Netcracker
An NEC Company

Go Runtime in depth.

Billing&Rating DevTalks

1. Go Runtime
2. Go Assembler
3. Memory Management
4. Bootstrap Process
5. Scheduler

C App vs Go App

C Application

```c
#include <stdio.h>


int main() {
    printf("Hello Cloud Billing\n");
    return 0;
}
```

C Application

```
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/1_c_app|
⇒  gcc ./hello.c -o hello
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/1_c_app|
⇒  ./hello
Hello Cloud Billing
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/1_c_app|
⇒  ls -lh
total 40
-rwxr-xr-x  1 dgodyna  staff    12K 16 мар 15:09 hello
-rwxr-xr-x  1 dgodyna  staff    87B 16 мар 15:09 hello.c
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/1_c_app|
⇒  objdump -d ./hello | wc -l
      44
```

Go Application

```go
package main

func main() {
    println( args...: "Hello Cloud Billing")
}
```

Go Application

```
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/1_go_app|
⇒  go build hello.go
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/1_go_app|
⇒  ./hello
Hello Cloud Billing
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/1_go_app|
⇒  ls -lh
total 2352
-rwxr-xr-x  1 dgodyna  staff   1,1M 16 мар 15:25 hello
-rw-r--r--  1 dgodyna  staff    62B 16 мар 15:21 hello.go
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/1_go_app|
⇒  objdump -d ./hello | wc -l
   97255
```

Go Runtime

- Runtime is a part of every Go program
- Runtime implements following:
  - Garbage Collection
  - Concurrency
  - Stack management
  - Memory allocations
- Runtime includes:
  - Debug information
  - Reflection
  - Standard library

Agenda

Today

- Go Assembler
- Go Memory Management
- Perfomance of different types of functions
- Scheduler

On Next Part

- Compiler and linker
- Pragmas
- CGo
- TLS
- FFI

Assembler

- Assembler allows to operate with PC on the base level
- Multiple realizations

Apollo 11 Guidance Computer

```
# TO ENTER A JOB REQUEST REQUIRING NO VAC AREA:

        COUNT   02/EXEC

NOVAC   INHINT
        AD      FAKEPRET        # LOC(MPAC +6) - LOC(QPRET)
        TS      NEWPRIO         # PRIORITY OF NEW JOB + NOVAC C(FIXLOC)

        EXTEND
        INDEX   Q               # Q WILL BE UNDISTURBED THROUGHOUT.
        DCA     0               # 2CADR OF JOB ENTERED.
        DXCH    NEWLOC
        CAF     EXECBANK
        XCH     FBANK
        TS      EXECTEM1
        TCF     NOVAC2          # ENTER EXECUTIVE BANK.
```

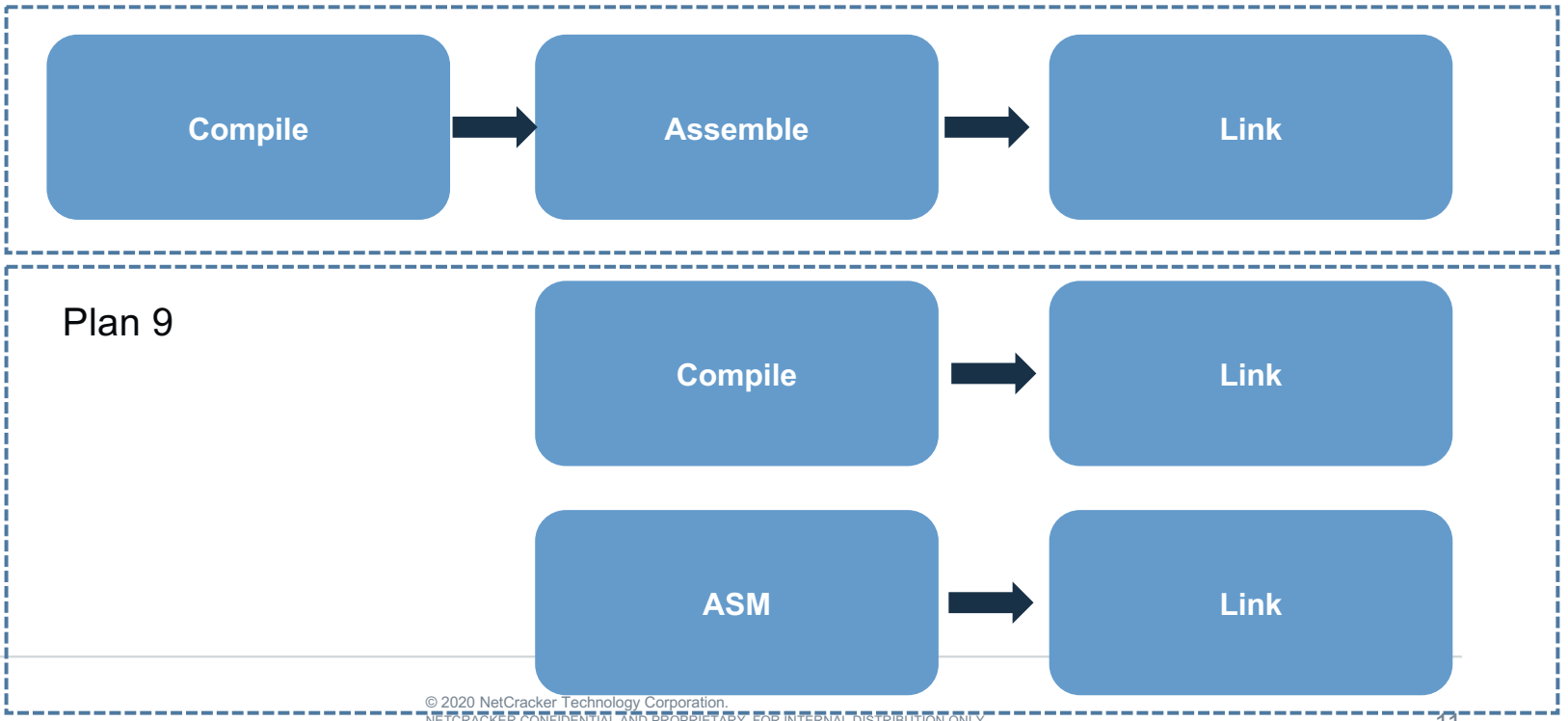
Motorola 68000

```
strtolower      public
                link    a6,#0           ;Set up stack frame
                movea   8(a6),a0        ;A0 = src, from stack
                movea   12(a6),a1       ;A1 = dst, from stack
loop            move.b  (a0)+,d0        ;Load D0 from (src)
                cmpi    #'A',d0         ;If D0 < 'A',
                blo     copy            ;skip
                cmpi    #'Z',d0         ;If D0 > 'Z',
                bhi     copy            ;skip
                addi    #'a'-'A',d0     ;D0 = lowercase(D0)
copy            move.b  d0,(a1)+        ;Store D0 to (dst)
                bne     loop            ;Repeat while D0 <> NUL
                unlk    a6              ;Restore stack frame
                rts                     ;Return
                end
```

Plan 9 Assembler

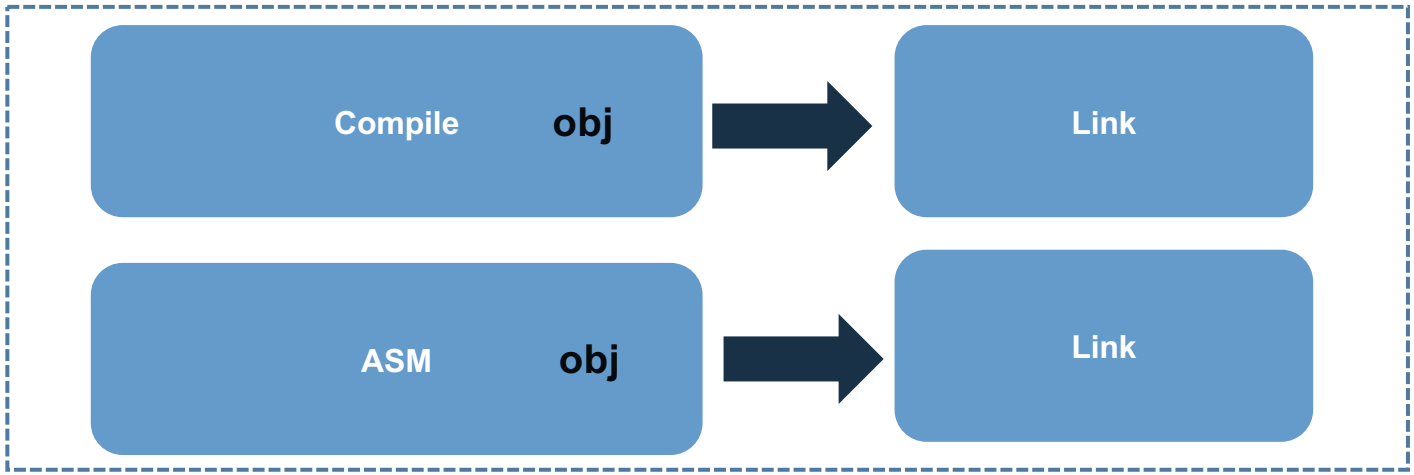
Ken Thompson

- Assembler as an intermediate representation of a program.
- Compiler will generate Plan 9 ASM code
- Linker will link Plan 9 ASM code to machine code



Go Assembler

- The assembler program is a way to parse a description of that semi-abstract instruction set and turn it into instructions to be input to the linker.
- Example GOOS=darwin GOARCH=arm



Go Assembler

- High-level (for an assembly language!)
- Architecture independent mnemonics
  - such as CALL and RET
- Instructions may be expanded by the assembler
- Assembler may insert prologues, epilogies, optimize away ‘unreachable’ instructions

Used In:

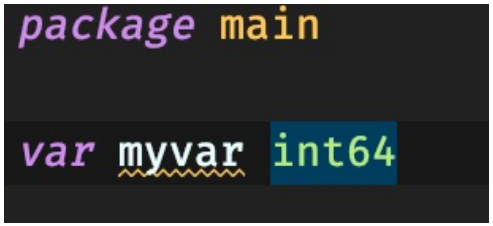
- crypto
- math
- reflect
- runtime
- sync
- syscall

Move instructions

	386	amd64	arm	arm64	mips64	ppc64	s390x
1-byte	MOVB	MOVB	MOVB	-	-	-	-
1-byte sign extend	MOVB LSX	MOVBQSX	MOVBS	MOVB	MOVB	MOVB	MOVB
1-byte zero extend	MOVB LZX	MOVBQZX	MOVBU	MOVBU	MOVBU	MOVBU	MOVBU
2-byte	MOV W	MOVW	MOVH	-	-	-	-
2-byte sign extend	MOV WLSX	MOVWQSX	MOVHS	MOVH	MOVH	MOVH	MOVH
2-byte zero extend	MOV WLZX	MOVWQZX	MOVHU	MOVHU	MOVHU	MOVHZ	MOVHZ
4-byte	MOVL	MOVL	MOVW	-	-	-	-
4-byte sign extend	-	MOVLQSX	-	MOVW	MOVW	MOVW	MOVW
4-byte zero extend	-	MOVLQZX	-	MOVWU	MOVWU	MOVWZ	MOVWZ
8-byte	-	MOVQ	-	MOVD	MOVV	MOVD	MOVD

Instructions

- Data moves from left to right
  - ADD R1, R2 // R2 += R1
  - SUB R3, R4, R5 // R5 = R4 - R3
  - MUL $7, R6 // R6 *= 7
- Memory operands: offset + reg1 + reg2*scale
  - MOV (R1), R2 // R2 = *R1
  - MOV 8(R3), R4 // R4 = *(8 + R3)
  - MOV 16(R5)(R6*1), R7 // R7 = *(16 + R5 + R6*1)
  - MOV ·myvar(SB), R8 // R8 = *myvar
- Addresses
  - MOV $8(R1)(R2*1), R3 // R3 = 8 + R1 + R2
  - MOV $·myvar(SB), R4 // R4 = &myvar



Branches

Jump to labels.

