Go: syscalls and the scheduler

Rahul Tiwari

Software Engineer at Simpl

Backend Developer.

Can write Go and Python.

Loves football, FOSS, classical music and metacognition in the order mentioned.

Why should you spend the next 15 minutes or so listening to me?

- Understand how high level languages function under the hood
- To become capable in understanding and debugging your program
- And if you're curious in general

What are syscalls?

```
SYSCALLS(2)              Linux Programmer's Manual            SYSCALLS(2)
NAME         top
       syscalls - Linux system calls
SYNOPSIS         top
       Linux system calls.
DESCRIPTION         top
       The system call is the fundamental interface between an
       application and the Linux kernel.
```

How do I see syscalls?

strace

```
STRACE(1)                  General Commands Manual                  STRACE(1)
NAME
       strace - trace system calls and signals
DESCRIPTION
       In the simplest case strace runs the specified command until it exits.
       It intercepts and records the system calls which are
       called by a process and the signals which are received by a process.
```

dtruss (A handy Mac replacement)

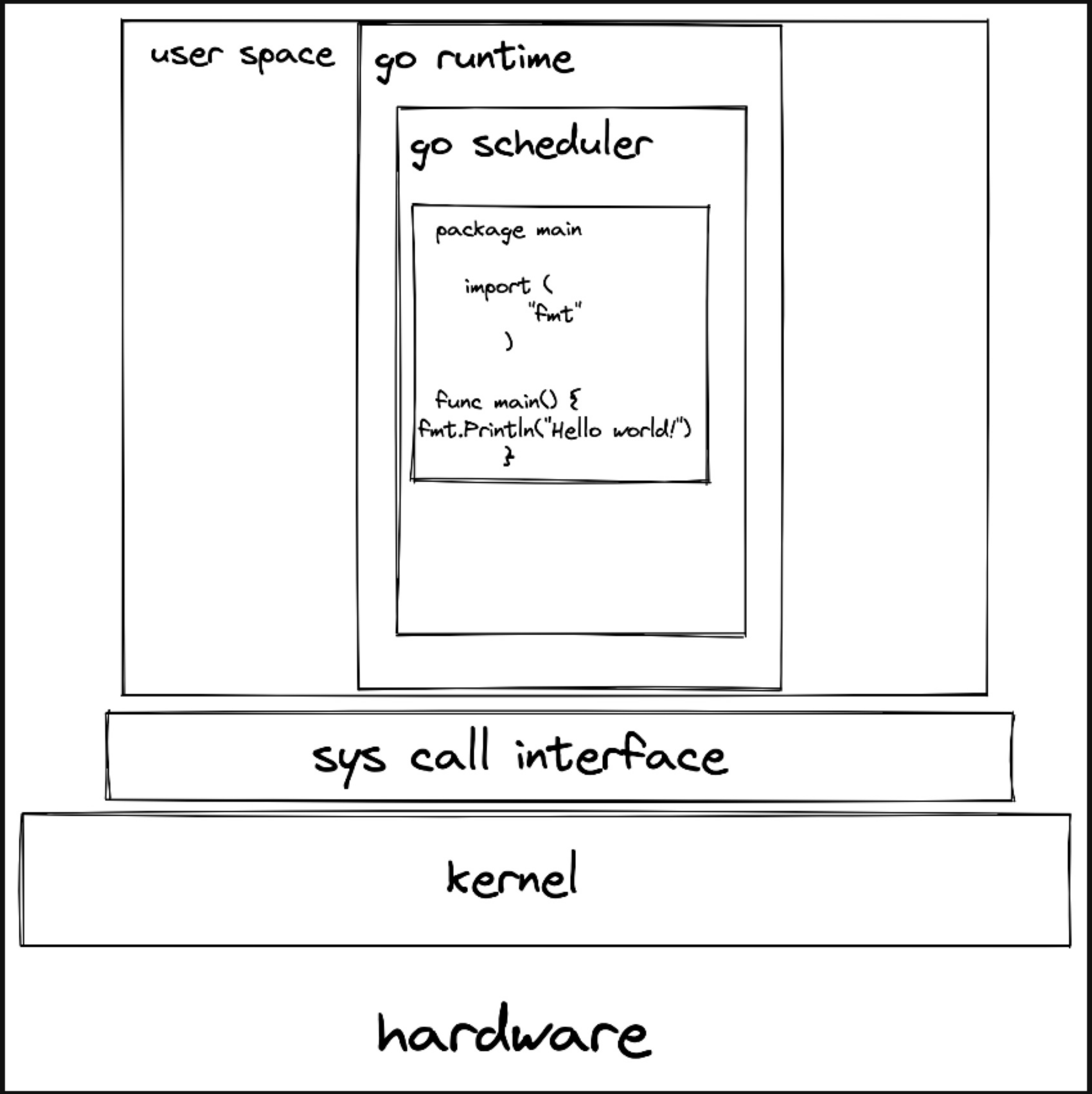
```
dtruss(1m)
NAME
       dtruss - process syscall details. Uses DTrace.
DESCRIPTION
       dtruss prints details on process system calls. It is like a DTrace
       version of truss, and has been designed to be less intrusive than truss.
```

