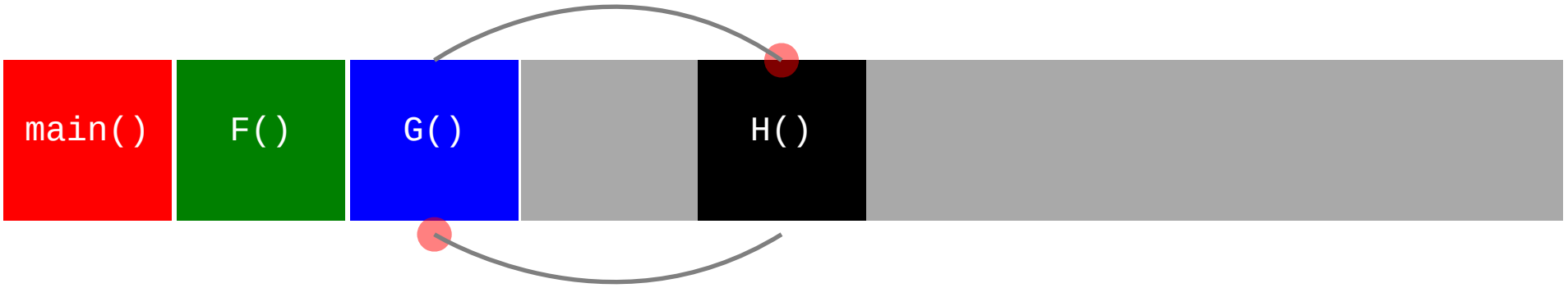
Segmented stacks (Go 1.0 - 1.2)



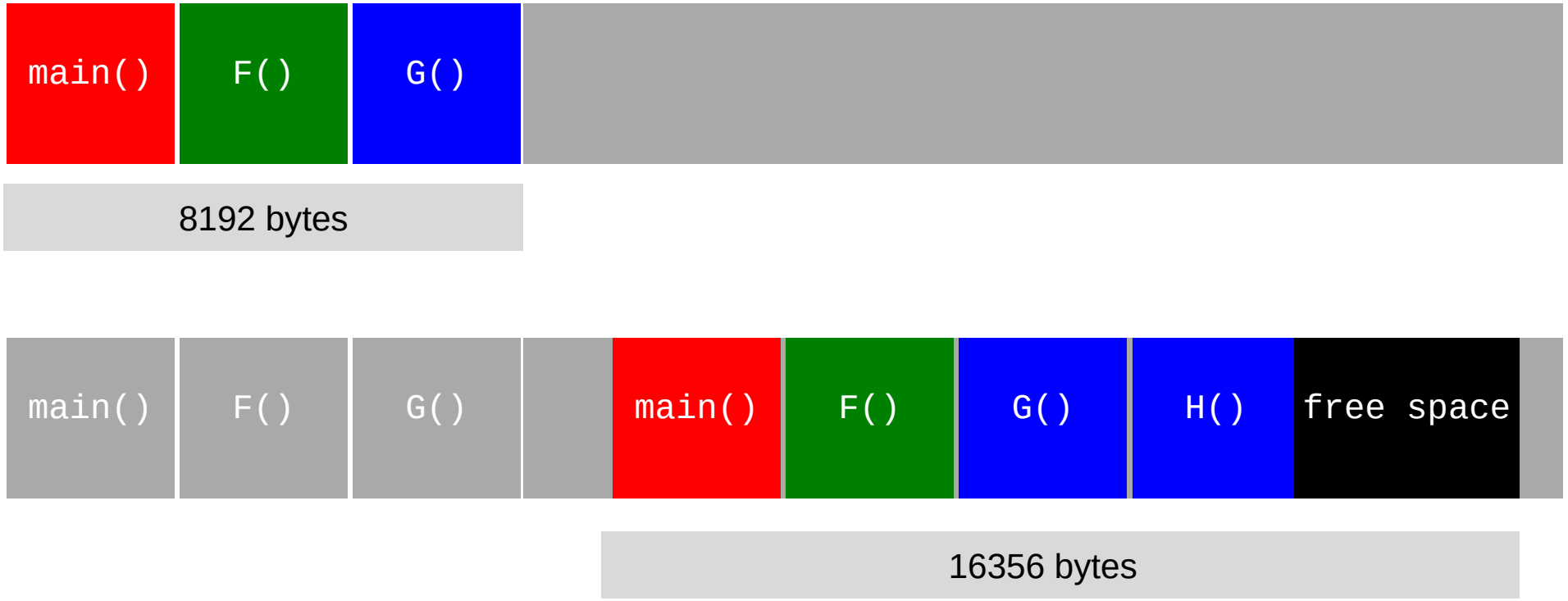
Hot split problem

```
func G(items []string) {
    for item := range items {
        H(item)
    }
}
```

New stack segment created and deleted on each call to H()



Copying stacks (Go 1.3)



Values

Inlining

Escape Analysis

Goroutines

Copying Stacks

Thank you

Thank you to the Gocon organiers for allowing me to speak today.

Thank you to Josh Bleecher Snyder, Bill Kennedy and Minux for their assistance in preparing this talk.

@davecheney http://dave.cheney.net dave@cheney.net