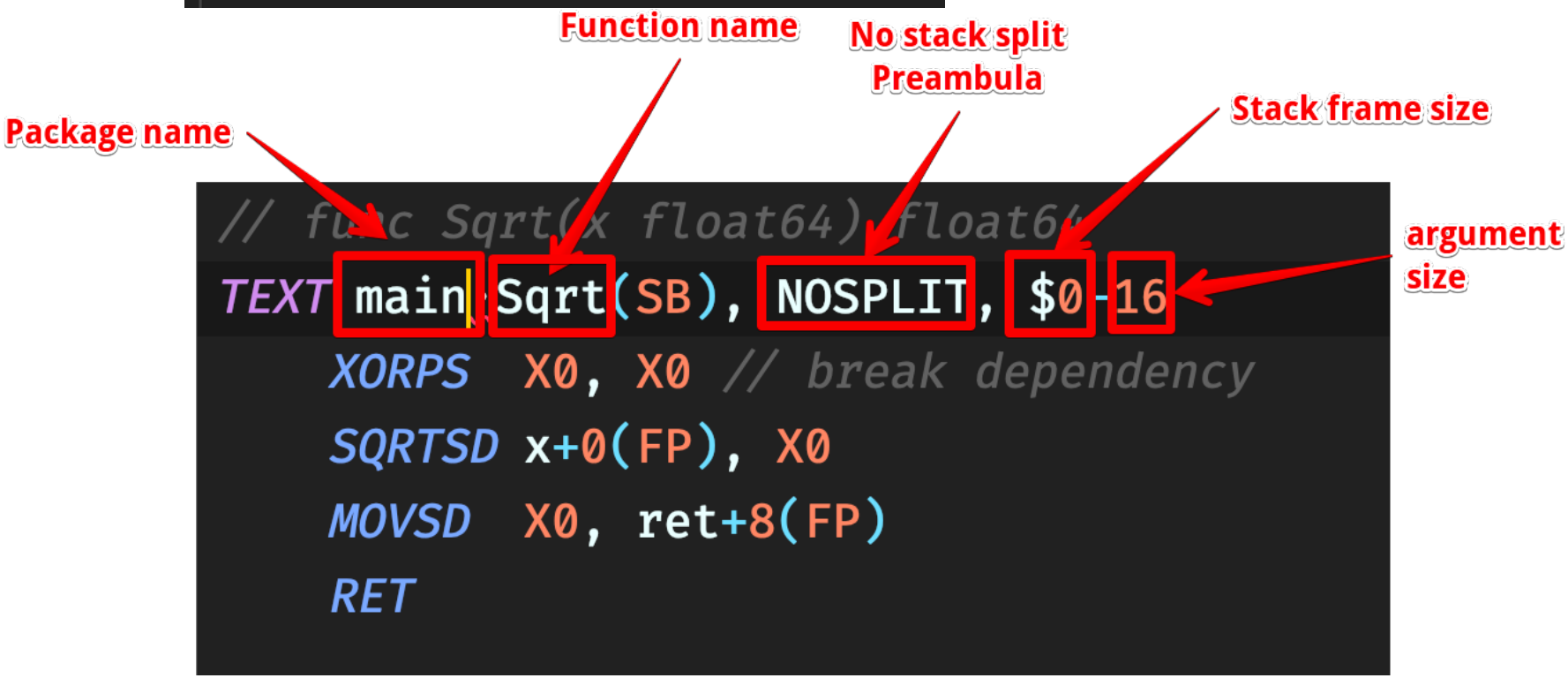
```
    JMP l1
    NOP
l1:
    NOP

l2:
    NOP
    NOP
    JMP l2
```

Jump relative to current position

```
JMP 2(PC)
NOP
NOP

NOP
NOP
JMP -2(PC)
```

Function declaration

```
func Sqrt(x float64) float64
```



Pseudo-registers

- There are four predeclared symbols that refer to pseudo-registers. These are not real registers, but rather virtual registers maintained by the toolchain, such as a frame pointer. The set of pseudo-registers is the same for all architectures:
- FP: Frame pointer: arguments and locals.
  - Points to the **bottom** of the argument list
  - Offsets are **positive**
  - Offsets must include a name, e.g. arg+0(FP)
- SP: Stack Pointer - top of stack
  - Points to the **top** of the space allocated for local variables
  - Offsets are **negative**
  - Offsets must include a name, e.g. ptr-8(SP)
  - **Positive offsets refer to hardware register on 386/amd64!**

Pseudo-registers

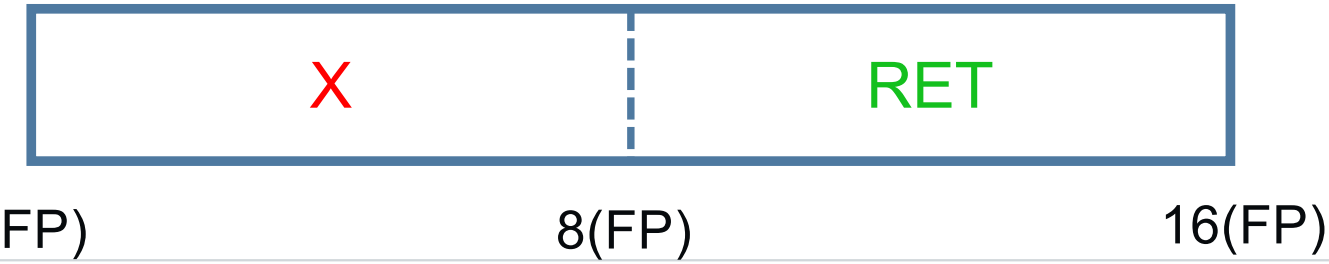
- SB: Static Base - global symbols.
  - Named offsets from a global base
- PC: Program Counter - jumps and branches
  - Used for branches
  - Offsets in **number of pseudo-instructions**

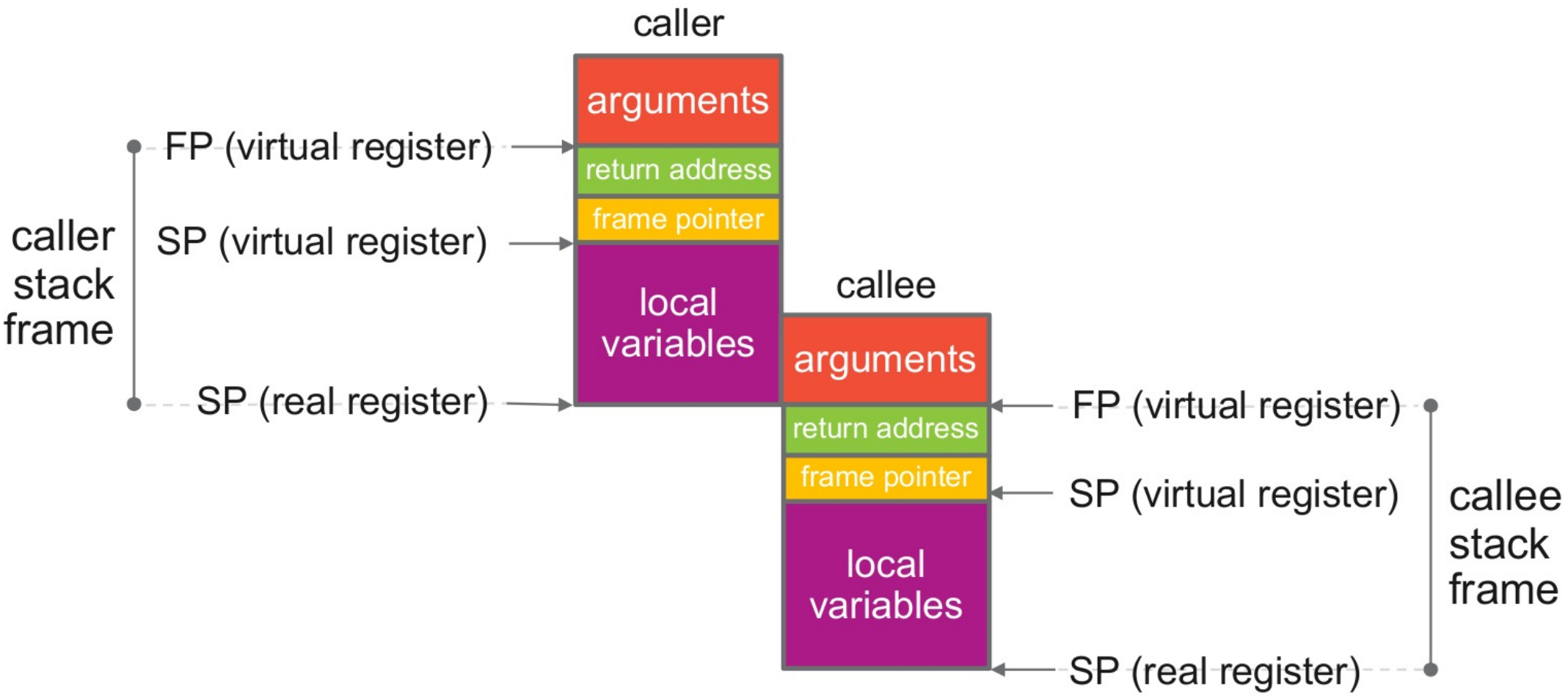
Calling convention

- All arguments passed on the stack
  - Offsets from FP
- Return arguments follow input arguments
  - Start of return arguments aligned to pointer size
- All registers are caller saved, except:
  - Stack pointer register
  - Zero register (if there is one)
  - G context pointer register (if there is one)
  - Frame pointer (if there is one)

Function arguments

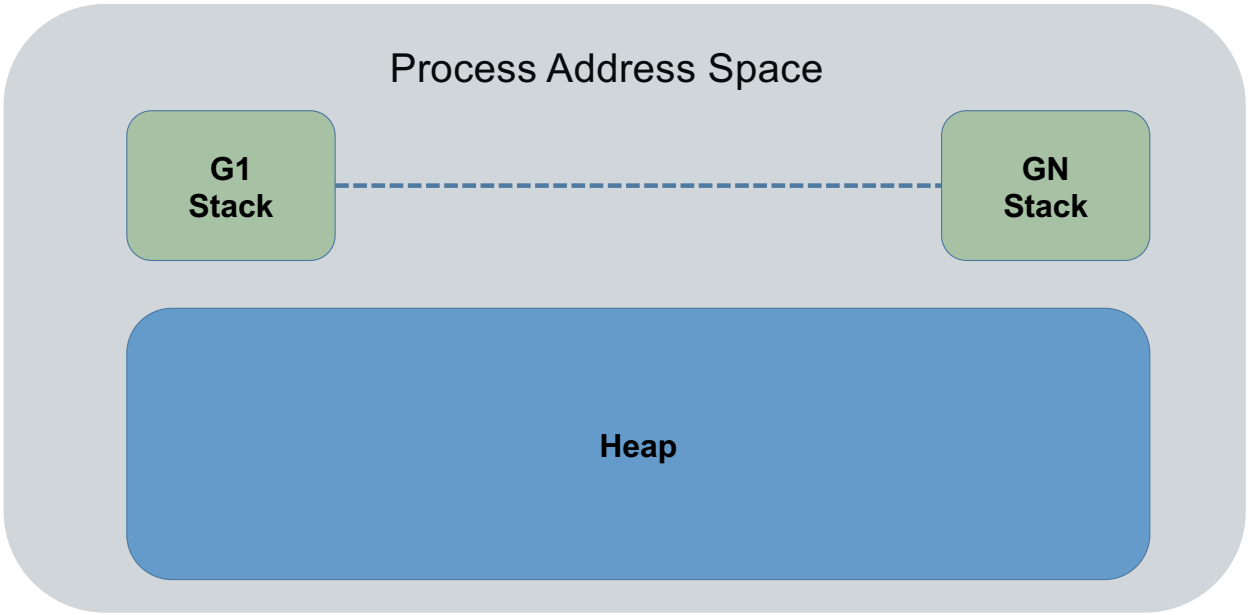
```
// func Sqrt(x float64) float64
TEXT main·Sqrt(SB), NOSPLIT, $0-16
    XORPS  X0, X0 // break dependency
    SQRTSD x+0(FP), X0
    MOVSD  X0, ret+8(FP)
    RET
```





The Go stack and heap

- Threads managed by the OS. Go use goroutines.
- Goroutines exist within user space - runtime, and not the OS, sets the rules of how stacks behave.



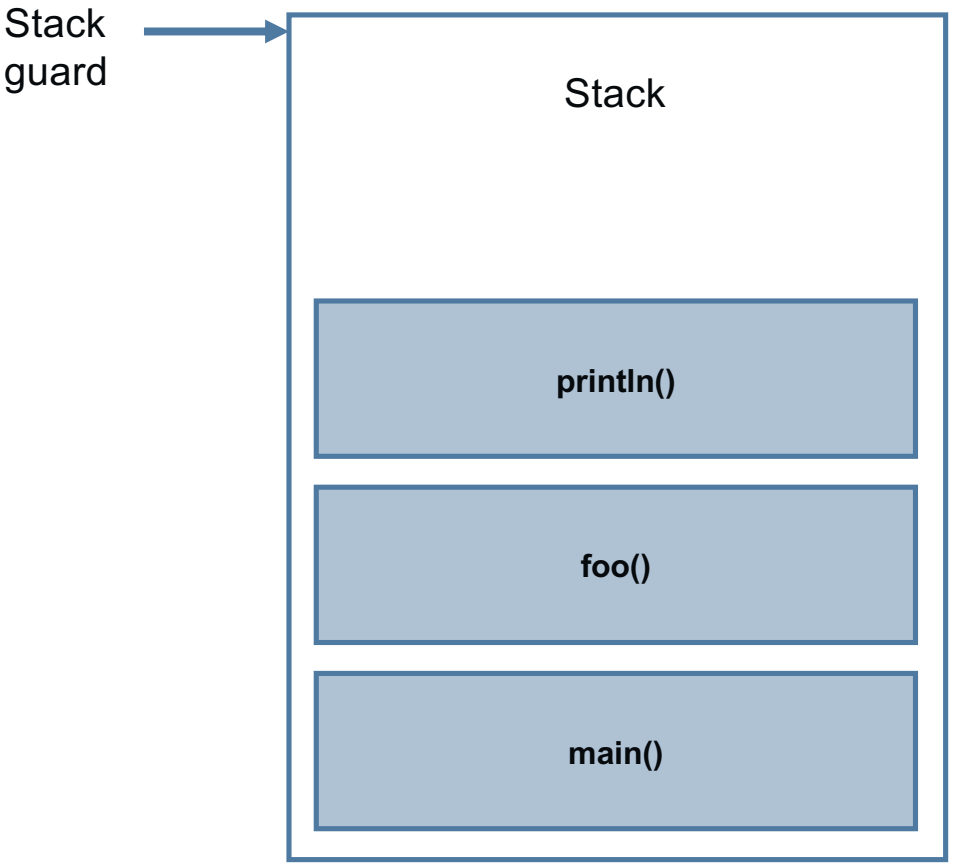
Goroutine Stack

- Initial size – 2KB (1-8 MB for POSIX threads)
- Linker preamble runtime·morestack to allocate more stack
- *stack splitting*

```
type g struct {
    // Stack parameters.
    // stack describes the actual stack memory: [stack.lo, stack.hi).
    // stackguard0 is the stack pointer compared in the Go stack growth prologue.
    // It is stack.lo+StackGuard normally, but can be StackPreempt to trigger a preemption.
    // stackguard1 is the stack pointer compared in the C stack growth prologue.
    // It is stack.lo+StackGuard on g0 and gsignal stacks.
    // It is ~0 on other goroutine stacks, to trigger a call to morestackc (and crash).
    stack       stack   // offset known to runtime/cgo
    stackguard0 uintptr // offset known to liblink
    stackguard1 uintptr // offset known to liblink
```

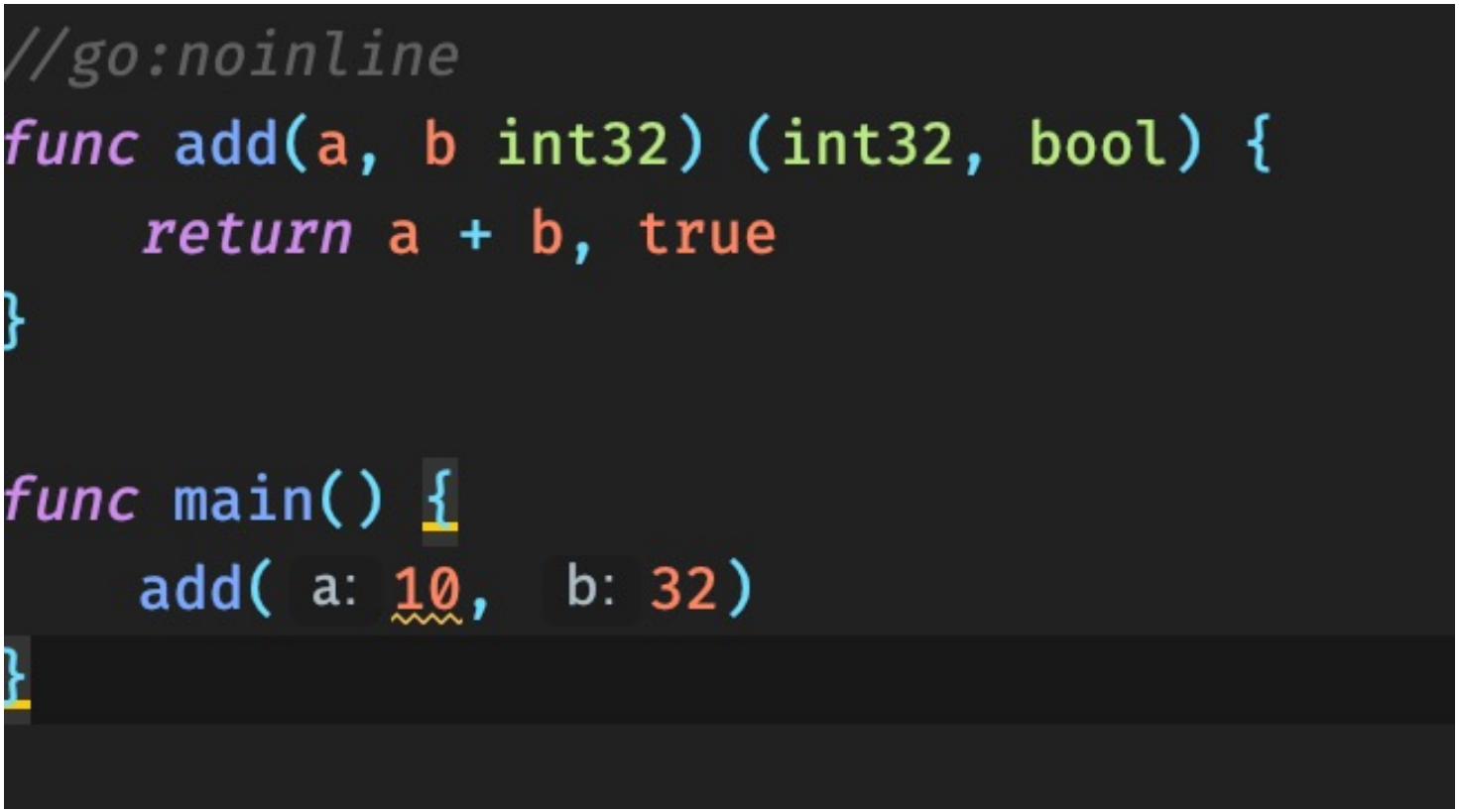
```
// Stack describes a Go execution stack.
// The bounds of the stack are exactly [lo, hi),
// with no implicit data structures on either side.
type stack struct {
    lo uintptr
    hi uintptr
}
```

```go
func main() {
    foo()
}

func foo() {
    println( args...: "Hello Cloud Billing")
}
```