Coming back to Go

```go
// A simple Hello world Go program
package main

import (
  "fmt"
)

func main() {
  fmt.Println("Hello world!")
}
```


user space
go runtime
go scheduler
package main
import (
"fmt"
)
func main() {
fmt.Println("Hello world!")
}
sys call interface
kernel
hardware

Tracing syscalls

```
# Output of sudo dtruss -f ./main

  PID/THRD  SYSCALL(args)       = return
Hello world!
16028/0x1d793:  fork()      = 0 0
16028/0x1d793:  access("/AppleInternal/XBS/.isChrooted\0
 ↳ ", 0x0, 0x0)     = -1 Err#2
16028/0x1d793:
 ↳   bsdthread_register(0x193D79084, 0x193D79078, 0x4000) =
 ↳ 1073742303 0
16028/0x1d793:
 ↳   bsdthread_create(0x1046644C0, 0x1400003C000, 0x16B883000) =
 ↳ 1804087296 0
16028/0x1d793:  __pthread_sigmask(0x3, 0x16B7FB6C8, 0x0) = 0 0
16028/0x1d796:  fork()      = 0 0
16028/0x1d793:
 ↳   bsdthread_create(0x1046644C0, 0x1400003C480, 0x16B90F000) =
 ↳ 1804660736 0
16028/0x1d797:  fork()      = 0 0
16028/0x1d793:  sigreturn(0x14000009C18, 0x1E, 0x97346E51C6D2C79B) =
 ↳ 0 Err#-2
16028/0x1d797:
 ↳   bsdthread_create(0x1046644C0, 0x14000080000, 0x16B99B000) =
 ↳ 1805234176 0
16028/0x1d797:  __pthread_sigmask(0x3, 0x16B90ECD8, 0x0) = 0 0
16028/0x1d793:  madvise(0x1400005C000, 0x8000, 0x8) = 0 0
16028/0x1d798:  fork()      = 0 0
16028/0x1d796:  __semwait_signal(0x903, 0x0, 0x1) = -1 Err#60
16028/0x1d793:  mlock(0x14000060000, 0x4000, 0x0) = 0 0
16028/0x1d798:  thread_selfid(0x0, 0x0, 0x0)  = 120728 0
16028/0x1d793:  __pthread_sigmask(0x3, 0x10472C1B0, 0x16B7FB588) = 0
 ↳  0
16028/0x1d798:  sigaltstack(0x0, 0x16B99AE70, 0x0) = 0 0
16028/0x1d797:  __semwait_signal(0x903, 0x0, 0x1) = -1 Err#60
16028/0x1d798:  sigaltstack(0x16B99AE30, 0x0, 0x0) = 0 0
16028/0x1d798:  __pthread_sigmask(0x3, 0x16B99AE84, 0x0) = 0 0
16028/0x1d793:
 ↳   bsdthread_create(0x1046644C0, 0x1400003C900, 0x16BA27000) =
 ↳ 1805807616 0
16028/0x1d798:  psynch_cvsignal(0x1400003C820, 0x100, 0x0) = 257 0
16028/0x1d799:  fork()      = 0 0
028/0x1d793:  write(0x1, "Hello world!\n\0", 0xD)    = 13 0
16028/0x1d796:  __semwait_signal(0x903, 0x0, 0x1) = -1 Err#60
16028/0x1d793:  kqueue(0x0, 0x0, 0x0)    = 3 0
```

Why more than one thread for just printing "Hello world!"

```
// Allow newproc to start new Ms.
mainStarted = true

if GOARCH != "wasm" { // no threads on wasm yet, so no sysmonsystemstack(func() {
newm(sysmon, nil, -1)
})
}

// Lock the main goroutine onto this, the main OS thread,
// during initialization. