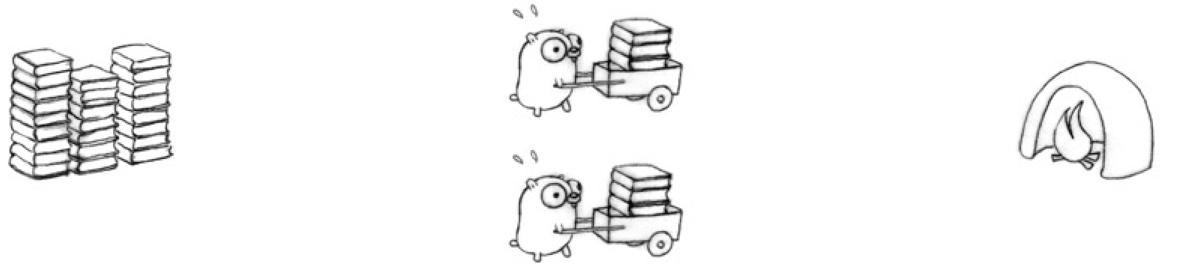
Concurrency

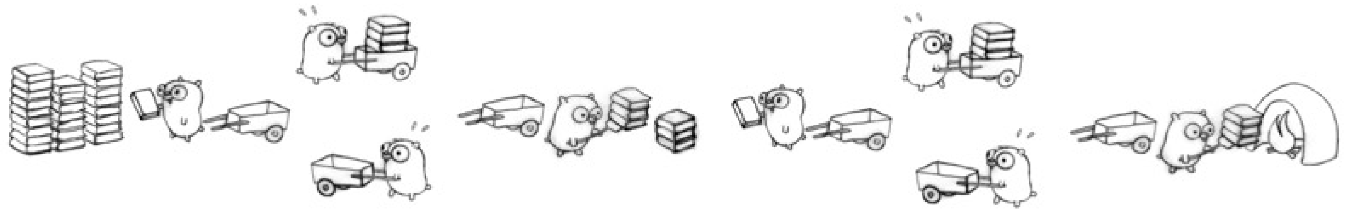
How it runs and where to Go

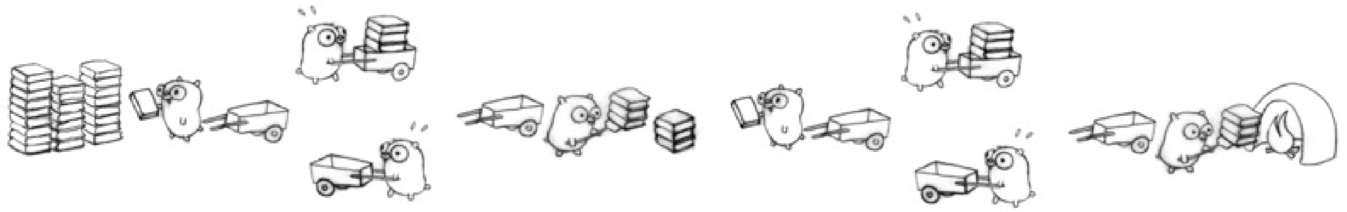
20 July 2016

Andrew Pogrebnoy
Technical Lead, Admobitec Ltd.

Concurrency is not parallelism







The roots

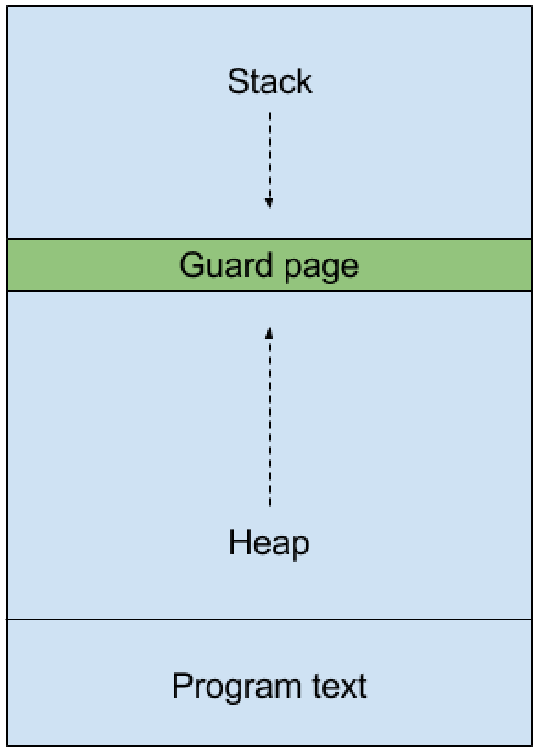
- Context switching

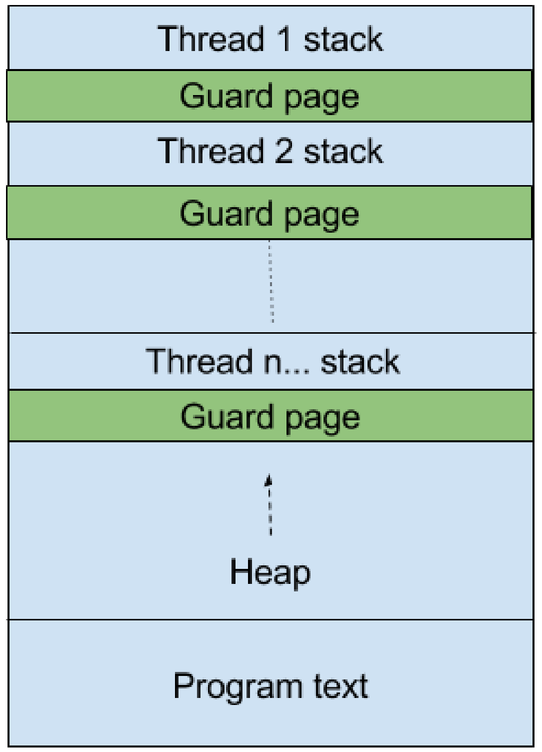
Kernel switches the CPU attention between processes.
Has huge overhead, because of need to storing/restoring all the CPU registers for processes.
Switches unpredictable (can occur at any point in an execution).

- Threads

Threads are the same processes but share common memory space. Easier to switches between. But still expensive.

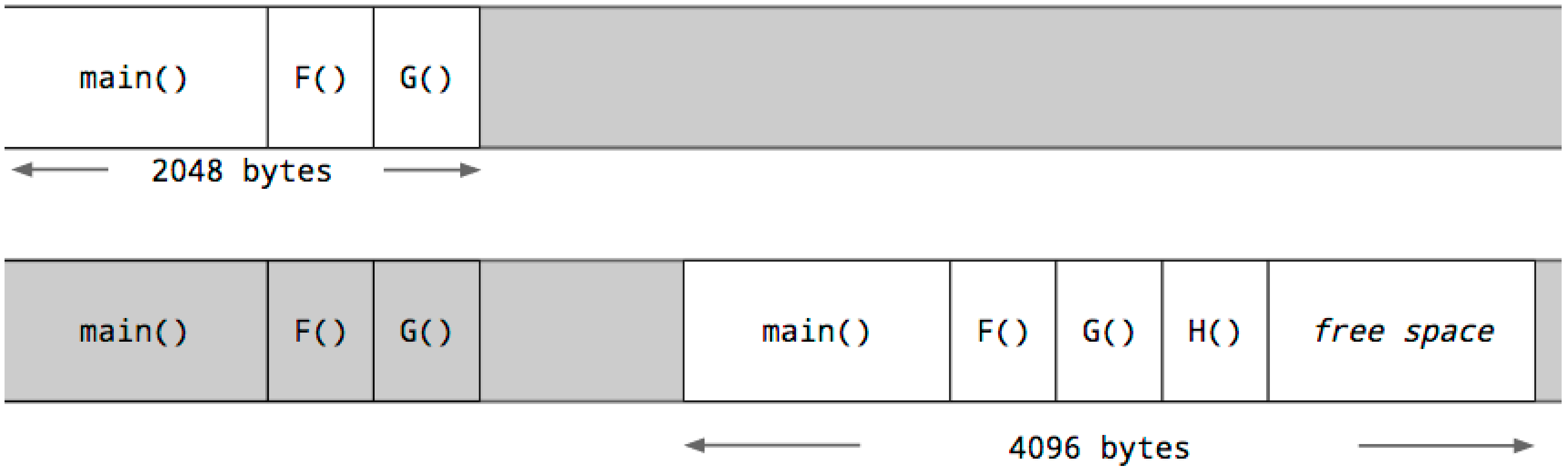
Stack


Stack
Guard page
Heap
Program text


Thread 1 stack
Guard page
Thread 2 stack
Guard page
Thread n... stack
Guard page
Heap
Program text