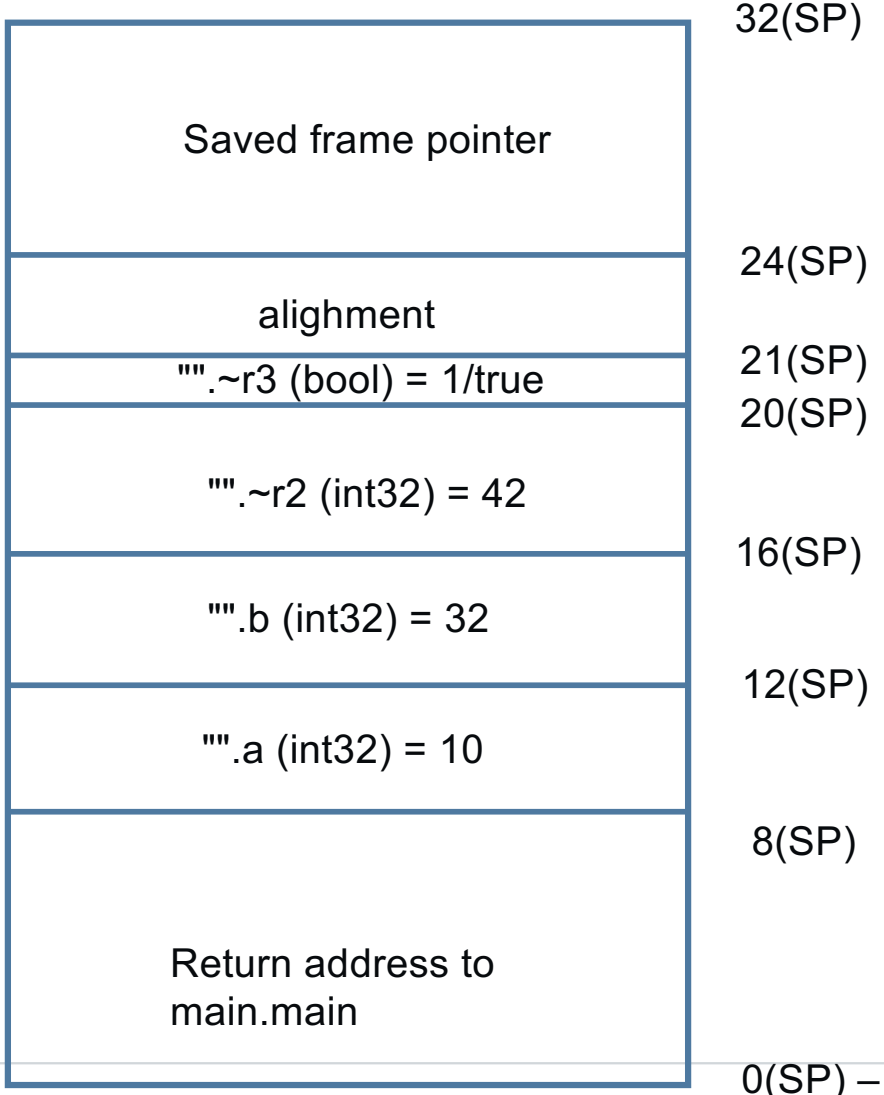
```
"".add2 STEXT size=89 args=0x18 locals=0x18 funcid=0x0
    0x0000 00000 (main.go:11)   TEXT    "".add2(SB), ABIInternal, $24-24
    0x0000 00000 (main.go:11)   MOVQ    (TLS), CX
    0x0009 00009 (main.go:11)   CMPQ    SP, 16(CX)
    0x000d 00013 (main.go:11)   PCDATA  $0, $-2
    0x000d 00013 (main.go:11)   JLS 82
    0x000f 00015 (main.go:11)   PCDATA  $0, $-1
    0x000f 00015 (main.go:11)   SUBQ    $24, SP
    0x0013 00019 (main.go:11)   MOVQ    BP, 16(SP)
    0x0018 00024 (main.go:11)   LEAQ    16(SP), BP
    0x001d 00029 (main.go:11)   FUNCDATA    $0, gclocals·f207267fbf96a0178e8758c6e3e0ce28(SB)
    0x001d 00029 (main.go:11)   FUNCDATA    $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
    0x001d 00029 (main.go:12)   LEAQ    type.int32(SB), AX
    0x0024 00036 (main.go:12)   MOVQ    AX, (SP)
    0x0028 00040 (main.go:12)   PCDATA  $1, $0
    0x0028 00040 (main.go:12)   CALL    runtime.newobject(SB)
    0x002d 00045 (main.go:12)   MOVQ    8(SP), AX
    0x0032 00050 (main.go:12)   MOVL    "".a+32(SP), CX
    0x0036 00054 (main.go:12)   ADDL    "".b(SB), CX
    0x003c 00060 (main.go:12)   MOVL    CX, (AX)
    0x003e 00062 (main.go:13)   MOVQ    AX, "".~r1+40(SP)
    0x0043 00067 (main.go:13)   MOVB    $1, "".~r2+48(SP)
    0x0048 00072 (main.go:13)   MOVQ    16(SP), BP
    0x004d 00077 (main.go:13)   ADDQ    $24, SP
    0x0051 00081 (main.go:13)   RET
    0x0052 00082 (main.go:13)   NOP
    0x0052 00082 (main.go:11)   PCDATA  $1, $-1
    0x0052 00082 (main.go:11)   PCDATA  $0, $-2
    0x0052 00082 (main.go:11)   CALL    runtime.morestack_noctxt(SB)
    0x0057 00087 (main.go:11)   PCDATA  $0, $-1
```

Simple Program



func add(a, b int32) (int32, bool)

```
"".add STEXT nosplit size=20 args=0x10 locals=0x0 funcid=0x0
    0x0000 00000 (main.go:4)    TEXT    "".add(SB), NOSPLIT|ABIInternal, $0-16
    0x0000 00000 (main.go:4)    FUNCDATA    $0, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
    0x0000 00000 (main.go:4)    FUNCDATA    $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
    0x0000 00000 (main.go:5)    MOVL    "".b+12(SP), AX
    0x0004 00004 (main.go:5)    MOVL    "".a+8(SP), CX
    0x0008 00008 (main.go:5)    ADDL    CX, AX
    0x000a 00010 (main.go:5)    MOVL    AX, "".~r2+16(SP)
    0x000e 00014 (main.go:5)    MOVB    $1, "".~r3+20(SP)
    0x0013 00019 (main.go:5)    RET
    0x0000 8b 44 24 0c 8b 4c 24 08 01 c8 89 44 24 10 c6 44  .D$..L$....D$..D
    0x0010 24 14 01 c3                                      $...
```



func main() { add(10, 14) }

```
"".main STEXT size=66 args=0×0 locals=0×18 funcid=0×0
    0×0000 00000 (main.go:8)    TEXT    "".main(SB), ABIInternal, $24-0
    0×0000 00000 (main.go:8)    MOVQ    (TLS), CX
    0×0009 00009 (main.go:8)    CMPQ    SP, 16(CX)
    0×000d 00013 (main.go:8)    PCDATA  $0, $-2
    0×000d 00013 (main.go:8)    JLS 58
    0×000f 00015 (main.go:8)    PCDATA  $0, $-1
    0×000f 00015 (main.go:8)    SUBQ    $24, SP
    0×0013 00019 (main.go:8)    MOVQ    BP, 16(SP)
    0×0018 00024 (main.go:8)    LEAQ    16(SP), BP
    0×001d 00029 (main.go:8)    FUNCDATA    $0, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
    0×001d 00029 (main.go:8)    FUNCDATA    $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
    0×001d 00029 (main.go:8)    MOVQ    $60129542154, AX
    0×0027 00039 (main.go:8)    MOVQ    AX, (SP)
    0×002b 00043 (main.go:8)    PCDATA  $1, $0
    0×002b 00043 (main.go:8)    CALL    "".add(SB)
    0×0030 00048 (main.go:8)    MOVQ    16(SP), BP
    0×0035 00053 (main.go:8)    ADDQ    $24, SP
    0×0039 00057 (main.go:8)    RET
    0×003a 00058 (main.go:8)    NOP
    0×003a 00058 (main.go:8)    PCDATA  $1, $-1
    0×003a 00058 (main.go:8)    PCDATA  $0, $-2
    0×003a 00058 (main.go:8)    CALL    runtime.morestack_noctxt(SB)
    0×003f 00063 (main.go:8)    PCDATA  $0, $-1
    0×003f 00063 (main.go:8)    NOP
    0×0040 00064 (main.go:8)    JMP 0
```

Prologue

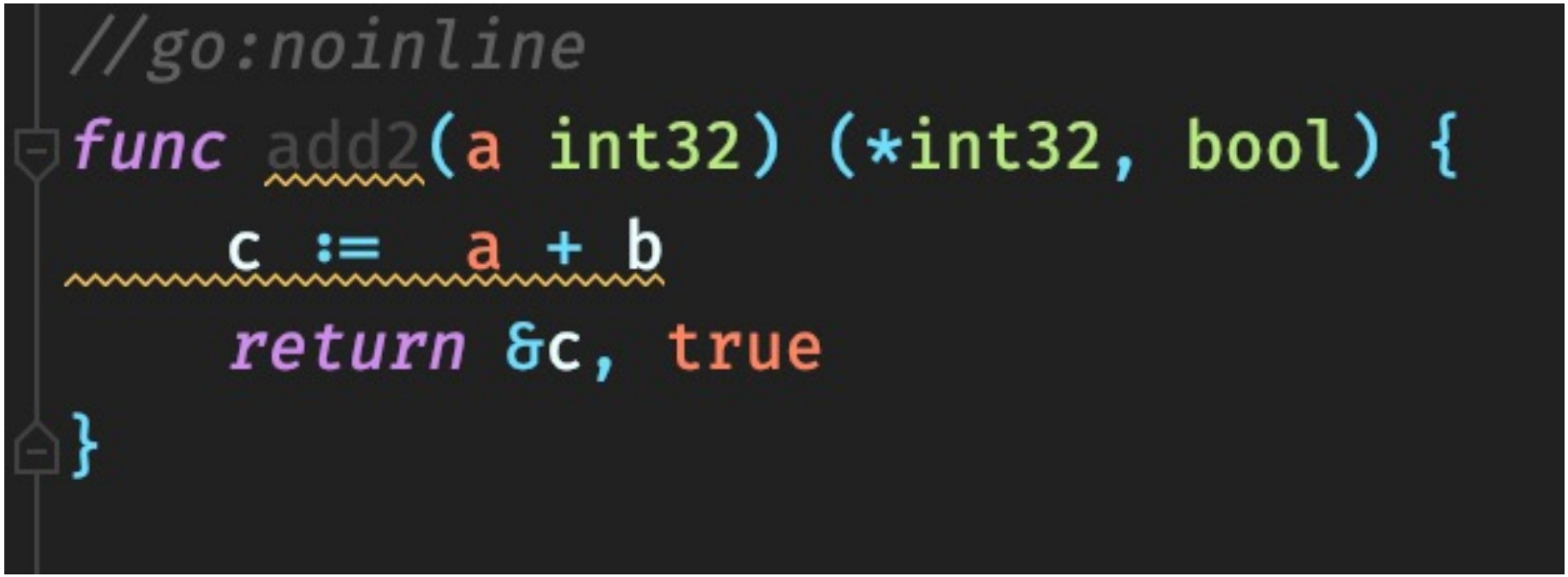
Epilogue

Using variables

```
var b = int32(14)

//go:noinline
func add(a int32) (int32, bool) {
    return a + b, true
}

func main() { add( a: 10) }
```

