DELVE

A Debugger for the Go Programming Language

Internal Architecture

What is This

- Delve is:
  - A symbolic debugger for Go

    https://github.com/derekparker/delve

  - Used by Goland IDE, VSCode Go, vim-go (and others)
- This talk will:
  - give a general overview of delve’s architecture
  - explain why other debuggers have difficulties with Go programs

Table of Contents

Assembly Basics

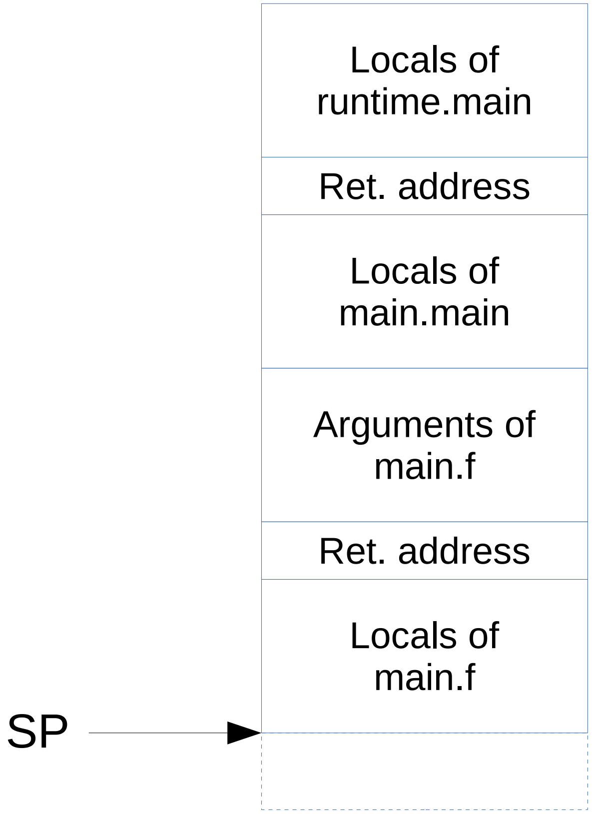
CPU

- Computers have CPUs
- CPUs have registers, in particular:
  - “Program Counter” (PC): address of the next instruction to execute
    - also known as Instruction Pointer, IP
  - “Stack Pointer” (SP): address of the “top” of the call stack
- CPUs execute assembly instructions that look like this:

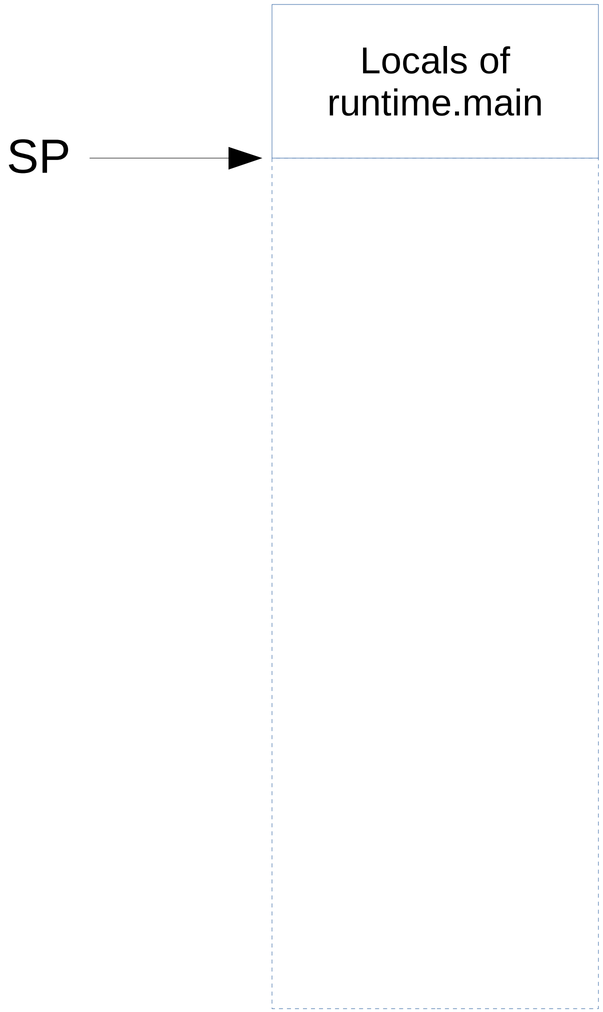
```
MOVQ DX, 0x58(SP)
```

Call Stack

- Stores arguments, local variables and return address of a function call



Call Stack



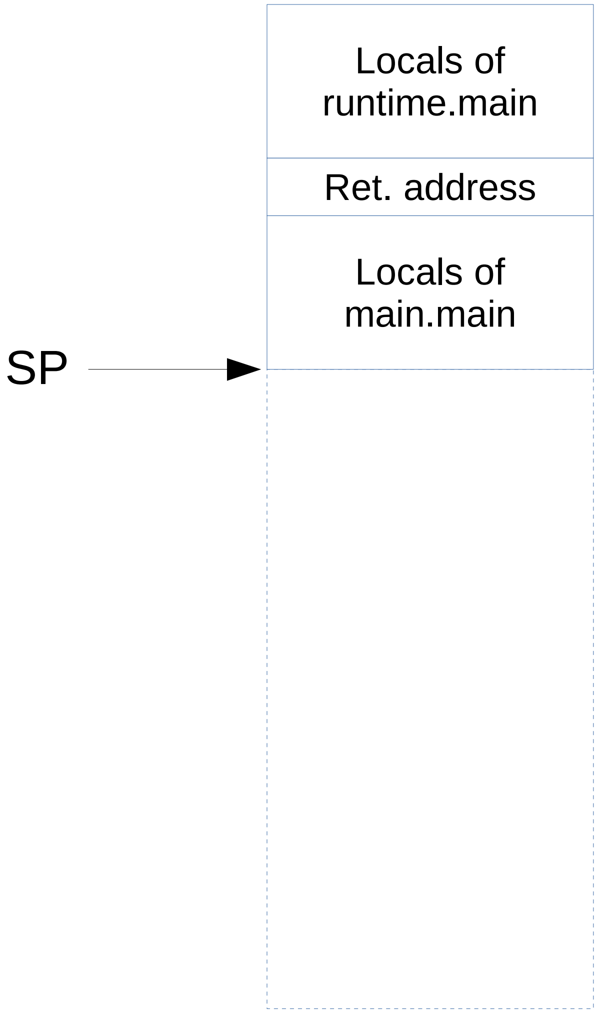
Goroutine 1 starts by calling runtime.main