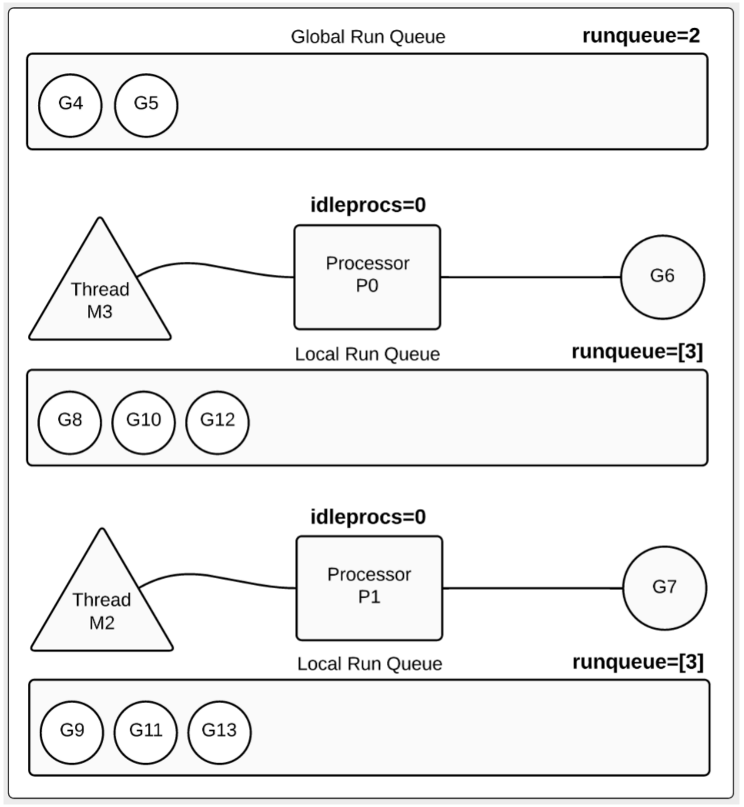
Goroutines

- Goroutine starts with the 2k stack on a heap. Compiler inserts check in every function call. And if there are not enough free space, it allocates new segment and moves contents there. So it can start with a small stack. Wich, in turn, can even be shrunk by GC.



- Cooperatively scheduled by the runtime. Cheap to create and manage. Runtime switches between at well-defined points. Unlike OS, which can switch threads at any time.

Scheduler

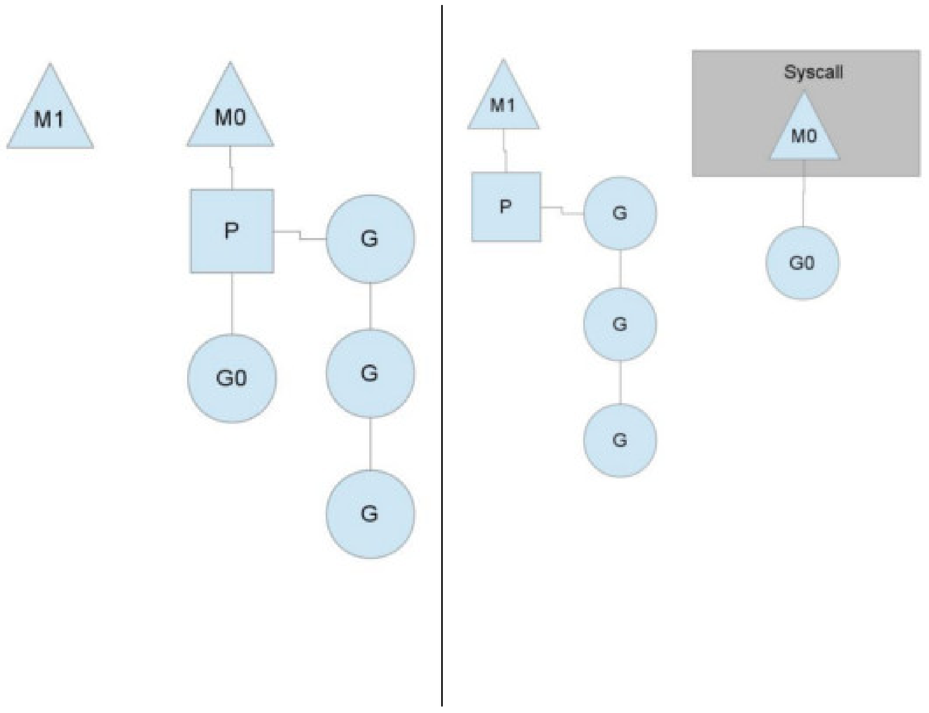


taken from: Scheduler Tracing In Go (https://www.goinggo.net/2015/02/scheduler-tracing-in-go.html)

M - OS thread, **P** - context of execution, **G** - gouroutine.

Scheduler

- **P** = **GOMAXPROCS**
- **M** s can be much more than **P**
- Work stealing



taken from: The Go scheduler (https://morsmachine.dk/go-scheduler)

Happens before

Compiler and processors can reorder reads and writes in single goroutine, in case it doesn't change program behavior.