func add(a int32) (int32, bool)

```
add STEXT nosplit size=20 args=0x10 locals=0x0 funcid=0x0
  0x0000 00000 (main.go:6)    TEXT    "".add(SB), NOSPLIT|ABIInternal, $0-16
  0x0000 00000 (main.go:6)    FUNCDATA    $0, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
  0x0000 00000 (main.go:6)    FUNCDATA    $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
  0x0000 00000 (main.go:7)    MOVL    "".a+8(SP), AX
  0x0004 00004 (main.go:7)    ADDL    "".b(SB), AX
  0x000a 00010 (main.go:7)    MOVL    AX, "".~r1+16(SP)
  0x000e 00014 (main.go:7)    MOVB    $1, "".~r2+20(SP)
  0x0013 00019 (main.go:7)    RET
```



```
"".add2 STEXT size=89 args=0x18 locals=0x18 funcid=0x0
    0x0000 00000 (main.go:11)   TEXT    "".add2(SB), ABIInternal, $24-24
    0x0000 00000 (main.go:11)   MOVQ    (TLS), CX
    0x0009 00009 (main.go:11)   CMPQ    SP, 16(CX)
    0x000d 00013 (main.go:11)   PCDATA  $0, $-2
    0x000d 00013 (main.go:11)   JLS 82
    0x000f 00015 (main.go:11)   PCDATA  $0, $-1
    0x000f 00015 (main.go:11)   SUBQ    $24, SP
    0x0013 00019 (main.go:11)   MOVQ    BP, 16(SP)
    0x0018 00024 (main.go:11)   LEAQ    16(SP), BP
    0x001d 00029 (main.go:11)   FUNCDATA    $0, gclocals·f207267fbf96a0178e8758c6e3e0ce28(SB)
    0x001d 00029 (main.go:11)   FUNCDATA    $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
    0x001d 00029 (main.go:12)   LEAQ    type.int32(SB), AX
    0x0024 00036 (main.go:12)   MOVQ    AX, (SP)
    0x0028 00040 (main.go:12)   PCDATA  $1, $0
    0x0028 00040 (main.go:12)   CALL    runtime.newobject(SB)
    0x002d 00045 (main.go:12)   MOVQ    8(SP), AX
    0x0032 00050 (main.go:12)   MOVL    "".a+32(SP), CX
    0x0036 00054 (main.go:12)   ADDL    "".b(SB), CX
    0x003c 00060 (main.go:12)   MOVL    CX, (AX)
    0x003e 00062 (main.go:13)   MOVQ    AX, "".~r1+40(SP)
    0x0043 00067 (main.go:13)   MOVB    $1, "".~r2+48(SP)
    0x0048 00072 (main.go:13)   MOVQ    16(SP), BP
    0x004d 00077 (main.go:13)   ADDQ    $24, SP
    0x0051 00081 (main.go:13)   RET
    0x0052 00082 (main.go:13)   NOP
    0x0052 00082 (main.go:11)   PCDATA  $1, $-1
    0x0052 00082 (main.go:11)   PCDATA  $0, $-2
    0x0052 00082 (main.go:11)   CALL    runtime.morestack_noctxt(SB)
    0x0057 00087 (main.go:11)   PCDATA  $0, $-1
    0x0057 00087 (main.go:11)   JMP 0
```

Calling by reference and by value

```go
type Account struct {
    name    string
    address string
    details string
    id      int64
    zipCode int32
}
```

```go
func GetByRef() {
    _ = ByRef( name: "test name")
}

func ByRef(name string) *Account {
    acc := Account{
        name:    name,
        address: "some account address here",
        details: "some account details here",
        id:      12312312,
        zipCode: 143010,
    }

    return &acc
}
func BenchmarkByRef(b *testing.B) {
    for i := 0; i < b.N; i++ {
        GetByRef()
    }
}
```

```go
func GetByValue() {
    _ = ByValue( name: "test name")
}

func ByValue(name string) Account {
    return Account{
        name:    name,
        address: "some account address here",
        details: "some account details here",
        id:      12312312,
        zipCode: 143010,
    }
}
func BenchmarkByValue(b *testing.B) {
    for i := 0; i < b.N; i++ {
        GetByValue()
    }
}
```

Benchmark

```
⇒  go test -v -bench=. -benchmem
goos: darwin
goarch: amd64
pkg: TeckTalk3/cmd/ref_value
cpu: Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz
BenchmarkByRef
BenchmarkByRef-12       1000000000               0.2364 ns/op          0 B/op          0 allocs/op
BenchmarkByValue
BenchmarkByValue-12     1000000000               0.2329 ns/op          0 B/op          0 allocs/op
PASS
ok      TeckTalk3/cmd/ref_value 0.845s
```

Adding noinline

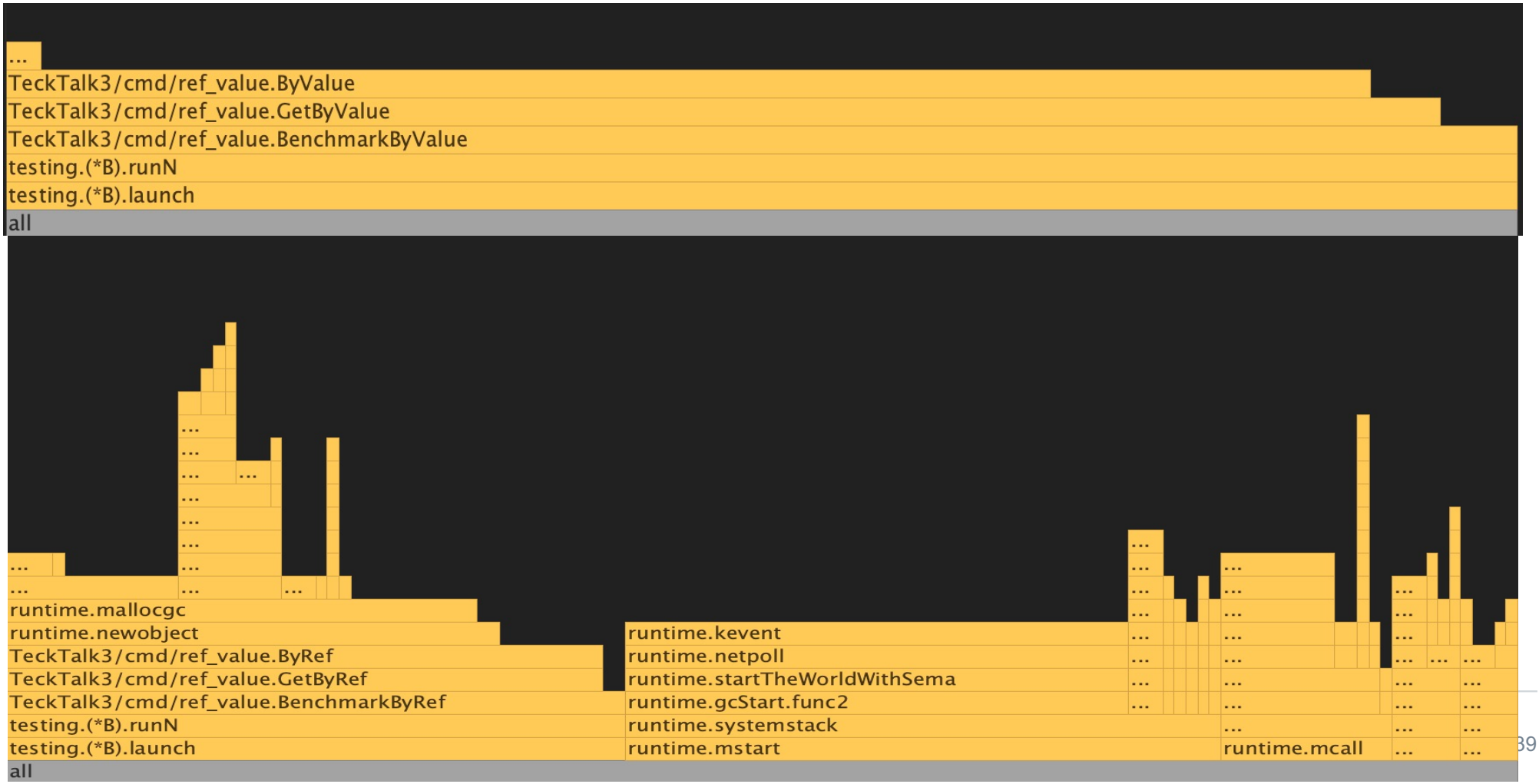
```go
//go:noinline
func ByRef(name string) *Account {
    acc := Account{
        name:    name,
        address: "some account address here",
        details: "some account details here",
        id:      12312312,
        zipCode: 143010,
    }

    return &acc
}
```

```go
//go:noinline
func ByValue(name string) Account {
    return Account{
        name:    name,
        address: "some account address here",
        details: "some account details here",
        id:      12312312,
        zipCode: 143010,
    }
}
```

Benchmark

```
⇒  go test -v -bench=. -benchmem
goos: darwin
goarch: amd64
pkg: TeckTalk3/cmd/ref_value
cpu: Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz
BenchmarkByRef
BenchmarkByRef-12       26352846                39.25 ns/op           64 B/op          1 allocs/op
BenchmarkByValue
BenchmarkByValue-12     154288635                7.856 ns/op           0 B/op          0 allocs/op
PASS
ok      TeckTalk3/cmd/ref_value 3.375s
```

CPU Profile

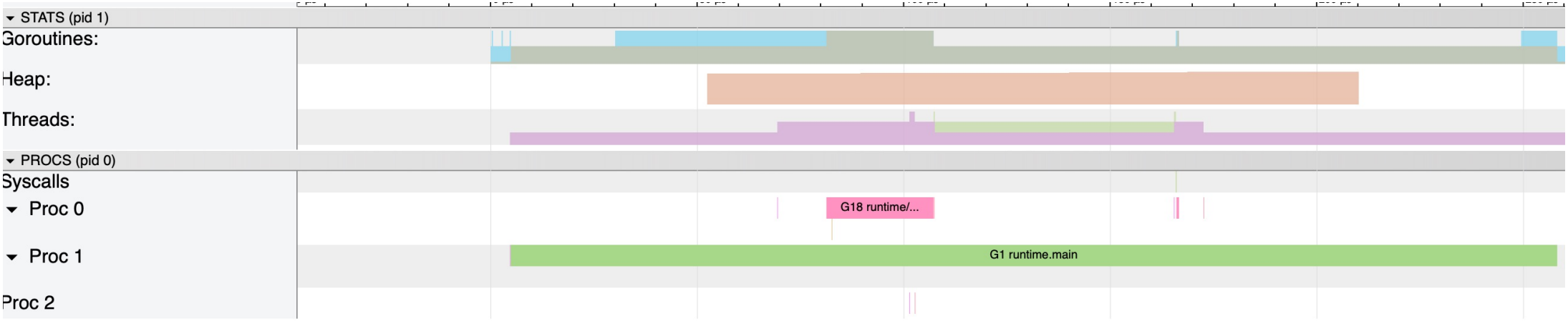

...
TeckTalk3/cmd/ref_value.ByValue
TeckTalk3/cmd/ref_value.GetByValue
TeckTalk3/cmd/ref_value.BenchmarkByValue
testing.(*B).runN
testing.(*B).launch
all
runtime.mallocgc
runtime.newobject
TeckTalk3/cmd/ref_value.ByRef
TeckTalk3/cmd/ref_value.GetByRef
TeckTalk3/cmd/ref_value.BenchmarkByRef
testing.(*B).runN
testing.(*B).launch
runtime.kevent
runtime.netpoll
runtime.startTheWorldWithSema
runtime.gcStart.func2
runtime.systemstack
runtime.mstart
runtime.mcall
all