Dotted box:
    Space allocated for the stack
Solid box:
    Space in use

Call Stack

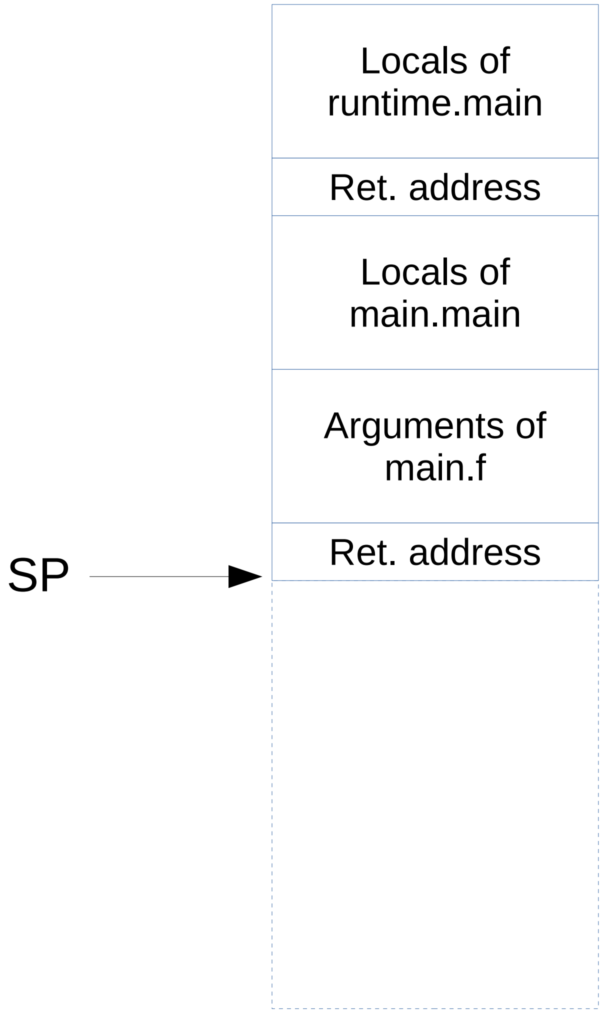
runtime.main calls main.main by pushing a return address on the stack

Call Stack



main.main pushes it’s local variables on the stack

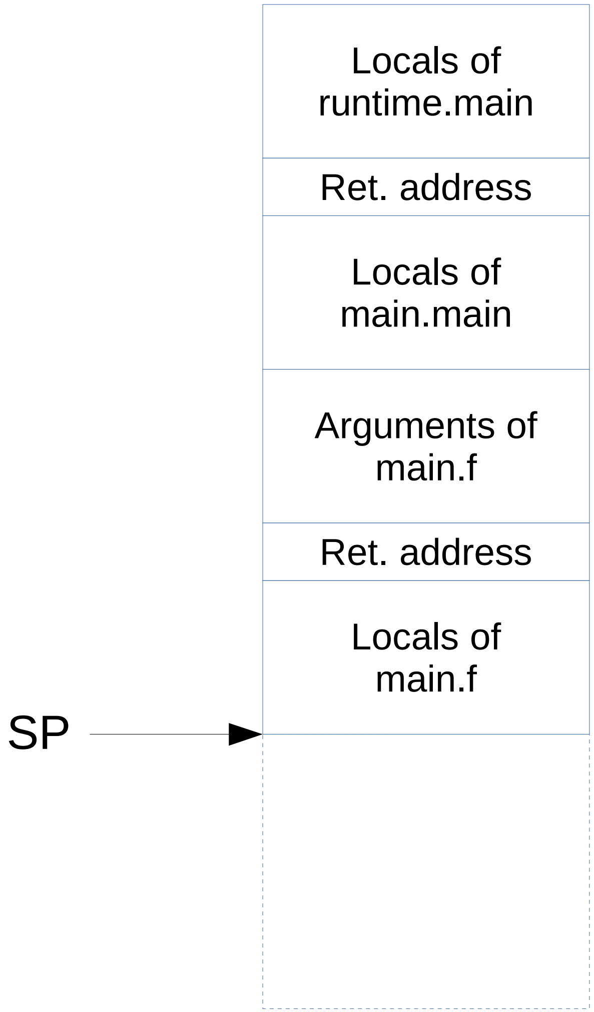
Call Stack



When main.main calls another function (main.f):

- pushes the arguments of main.f on the stack
- pushes the return value on the stack

Call Stack



Finally main.f pushes its local variables on the stack

Threads and Goroutines

- M:N threading / green threads
  - M goroutines are scheduled cooperatively on N threads
  - N initially equal to $GOMAXPROCS (by default the number of CPU cores)
- Unlike threads, goroutines:
  - are scheduled cooperatively
  - their stack starts small and grows/shrinks during execution

Threads and Goroutines

- When a go function is called
  - it checks that there is enough space on the stack for its local variables
  - if the space is not enough runtime.morestack_noctxt is called
  - runtime.morestack_noctxt allocates more space for the stack
  - if the memory area below the current stack is already used **the stack is copied somewhere else in memory and then expanded**
- Goroutine stacks **can move in memory**
  - debuggers normally assume stacks don’t move

Architecture of Delve

Architecture of Delve

Architecture of a Symbolic Debugger

UI Layer	
Symbolic Layer	knows about line numbers, types, variable names, etc.
Target Layer	controls target process, doesn’t know anything about your source code.