- A read *r* of a variable *v* is allowed to observe a write *w* to *v* if both of the following hold:

```
1. `r` does not happen before `w`.
2. There is no other write `w` to v that happens after `w` but before `r`.
```

- That is, *r* is guaranteed to observe w if both of the following hold:

```
1. `w` happens before `r`.
2. Any other write to the shared variable `v` either happens before `w` or after `r`.
```

Both conditions equivalent if there is no concurrency, but otherwise the second is stronger.

The goroutine creation happens before it's execution begins.
But the exit of a goroutine is not guaranteed to happen before any event in the program.

```go
package main

import "fmt"

var a string

func f() {
    a = "hello"
}

func main() {
    go f()
    fmt.Print(a)
}
```

Run

the assignment to a is not followed by any synchronization event, so it is not guaranteed to be observed by any other goroutine.

Channels

- The main method of communication between goroutines in go
- Has two main operations: *send* and *recieve*. Also, it has *close*
- Operations on an *unbuffered* channels are blocking.
- Buffered channel blocks *receiving* when a queue is empty and *sending* when it's full.
- A send on a channel happens before the corresponding receive from that channel completes.
- A receive from an *unbuffered* channel happens before the send on that channel completes.
- The closing of a channel happens before a receive that returns a zero value because the channel is closed.

Channels

```
package main

import "fmt"

var a string
var c = make(chan struct{})

func f() {
    a = "hello"
    c <- struct{}{}
}

func main() {
    go f()
    <-c
    fmt.Print(a)
}
```

Run

Mutex

- Mutex
- RWMutex
- For any sync.Mutex or sync.RWMutex variable *l* and *n* < *m*, call *n* of l.Unlock() happens before call *m* of l.Lock() returns.
- Mutex can be created as part of the structure

Atomic

An atomic operation is an operation that appears to the rest of the system to occur instantaneously. In other words no other go routines can see intermediate state of an atomic operation.

- Store, Add, Load, Swap, CAS
- atomic.Value

```
// https://golang.org/src/sync/atomic/asm_amd64.s

// func StoreUint32(addr *uint32, val uint32)
TEXT ·StoreUint32(SB),NOSPLIT,$0-12
    MOVQ    addr+0(FP), BP
    MOVL    val+8(FP), AX
    XCHGL    AX, 0(BP)
    RET

// func LoadUint32(addr *uint32) (val uint32)
TEXT ·LoadUint32(SB),NOSPLIT,$0-12
    MOVQ    addr+0(FP), AX
    MOVL    0(AX), AX
    MOVL    AX, val+8(FP)
    RET
```

Race detector

- A data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write.
- The race detector is integrated with the go tool chain. When the -race command-line flag is set, the compiler instruments all memory accesses with code that records when and how the memory was accessed, while the runtime library watches for unsynchronized accesses to shared variables.
- It can catch races only when they are really triggered. Make sense to run on realistic load.
- Based on Google's C/C++ ThreadSanitizer runtime library.

Sync example

What else

- CSP
- sync.Pool, sync.Once()
- Dead-locks
- Concurrency patterns
- Concurrent GC
- Lock-free data structures

Where to Go

- Concurrency is not parallelism (https://talks.golang.org/2012/waza.slide)
- The Free Lunch Is Over (http://www.gotw.ca/publications/concurrency-ddj.htm)
- dave.cheney.net/2015/08/08/performance-without-the-event-loop (http://dave.cheney.net/2015/08/08/performance-without-the-event-loop)
- Go Scheduler Design Doc (https://golang.org/s/go11sched)
- Go memory model (https://golang.org/ref/mem)
- Go Concurrency Patterns (https://talks.golang.org/2012/concurrency.slide)
- Advanced Go Concurrency Patterns (https://blog.golang.org/advanced-go-concurrency-patterns)
- github.com/golang/go/wiki/MutexOrChannel (https://github.com/golang/go/wiki/MutexOrChannel)
- ThreadSanitizer runtime library (https://github.com/google/sanitizers)
- Data Race Detector (https://golang.org/doc/articles/race_detector.html)
- Lock-free Algorithms (http://www.1024cores.net/home/lock-free-algorithms)

- Herb Sutter's talks on lock-free (https://www.youtube.com/watch?v=c1gO9aB9nbs)
- Runtime sources (https://golang.org/src/runtime/)

Thank you

Andrew Pogrebnoy
Technical Lead, Admobitec Ltd.
andrew@admobitec.com (mailto:andrew@admobitec.com)
https://github.com/absourdnoise (https://github.com/absourdnoise)