Trace

```
var cycles = 1_000_000_000

func TestByRef(t *testing.T) {
    for i := 0; i < cycles; i++ {
        GetByRef()
    }
}

func TestByValue(t *testing.T) {
    for i := 0; i < cycles; i++ {
        GetByValue()
    }
}
```

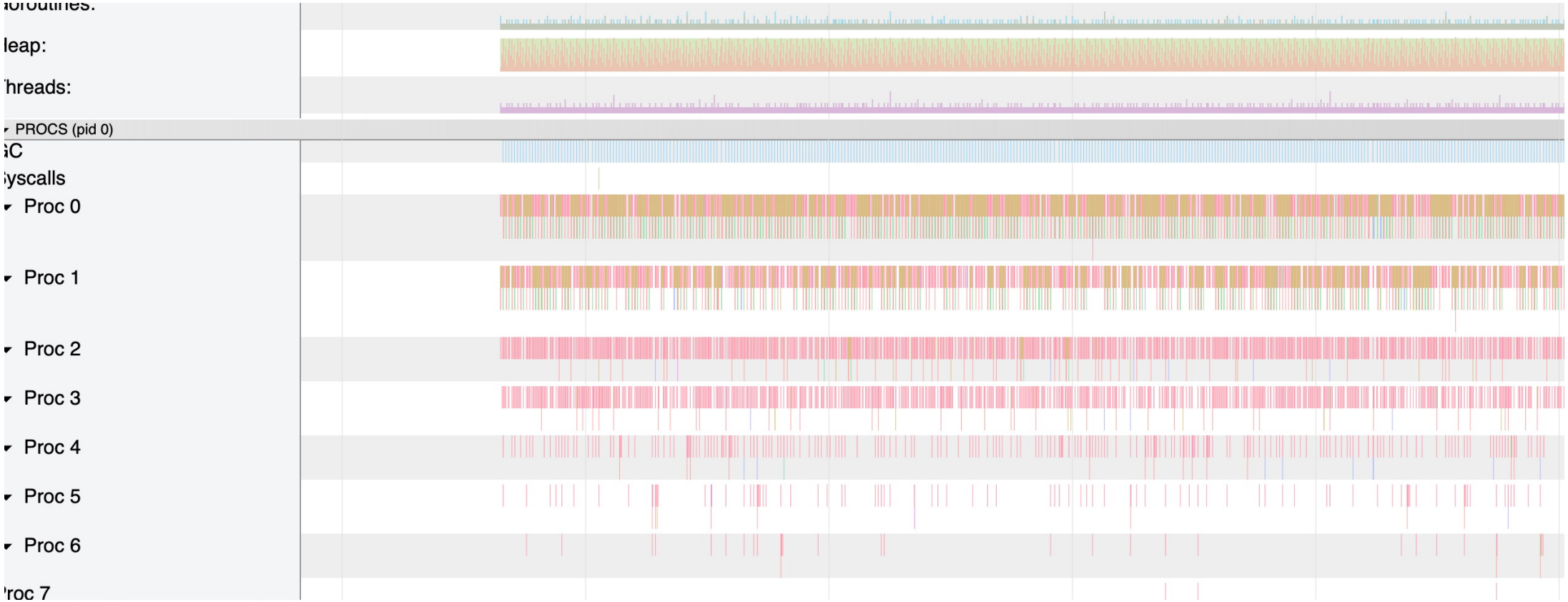
```
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/cmd/ref_value
⇒  go test -run TestByRef -trace=ref_trace.out
PASS
ok      TeckTalk3/cmd/ref_value 40.632s
dgodyna@MacBook-Pro-Dima:~/work/My/TeckTalk3/cmd/ref_value
⇒  go test -run TestByValueTestByValue -trace=value_trace.out
testing: warning: no tests to run
PASS
ok      TeckTalk3/cmd/ref_value 0.086s
```

By Value



Goroutine Name: testing.runTests.func1.1
Number of Goroutines: 1
Execution Time: 1.65% of total program execution time
Network Wait Time: graph(download)
Sync Block Time: graph(download)
Blocking Syscall Time: graph(download)
Scheduler Wait Time: graph(download)

Goroutine	Total		Execution	Network wait	Sync block	Blocking syscall	Scheduler wait	GC sweeping	GC pause
19	19μs		5420ns	0ns	0ns	0ns	14μs	0ns (0.0%)	0ns (0.0%)



Goroutine Name: testing.tRunner
Number of Goroutines: 1
Execution Time: 83.47% of total program execution time
Network Wait Time: graph(download)
Sync Block Time: graph(download)
Blocking Syscall Time: graph(download)
Scheduler Wait Time: graph(download)

Goroutine	Total		Execution	Network wait	Sync block	Blocking syscall	Scheduler wait	GC sweeping	GC pause
19	40s		38s	0ns	2562µs	221µs	1797ms	1739ms (4.3%)	4219ms (10.4%)

```go
var cycles = 1_000_000_000

func TestByRef(t *testing.T) {
    start := time.Now()
    for i := 0; i < cycles; i++ {
        GetByRef()
    }
    duration := time.Now().Sub(start)
    stats := runtime.MemStats{}
    runtime.ReadMemStats(&stats)

    fmt.Printf( format: "Duration:    %v\n", duration)
    fmt.Printf( format: "Heap Alloc:  %v\n", humanize.Bytes(stats.HeapAlloc))
    fmt.Printf( format: "Num GC:      %v\n", stats.NumGC)
    fmt.Printf( format: "STW Count:   %v\n", time.Duration(stats.PauseTotalNs))
}

func TestByValue(t *testing.T) {
    start := time.Now()
    for i := 0; i < cycles; i++ {
        GetByValue()
    }
    duration := time.Now().Sub(start)
    stats := runtime.MemStats{}
    runtime.ReadMemStats(&stats)

    fmt.Printf( format: "Duration:    %v\n", duration)
    fmt.Printf( format: "Heap Alloc:  %v\n", humanize.Bytes(stats.HeapAlloc))
    fmt.Printf( format: "Num GC:      %v\n", stats.NumGC)
    fmt.Printf( format: "STW Count:   %v\n", time.Duration(stats.PauseTotalNs))
}
```

```
⇒   go test -run TestByValue
Duration:    9.639681754s
Heap Alloc:  246 kB
Num GC:      0
STW Count:   0s
PASS
ok      TeckTalk3/cmd/ref_value 9.919s
```

```
⇒   go test -run TestByRef
Duration:    40.845879506s
Heap Alloc:  3.7 MB
Num GC:      16114
STW Count:   813.803574ms
PASS
ok      TeckTalk3/cmd/ref_value 41.132s
```

```
⇒  GOGC=10000 go test -run TestByRef
Duration:    38.487190723s
Heap Alloc:  279 MB
Num GC:      152
STW Count:   11.596115ms
PASS
ok      TeckTalk3/cmd/ref_value 38.588s
```

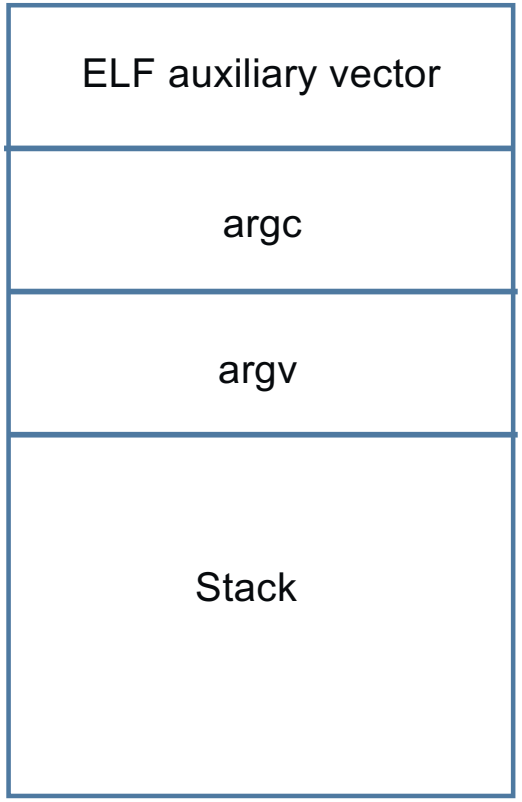
Bootstrap Process

```
[root@01484cd663b7 1_go_app]# objdump -f ./hello

./hello:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x000000000045b9e0

[root@01484cd663b7 1_go_app]# objdump -d ./hello > dump.txt
[root@01484cd663b7 1_go_app]#
```

```
000000000045b9e0 <_rt0_amd64_linux>:
  45b9e0:   e9 9b cb ff ff          jmpq   458580 <_rt0_amd64>
  45b9e5:   cc                      int3
  45b9e6:   cc                      int3
  45b9e7:   cc                      int3
  45b9e8:   cc                      int3
  45b9e9:   cc                      int3
  45b9ea:   cc                      int3
```

```
// _rt0_amd64 is common startup code for most amd64 systems when using
// internal linking. This is the entry point for the program from the
// kernel for an ordinary -buildmode=exe program. The stack holds the
// number of arguments and the C-style argv.
TEXT _rt0_amd64(SB),NOSPLIT,$-8
    MOVQ    0(SP), DI    // argc
    LEAQ    8(SP), SI    // argv
    JMP runtime·rt0_go(SB)
```

```
// Defined as ABIInternal since it does not use the stack-based Go ABI (and
// in addition there are no calls to this entry point from Go code).
TEXT runtime·rt0_go<ABIInternal>(SB),NOSPLIT,$0
    // copy arguments forward on an even stack
    MOVQ    DI, AX      // argc
    MOVQ    SI, BX      // argv
    SUBQ    $(4*8+7), SP        // 2args 2auto
    ANDQ    $~15, SP
    MOVQ    AX, 16(SP)
    MOVQ    BX, 24(SP)
```



```
// create istack out of the given (operating system) stack.
// _cgo_init may update stackguard.
MOVQ    $runtime·g0(SB), DI
LEAQ    (-64*1024+104)(SP), BX
MOVQ    BX, g_stackguard0(DI)
MOVQ    BX, g_stackguard1(DI)
MOVQ    BX, (g_stack+stack_lo)(DI)
MOVQ    SP, (g_stack+stack_hi)(DI)
```

```
ok:
    // set the per-goroutine and per-mach "registers"
    get_tls(BX)
    LEAQ    runtime·g0(SB), CX
    MOVQ    CX, g(BX)
    LEAQ    runtime·m0(SB), AX

    // save m->g0 = g0
    MOVQ    CX, m_g0(AX)
    // save m0 to g0->m
    MOVQ    AX, g_m(CX)

    CLD             // convention is D is always left cleared
    CALL    runtime·check(SB)

    MOVL    16(SP), AX      // copy argc
    MOVL    AX, 0(SP)
    MOVQ    24(SP), AX      // copy argv
    MOVQ    AX, 8(SP)
    CALL    runtime·args(SB)
    CALL    runtime·osinit(SB)
    CALL    runtime·schedinit(SB)
```