Features of the Target Layer

- Attach/detach from target process
- Enumerate threads in the target process
- Can start/stop individual threads (or the whole process)
- Receives “debug events” (thread creation/death and most importantly thread stop on a breakpoint)
- Can read/write the memory of the target process
- Can read/write the CPU registers of a stopped thread
  - actually this is the CPU registers saved in the thread descriptor of the OS scheduler

Target Layer in Delve (1)

- We have 3 implementations of the target layer:
  - `pkg/proc/native`: controls target process using OS API calls, supports:
    - Windows
      - WaitForDebugEvent, ContinueDebugEvent, SuspendThread...
    - Linux
      - ptrace, waitpid, tgkill..
    - macOS
      - notification/exception ports, ptrace, mach_vm_region…
  - default backend on Windows and Linux

Target Layer in Delve (2)

- Second implementation of Target Layer:
  - `pkg/proc/core`: reads linux_amd64 core files

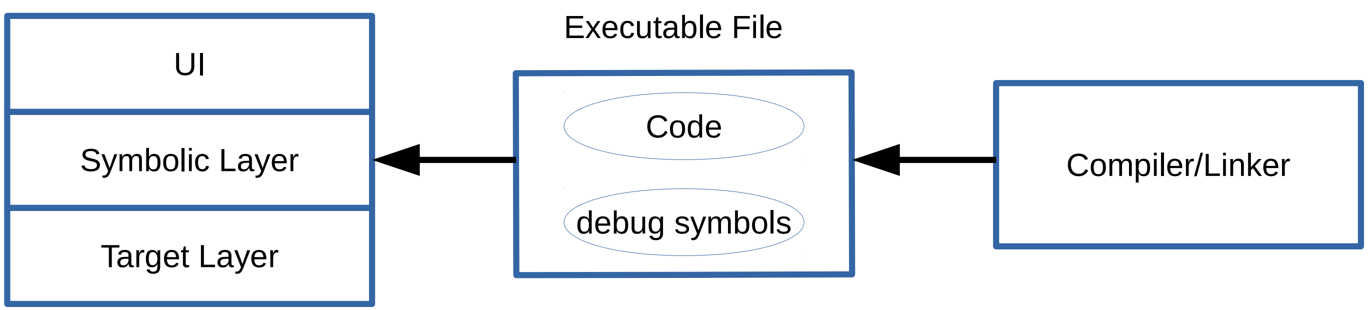
Target Layer in Delve (3)

- We have 3 (but really 5) implementations of the target layer:
  - `pkg/proc/gdbserial`: used to connect to:
    - debugserver on macOS (default setup on macOS)
    - lldb-server
    - Mozilla RR (a time travel debugger backend, only works on linux/amd64)
  - The name comes from the protocol it speaks, the Gdb Remote Serial Protocol
    - https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html
    - https://github.com/llvm-mirror/lldb/blob/master/docs/lldb-gdb-remote.txt

About debugserver

- `pkg/proc/gdbserial` connected to `debugserver` is the default target layer for macOS
- Two reasons:
  - the native backend uses undocumented API and never worked properly
  - the kernel API used by the native backend are restricted and require a signed executable
    - distributing a signed executable as an open source project is problematic
    - users often got the self-signing process wrong

Symbolic Layer



- Does its job by opening the executable file and reading the debug symbols that the compiler wrote
- The format of the debug symbols for Go is DWARFv4:

http://dwarfstd.org/

DWARF Sections (1)

- Defines many sections:

debug_info	debug_types	debug_loc
debug_ranges	debug_line	debug_pubnames
debug_pubtypes	debug_aranges	debug_macinfo
debug_frame	debug_str	debug_abbrev

DWARF Sections (1)

- The important ones:

debug_info	debug_types	debug_loc
debug_ranges	debug_line	debug_pubnames
debug_pubtypes	debug_aranges	debug_macinfo
debug_frame	debug_str	debug_abbrev

- debug_line: a table mapping instruction addresses to file:line pairs
- debug_frame: stack unwind information
- debug_info: describes all functions, types and variables in the program

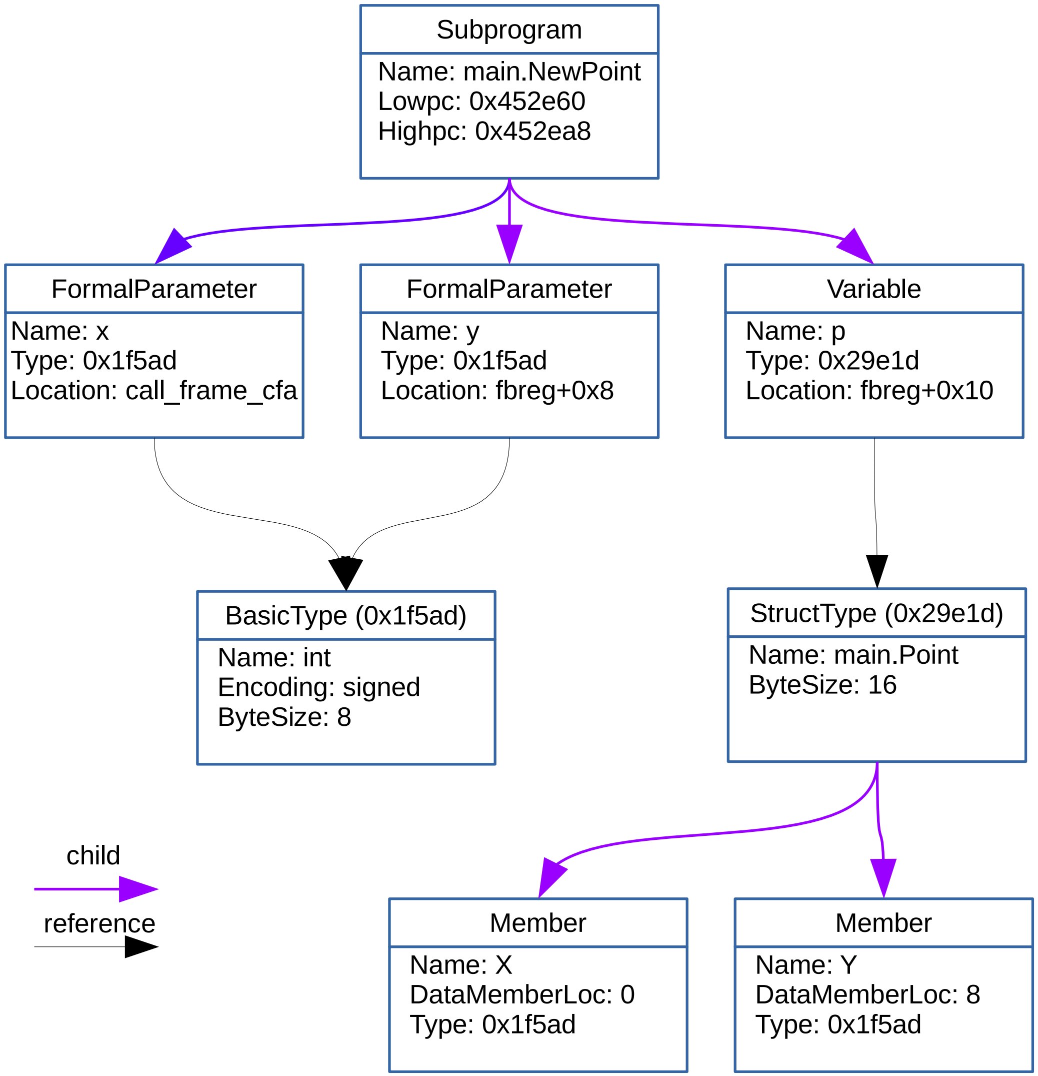
debug_info example (1)

```
package main

type Point struct {
    X, Y int
}

func NewPoint(x, y int) Point {
    p := Point{ x, y }
    return p
}
```

debug_info example (2)

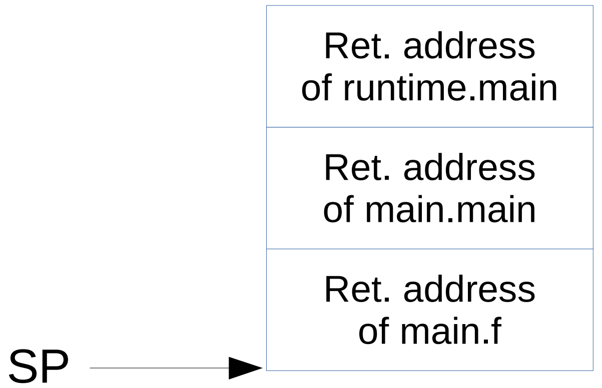


Stacktraces

```
2  0x00000000004519c9 in main.f
   at ./panicy.go:4
3  0x0000000000451a00 in main.main
   at ./panicy.go:8
4  0x0000000000426450 in runtime.main
   at /usr/local/go/src/runtime/proc.go:198
5  0x000000000044c021 in runtime.goexit
   at /usr/local/go/src/runtime/asm_amd64.s:2361
```

- Get the list of instruction addresses
  - `0x4519c9, 0x451a00, 0x426450, 0x44c021`
- Look up debug_info to find the name of the function
- Look up debug_line to find the source line correesponding to the instruction

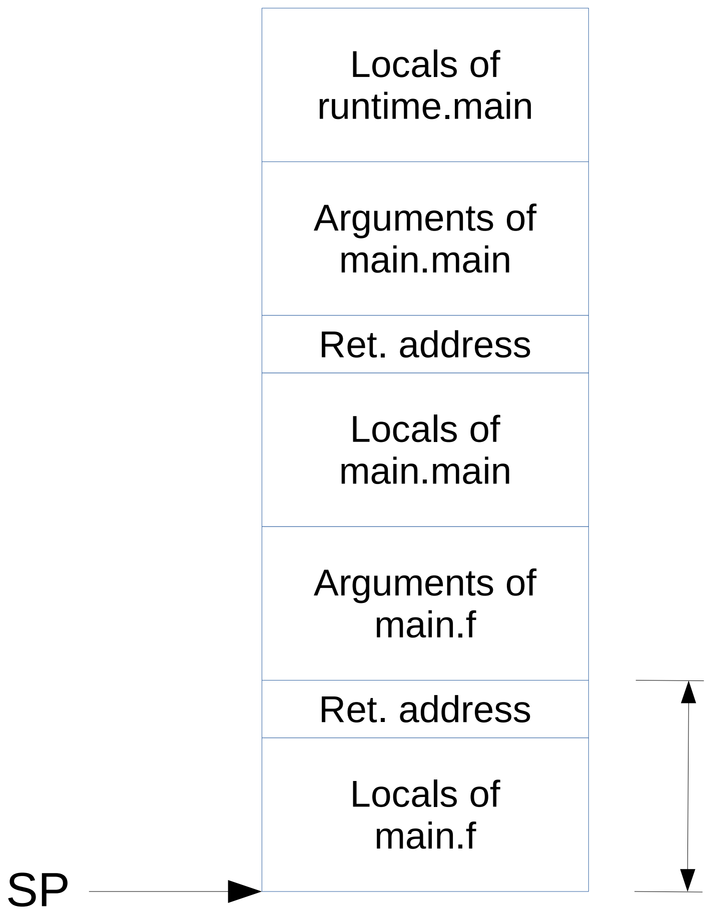
Stacktraces (2)



- If functions had no local variables of arguments this would be easy
- A stack trace is the value of PC register
- Followed by reading the stack starting at SP

debug_frame

- A table giving you the size of the current stack frame given the address of an instruction
  - Actually has many more features, but that’s the only thing you need for pure Go

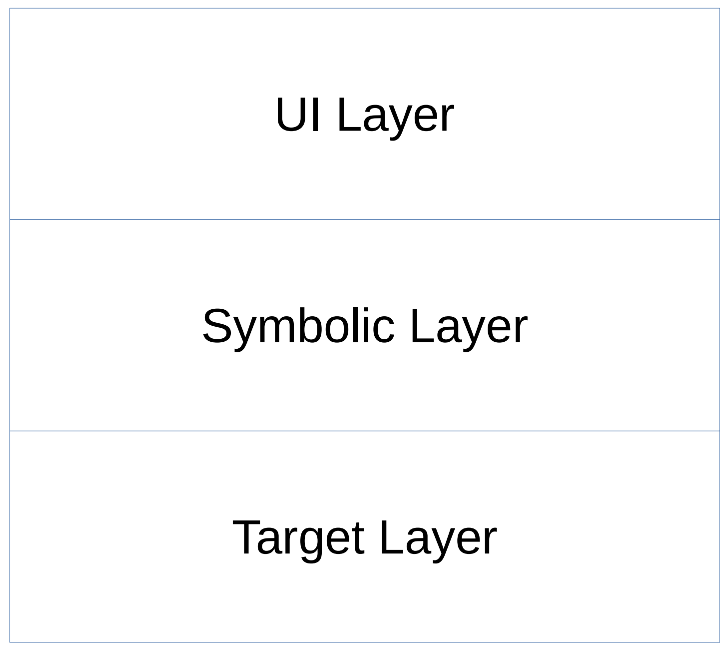


- To create a stack trace:
  - start with
    - PC_0 = the value of the PC register
    - SP_0 = the value of the SP register
  - look up PC_i in debug_frame
    - get size of the current frame sz_i
  - get return address ret_i at SP_i+sz_i-8
  - repeat the procedure with
    - $PC_{i+1} = reti$
    - $SP_{i+1} = SP_i+sz_i$
  - The stack trace is PC_0, PC_1, PC_2...

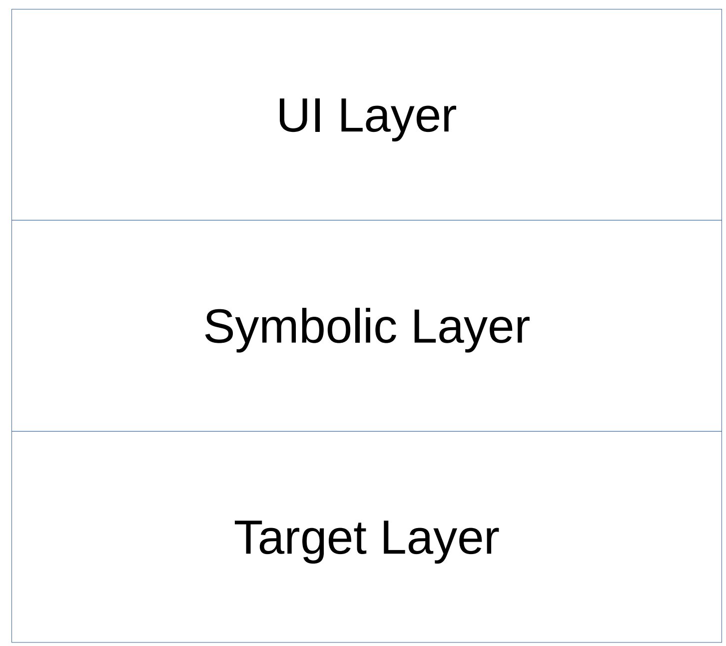
Symbolic Layer in Delve

- mostly `pkg/proc`
- support code in `pkg/dwarf` and stdlib `debug/dwarf`

Actual Architecture of Delve (1)

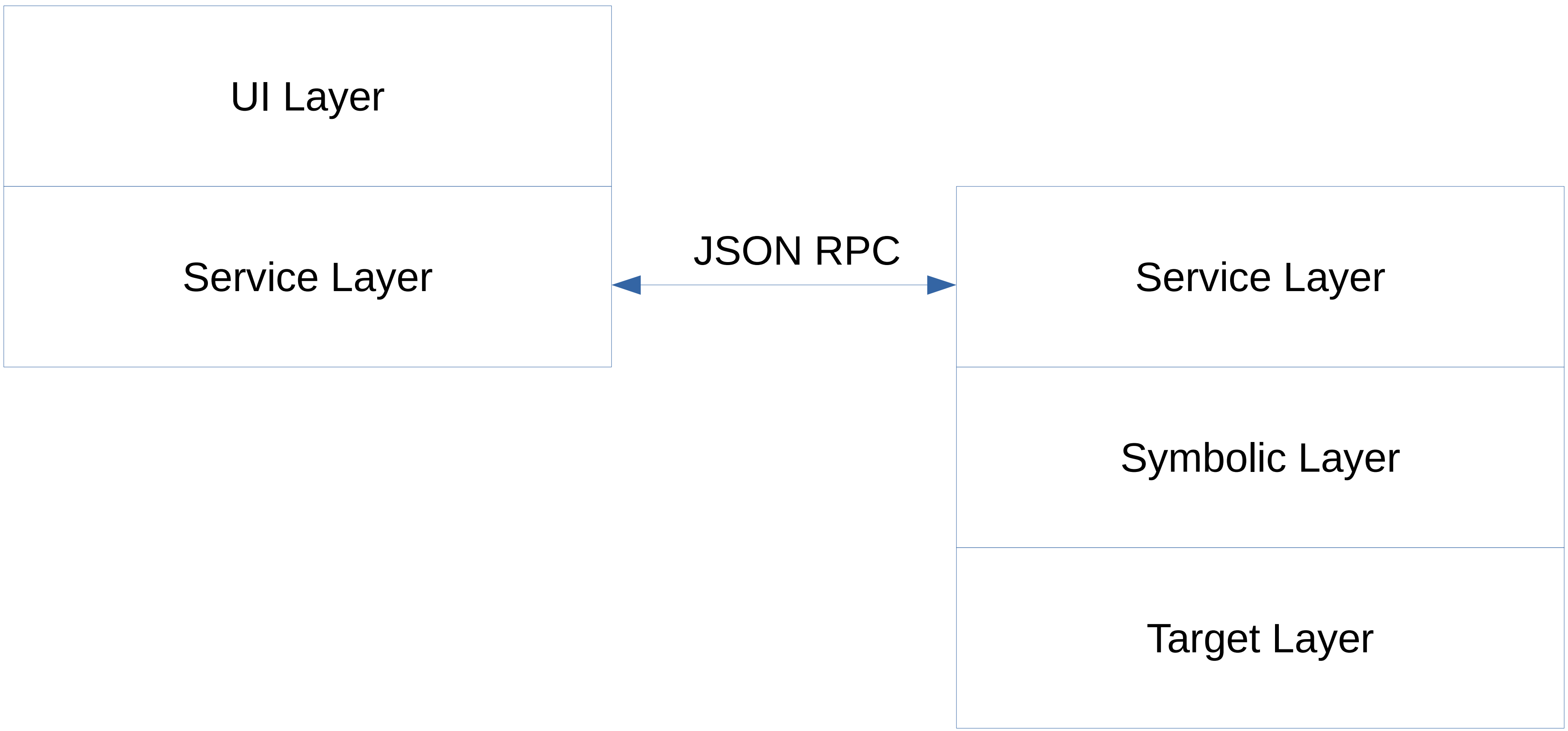


Actual Architecture of Delve (1)



This is a Lie

Actual Architecture of Delve (2)