runtime.check()

```
func check() {
    var (
        a     int8
        b     uint8
        c     int16
        d     uint16
        e     int32
        f     uint32
        g     int64
        h     uint64
        i, i1 float32
        j, j1 float64
        k     unsafe.Pointer
        l     *uint16
        m     [4]byte
    )
```

```
type x1t struct {
    x uint8
}
type y1t struct {
    x1 x1t
    y  uint8
}
var x1 x1t
var y1 y1t

if unsafe.Sizeof(a) ≠ 1 {
    throw( s: "bad a")
}
if unsafe.Sizeof(b) ≠ 1 {
    throw( s: "bad b")
}
if unsafe.Sizeof(c) ≠ 2 {
    throw( s: "bad c")
}
if unsafe.Sizeof(d) ≠ 2 {
    throw( s: "bad d")
}
```

```
// create istack out of the given (operating system) stack.
// _cgo_init may update stackguard.
MOVQ    $runtime·g0(SB), DI
LEAQ    (-64*1024+104)(SP), BX
MOVQ    BX, g_stackguard0(DI)
MOVQ    BX, g_stackguard1(DI)
MOVQ    BX, (g_stack+stack_lo)(DI)
MOVQ    SP, (g_stack+stack_hi)(DI)
```

```
ok:
    // set the per-goroutine and per-mach "registers"
    get_tls(BX)
    LEAQ    runtime·g0(SB), CX
    MOVQ    CX, g(BX)
    LEAQ    runtime·m0(SB), AX

    // save m->g0 = g0
    MOVQ    CX, m_g0(AX)
    // save m0 to g0->m
    MOVQ    AX, g_m(CX)

    CLD             // convention is D is always left cleared
    CALL    runtime·check(SB)

    MOVL    16(SP), AX      // copy argc
    MOVL    AX, 0(SP)
    MOVQ    24(SP), AX      // copy argv
    MOVQ    AX, 8(SP)
    CALL    runtime·args(SB)
    CALL    runtime·osinit(SB)
    CALL    runtime·schedinit(SB)
```

runtime.args

```
func args(c int32, v **byte) {
    argc = c
    argv = v
    sysargs(c, v)
}
```

```
func sysargs(argc int32, argv **byte) {
    // skip over argv, envv and the first string will be the path
    n := argc + 1
    for argv_index(argv, n) ≠ nil {
        n++
    }
    executablePath = gostringnocopy(argv_index(argv, n+1))

    // strip "executable_path=" prefix if available, it's added after OS X 10.11.
    const prefix = "executable_path="
    if len(executablePath) > len(prefix) && executablePath[:len(prefix)] == prefix {
        executablePath = executablePath[len(prefix):]
    }
}
```

```
// create istack out of the given (operating system) stack.
// _cgo_init may update stackguard.
MOVQ    $runtime·g0(SB), DI
LEAQ    (-64*1024+104)(SP), BX
MOVQ    BX, g_stackguard0(DI)
MOVQ    BX, g_stackguard1(DI)
MOVQ    BX, (g_stack+stack_lo)(DI)
MOVQ    SP, (g_stack+stack_hi)(DI)
```

```
ok:
    // set the per-goroutine and per-mach "registers"
    get_tls(BX)
    LEAQ    runtime·g0(SB), CX
    MOVQ    CX, g(BX)
    LEAQ    runtime·m0(SB), AX

    // save m->g0 = g0
    MOVQ    CX, m_g0(AX)
    // save m0 to g0->m
    MOVQ    AX, g_m(CX)

    CLD             // convention is D is always left cleared
    CALL    runtime·check(SB)

    MOVL    16(SP), AX      // copy argc
    MOVL    AX, 0(SP)
    MOVQ    24(SP), AX      // copy argv
    MOVQ    AX, 8(SP)
    CALL    runtime·args(SB)
    CALL    runtime·osinit(SB)
    CALL    runtime·schedinit(SB)
```

runtime.osInit

```
// +build arm64
// +build linux
// +build !android

package cpu

func osInit() {
    hwcapInit("linux")
}
```

```
func osInit() {
    ARM64.HasATOMICS = sysctlEnabled([]byte("hw.optional.armv8_1_atomics\x00"))
    ARM64.HasCRC32 = sysctlEnabled([]byte("hw.optional.armv8_crc32\x00"))

    // There are no hw.optional sysctl values for the below features on Mac OS
    // to detect their supported state dynamically. Assume the CPU features th
    // Apple Silicon M1 supports to be available as a minimal set of features
    // to all Go programs running on darwin/arm64.
    ARM64.HasAES = true
    ARM64.HasPMULL = true
    ARM64.HasSHA1 = true
    ARM64.HasSHA2 = true
}
```

```
func hwcapInit(os string) {
    // HWCap was populated by the runtime from the auxiliary vector.
    // Use HW          since reading aarch64 system registers
    // is not supported in user space on older linux kernels.
    ARM64.HasAES = isSet(HWCap, hwcap_AES)
    ARM64.HasPMULL = isSet(HWCap, hwcap_PMULL)
    ARM64.HasSHA1 = isSet(HWCap, hwcap_SHA1)
    ARM64.HasSHA2 = isSet(HWCap, hwcap_SHA2)
    ARM64.HasCRC32 = isSet(HWCap, hwcap_CRC32)
    ARM64.HasCPUID = isSet(HWCap, hwcap_CPUID)

    // The Samsung S9+ kernel reports support for atomics, but not all cores
    // actually support them, resulting in SIGILL. See issue #28431.
    // TODO(elias.naur): Only disable the optimization on bad chipsets on android.
    ARM64.HasATOMICS = isSet(HWCap, hwcap_ATOMICS) && os ≠ "android"
```

type bool bool

```
// create istack out of the given (operating system) stack.
// _cgo_init may update stackguard.
MOVQ    $runtime·g0(SB), DI
LEAQ    (-64*1024+104)(SP), BX
MOVQ    BX, g_stackguard0(DI)
MOVQ    BX, g_stackguard1(DI)
MOVQ    BX, (g_stack+stack_lo)(DI)
MOVQ    SP, (g_stack+stack_hi)(DI)
```

```
ok:
    // set the per-goroutine and per-mach "registers"
    get_tls(BX)
    LEAQ    runtime·g0(SB), CX
    MOVQ    CX, g(BX)
    LEAQ    runtime·m0(SB), AX

    // save m->g0 = g0
    MOVQ    CX, m_g0(AX)
    // save m0 to g0->m
    MOVQ    AX, g_m(CX)

    CLD             // convention is D is always left cleared
    CALL    runtime·check(SB)

    MOVL    16(SP), AX      // copy argc
    MOVL    AX, 0(SP)
    MOVQ    24(SP), AX      // copy argv
    MOVQ    AX, 8(SP)
    CALL    runtime·args(SB)
    CALL    runtime·osinit(SB)
    CALL    runtime·schedinit(SB)
```

```
// The bootstrap sequence is:
//
//  call osinit
//  call schedinit
//  make & queue new G
//  call runtime·mstart
//
// The new G calls runtime·main.
func schedinit() {
```

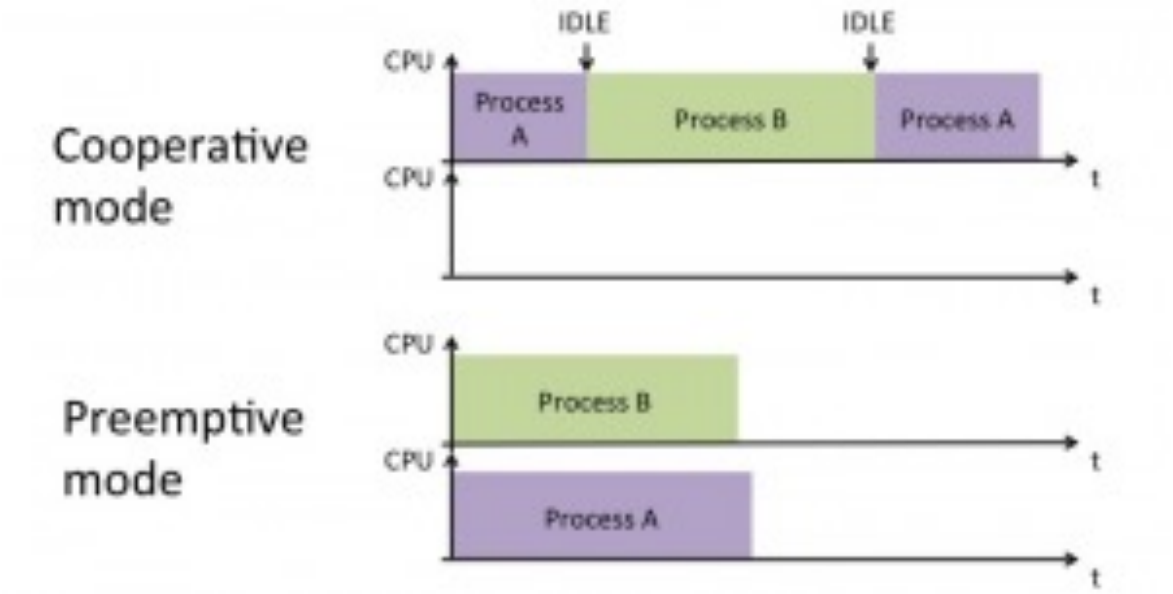
```
// raceinit must be the first call to race detector.
// In particular, it must be done before mallocinit below calls racemapshadow.
_g_ := getg()
if raceenabled {
    _g_.racectx, raceprocctx0 = raceinit()
}

// The world starts stopped.
worldStopped()

moduledataverify()
stackinit()
mallocinit()
fastrandinit() // must run before mcommoninit
mcommoninit(_g_.m,  id: -1)
cpuinit()       // must run before alginit
alginit()       // maps must not be used before this call
modulesinit()   // provides activeModules
typelinksinit() // uses maps, activeModules
itabsinit()     // uses activeModules
```

```
goargs()
goenvs()
parsedebugvars()
gcinit()

lock(&sched.lock)
sched.lastpoll = uint64(nanotime())
procs := ncpu
if n, ok := atoi32(gogetenv( key: "GOMAXPROCS")); ok && n > 0 {
    procs = n
}
if procresize(procs) != nil {
    throw( s: "unknown runnable goroutine during bootstrap")
}
unlock(&sched.lock)

// World is effectively started now, as P's can run.
worldStarted()
```

```
// create a new goroutine to start program
MOVQ    $runtime·mainPC(SB), AX     // entry
PUSHQ   AX
PUSHQ   $0          // arg size
CALL    runtime·newproc(SB)
POPQ    AX
POPQ    AX

// start this M
CALL    runtime·mstart(SB)

CALL    runtime·abort(SB)   // mstart should never return
RET
```