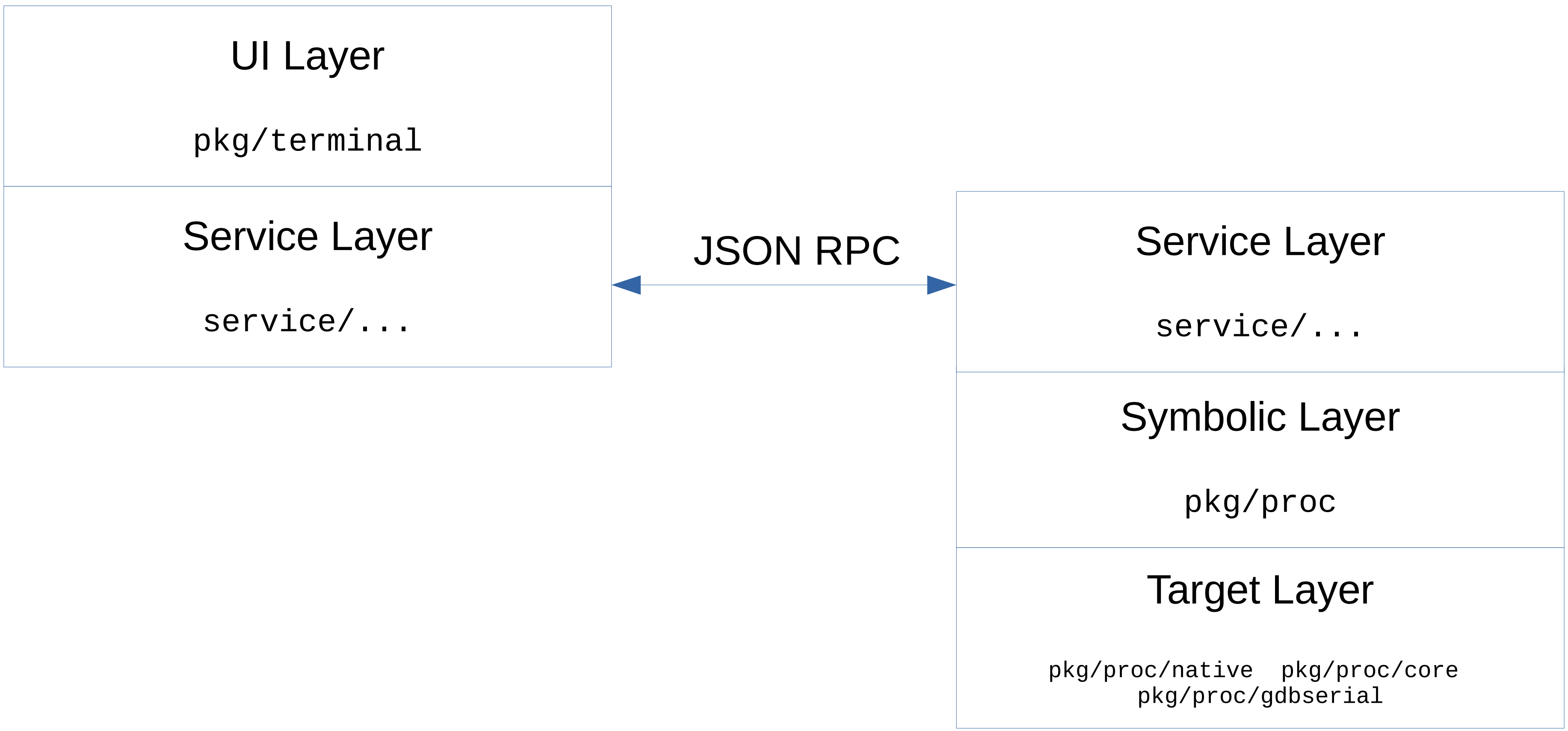


This makes embedding Delve into other programs easier

User Interfaces for Delve

- Built-in command line prompt
- Plugins
  - Atom plugin https://github.com/lloiser/go-debug
  - Emacs plugin https://github.com/benma/go-dlv.el/
  - Vim-go https://github.com/fatih/vim-go
  - VS Code Go https://github.com/Microsoft/vscode-go
- IDE
  - JetBrains Goland IDE https://www.jetbrains.com/go
  - LiteIDE https://github.com/visualfc/liteide
- Standalone GUI debuggers
  - Gdlv https://github.com/aarzilli/gdlv

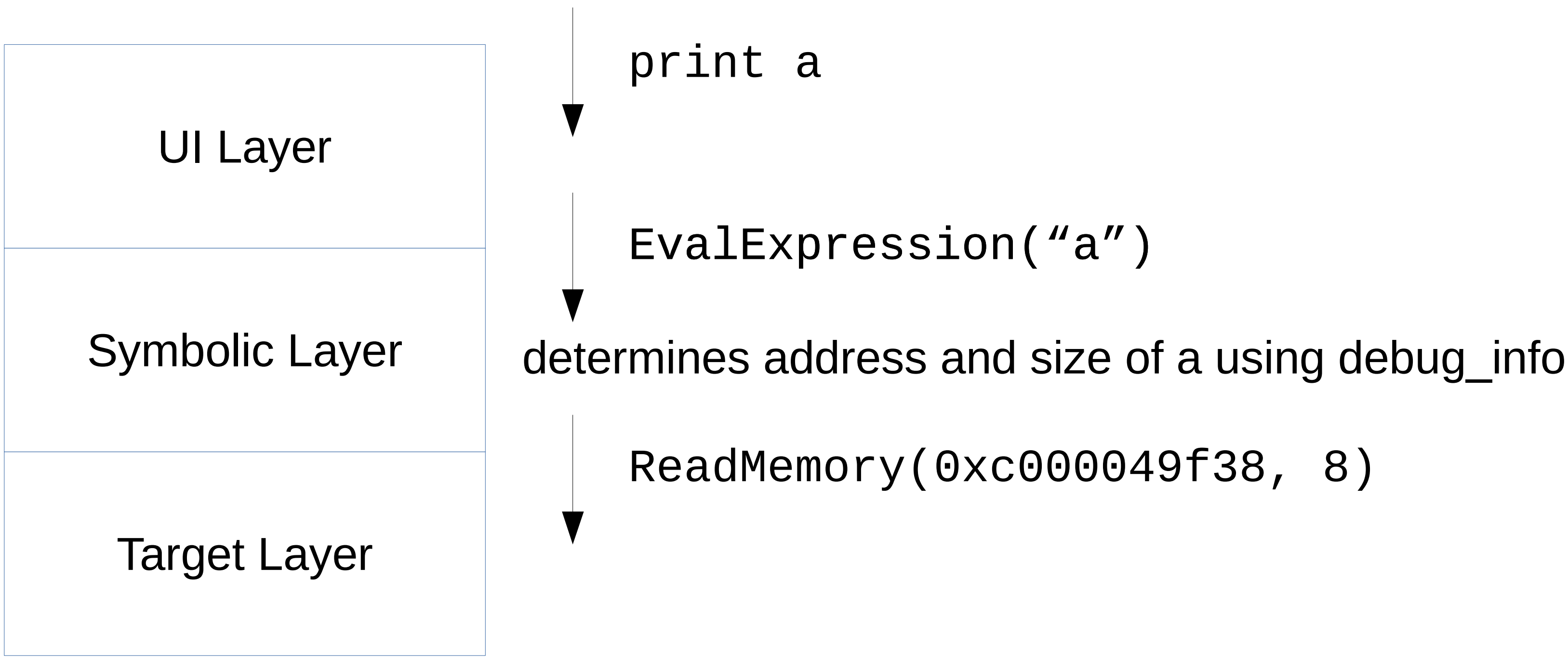
Actual Architecture of Delve (3)



Implementation of some Delve features

Variable Evaluation

(on the way down)



Variable Evaluation

(on the way up)


```
a = int(1)

Variable{
   Address: 0xc000049f38,
   Name: “a”,
   Type: “int”,
   Value: 1, ... }

[]byte{ 0x01, 0x00, 0x00… }
```

Variable Evaluation

gdb vs delve

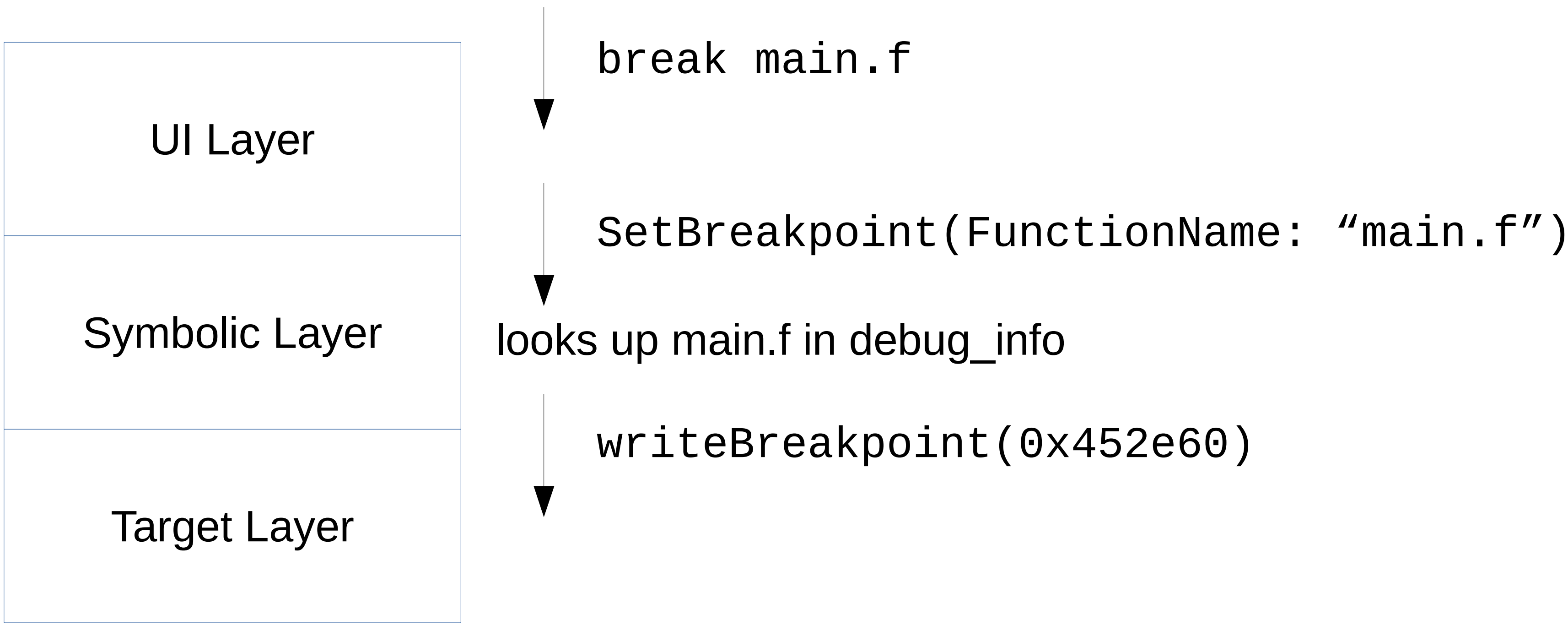
```
(gdb) p err1
$1 = {tab = 0x4f4ca0
<*main.astruct,error>, data =
0xc00008c030}
```

```
(dlv) print err1
error(*main.astruct) *{A: 1, B: 2}
```

```
(gdb) print ch1
$5 = (void *) 0xc0000b2000
```

```
(dlv) print ch1
chan int {
    qcount: 4,
    dataqsiz: 10,
    buf: *[10]int [1,4,3,2,0,0,0,0,0,0],
...
```

Creating Breakpoints



- The target layer overwrites the instruction at `0x452e60` with an instruction that, when executed, stops execution of the thread and makes the OS notify the debugger.
  - In intel amd64 it’s the instruction `INT 3` which is encoded as `0xCC`