Multitasking

- Preemptive multitasking
- Cooperative multitasking

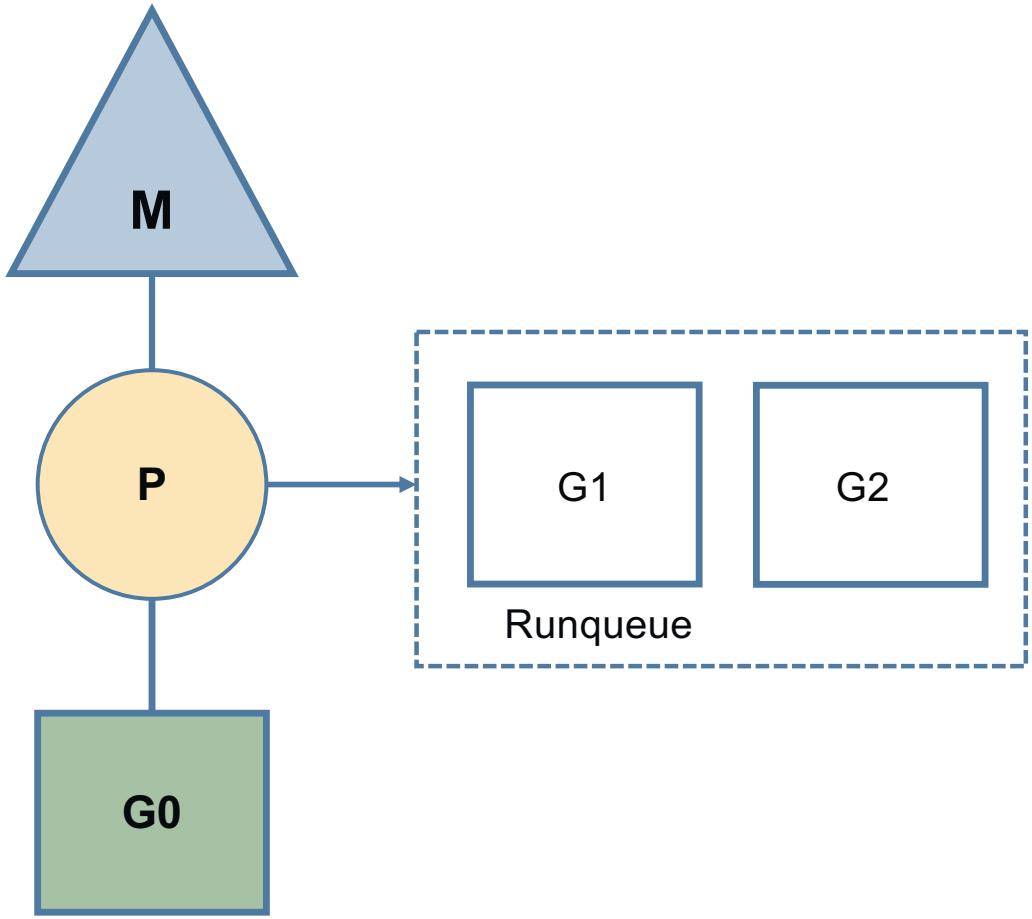


Concurrency

- Multiple OS threads
- Multiple goroutines
- User interact only with gorounites

G M P

- M (machine) – OS thread
- P (processor) – Execution Context, holds runqueue
- G – Goroutine

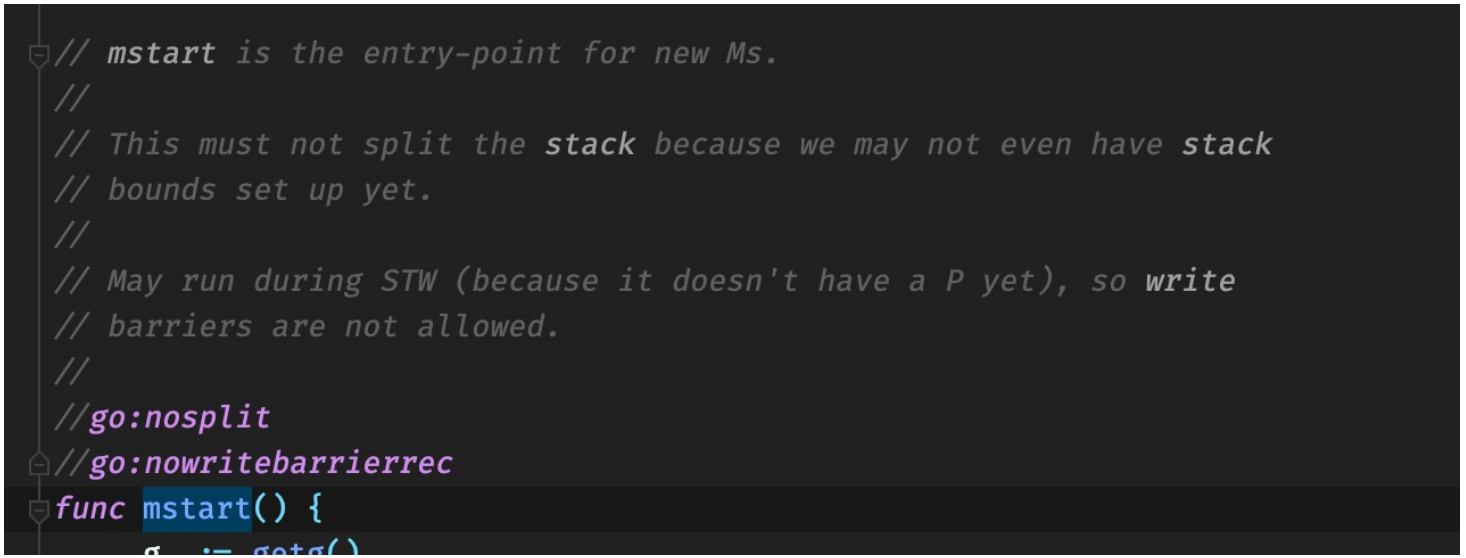


```
// create a new goroutine to start program
MOVQ    $runtime·mainPC(SB), AX     // entry
PUSHQ   AX
PUSHQ   $0          // arg size
CALL    runtime·newproc(SB)
POPQ    AX
POPQ    AX

// start this M
CALL    runtime·mstart(SB)

CALL    runtime·abort(SB)   // mstart should never return
RET
```

```go
// Create a new g running fn with siz bytes of arguments.
// Put it on the queue of g's waiting to run.
// The compiler turns a go statement into a call to this.
//
// The stack layout of this call is unusual: it assumes that the
// arguments to pass to fn are on the stack sequentially immediately
// after &fn. Hence, they are logically part of newproc's argument
// frame, even though they don't appear in its signature (and can't
// because their types differ between call sites).
//
// This must be nosplit because this stack layout means there are
// untyped arguments in newproc's argument frame. Stack copies won't
// be able to adjust them and stack splits won't be able to copy them.
//
//go:nosplit
func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    gp := getg()
    pc := getcallerpc()
    systemstack(func() {
        newg := newproc1(fn, argp, siz, gp, pc)

        _p_ := getg().m.p.ptr()
        runqput(_p_, newg, next: true)

        if mainStarted {
            wakep()
        }
    })
}
```

```go
// Create a new g in state _Grunnable, starting at fn, with narg bytes
// of arguments starting at argp. callerpc is the address of the go
// statement that created this. The caller is responsible for adding
// the new g to the scheduler.
//
// This must run on the system stack because it's the continuation of
// newproc, which cannot split the stack.
//
//go:systemstack
func newproc1(fn *funcval, argp unsafe.Pointer, narg int32, callergp *g, callerpc uintptr) *g {
```

```
// create a new goroutine to start program
MOVQ    $runtime·mainPC(SB), AX     // entry
PUSHQ   AX
PUSHQ   $0          // arg size
CALL    runtime·newproc(SB)
POPQ    AX
POPQ    AX

// start this M
CALL    runtime·mstart(SB)

CALL    runtime·abort(SB)   // mstart should never return
RET
```



```
// mstart is the entry-point for new Ms.
//
// This must not split the stack because we may not even have stack
// bounds set up yet.
//
// May run during STW (because it doesn't have a P yet), so write
// barriers are not allowed.
//
//go:nosplit
//go:nowritebarrierrec
func mstart() {
```

```
    // Record the caller for use as the top of stack in mcall and
    // for terminating the thread.
    // We're never coming back to mstart1 after we call schedule,
    // so other calls can reuse the current frame.
    save(getcallerpc(), getcallersp())
    asminit()
    minit()

    // Install signal handlers; after minit so that minit can
    // prepare the thread to be able to handle the signals.
    if _g_.m == &m0 {
        mstartm0()
    }

    if fn := _g_.m.mstartfn; fn != nil {
        fn()
    }

    if _g_.m != &m0 {
        acquirep(_g_.m.nextp.ptr())
        _g_.m.nextp = 0
    }
    schedule()
}
```

```
// One round of scheduler: find a runnable goroutine and execute it.
// Never returns.
func schedule() {
 if gp == nil && gcBlackenEnabled ≠ 0 {
     gp = gcController.findRunnableGCWorker(_g_.m.p.ptr())
     tryWakeP = tryWakeP || gp ≠ nil
 }
// findRunnableGCWorker returns a background mark worker for _p_ if it
// should be run. This must only be called when gcBlackenEnabled ≠ 0.
func (c *gcControllerState) findRunnableGCWorker(_p_ *p) *g {
 if gp == nil {
     gp, inheritTime = runqget(_g_.m.p.ptr())
     // We can see gp ≠ nil here even if the M is spinning,
     // if checkTimers added a local goroutine via goready.
 }
 if gp == nil {
     gp, inheritTime = findrunnable() // blocks until work is available
 }

// Finds a runnable goroutine to execute.
// Tries to steal from other P's, get g from local or global queue, poll network.
func findrunnable() (gp *g, inheritTime bool) {

    execute(gp, inheritTime)
```

```
// func mcall(fn func(*g))
// Switch to m->g0's stack, call fn(g).
// Fn must never return. It should gogo(&g->sched)
// to keep running g.
TEXT runtime·mcall(SB), NOSPLIT, $0-8
    MOVQ    fn+0(FP), DI

    get_tls(CX)
    MOVQ    g(CX), AX   // save state in g->sched
    MOVQ    0(SP), BX   // caller's PC
    MOVQ    BX, (g_sched+gobuf_pc)(AX)
    LEAQ    fn+0(FP), BX    // caller's SP
    MOVQ    BX, (g_sched+gobuf_sp)(AX)
    MOVQ    AX, (g_sched+gobuf_g)(AX)
    MOVQ    BP, (g_sched+gobuf_bp)(AX)

    // switch to m->g0 & its stack, call fn
    MOVQ    g(CX), BX
    MOVQ    g_m(BX), BX
    MOVQ    m_g0(BX), SI
    CMPQ    SI, AX  // if g == m->g0 call badmcall
    JNE 3(PC)
    MOVQ    $runtime·badmcall(SB), AX
    JMP AX
    MOVQ    SI, g(CX)   // g = m->g0
    MOVQ    (g_sched+gobuf_sp)(SI), SP  // sp = m->g0->sched.sp
    PUSHQ   AX
    MOVQ    DI, DX
    MOVQ    0(DI), DI
    CALL    DI
    POPQ    AX
    MOVQ    $runtime·badmcall2(SB), AX
    JMP AX
    RET
```


```
//go:linkname main_main main.main
func main_main()
```

Summary

- Go Assembler
- Calling convention
- Stack and heap, allocations
- References vs values
- Bootstrap process
- Concurrency

Q&A


Netcracker
An NEC Company