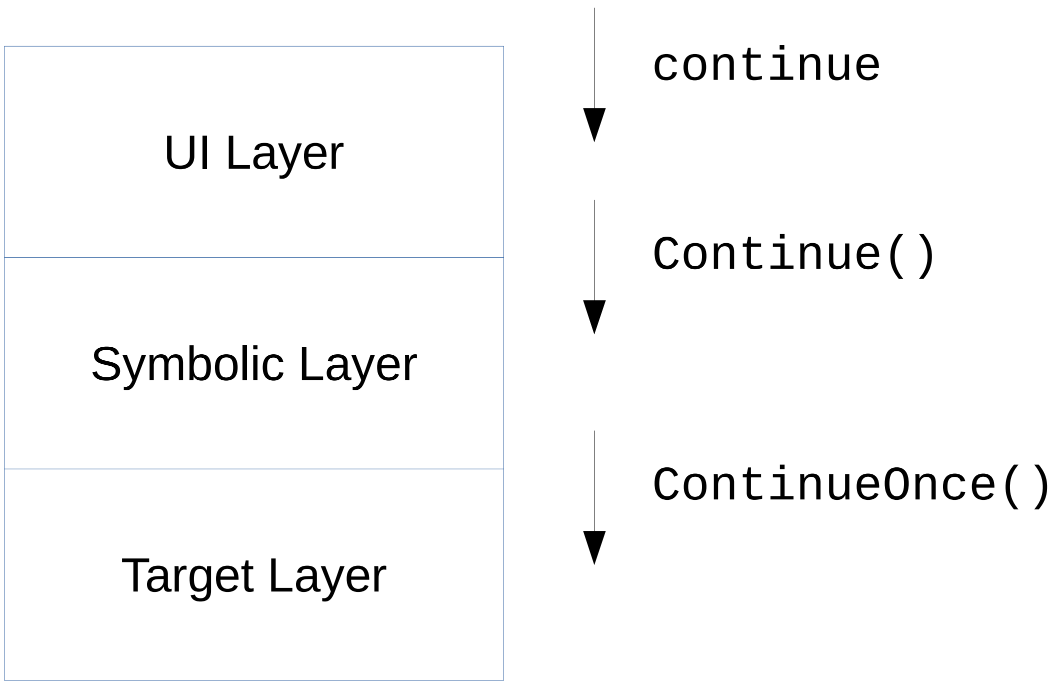
Creating Breakpoints

gdb vs delve

```
0x452eb0    MOVQ FS:0xfffffff8, CX                  gdb
0x452eb9    CMPQ 0x10(CX), SP
0x452ebd    JBE 0x452ee3
0x452ebf    SUBQ $0x28, SP                          dlv
0x452ec3    MOVQ BP, 0x20(SP)
0x452ec8    LEAQ 0x20(SP), BP
0x452ecd    XORPS X0, X0
0x452ed0    MOVUPS X0, 0(SP)
0x452ed4    CALL main.NewPoint(SB)
0x452ed9    MOVQ 0x20(SP), BP
0x452ede    ADDQ $0x28, SP
0x452ee2    RET
0x452ee3    CALL runtime.morestack_noctxt(SB)
0x452ee8    JMP main.main(SB)
```

- Instructions in red are the stack-split prologue
  - checks if the function needs more stack and calls runtime.morestack if it does
- A breakpoint set on the function's entry point will be hit twice if when the stack is resized, giving the impression that the function was executed twice

Continue
(on the way down)



- ContinueOnce resumes all threads and waits for a debug event

Continue

(on the way up)

`> main.main() ./main.go:200 (PC: 0x4a3277)`

list of running goroutines with their file:line position, the function they are executing and which breakpoint they are stopped at, if any

returns value of PC register for all threads

Mapping Goroutines to Threads

- Each goroutine is described by a runtime.g struct

```
type g struct {
    stack       stack
    stackguard0 uintptr
    stackguard1 uintptr

    _panic         *_panic // innermost panic - offset known to liblink
    _defer         *_defer // innermost defer
    ...
    goid           int64
    ...
}
```

- All g structs are saved into runtime.allgs
- The goroutine running on a given thread is stored in the Thread's Local Storage
  - Actual implementation varies depending on GOOS and GOARCH
    - linux/amd64: FS:0xfffffff8
    - windows/amd64: GS:0x28
    - macOS/amd64: GS:0x8a0 or GS:0x30 (starting with go1.11)

Conditional Breakpoints

- A breakpoint that should stop the execution of the program only when a boolean condition is true
- Setting them is the same as setting normal breakpoints
- When `ContinueOnce` (target layer) returns:
  - `Continue` (symbolic layer) evaluates the condition(s) associated with (all) the current breakpoint(s)
  - if it’s true `Continue` returns
  - otherwise `ContinueOnce` is called again.
- Optimizations are possible
  - Peter B. Kessler. 1990. Fast Breakpoints: Design and Implementation. PLDI '90 Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation. Pages 78-84

Step Over

- Executes one line of source code, “steps over” function calls
- Also known as “next”

Wrong “next” strategy, step 0

```
package main

func fib(n int) int {
    if n == 0 {
        return 1
    }
    if n == 1 {
        return 1
    }
    a := fib(n-1)
    b := fib(n-2)
    return a+b
}

func main() {
    r := fib(10)
    println(r)
}
```

Wrong "next" strategy, step 1

```
package main

func fib(n int) int {
    if n == 0 {
        return 1
    }
    if n == 1 {
        return 1
    }
    a := fib(n-1)
    b := fib(n-2)
    return a+b
}

func main() {
    r := fib(10)
    println(r)
}
```

Set a breakpoint on every line of the current function

Wrong “next” strategy, step 2

```
package main

func fib(n int) int {
    if n == 0 {
        return 1
    }
    if n == 1 {
        return 1
    }
    a := fib(n-1)
    b := fib(n-2)
    return a+b
}

func main() {
    r := fib(10)
    println(r)
}
```

Set a breakpoint on the return address of the current frame

Wrong “next” strategy, step 3

- Set a breakpoint on the first deferred function
- Call continue

Wrong “next” strategy, bug 1:

Can’t handle concurrency

```
package main

func fib(n int) int {
    if n == 0 {
        return 1
    }
    if n == 1 {
        return 1
    }
    a := fib(n-1)
    b := fib(n-2)
    return a+b
}

func main() {
    for i := 1; i < 10; i++ {
        go func() {
            r := fib(i)
            println(r)
        }()
    }
}
```

Wrong “next” strategy, bug 2:

Can’t handle recursion

```
package main

func fib(n int) int {
    if n == 0 {
        return 1
    }
    if n == 1 {
        return 1
    }
    a := fib(n-1)
    b := fib(n-2)
    return a+b
}

func main() {
    r := fib(i)
    println(r)
}
```

Better “next” strategy

- Set a breakpoint on every line of the current function
  - condition: stay on the same goroutine & stack frame
- Set a breakpoint on the return address of the current frame
  - condition: stay on the same goroutine & previous stack frame
- Set a breakpoint on the most recently deferred function
  - condition: stay on the same goroutine & check that it was called through a panic
- Call `Continue`

Better “next” strategy

gdb vs. delve

- gdb doesn’t know about defer
- gdb doesn’t know about goroutines
- gdb can’t check that we didn’t change stack frame
  - goroutine stacks will move when resized
  - gdb assumes stacks always stay in the same place

Implementing “next” checks

- “same goroutine” check:
  - read the goid field of the runtime.g struct on the current thread
- “same frame” check:
  - SP + current_frame_size – g.stack.stackhi
    - where g is the runtime.g struct for the current thread

The End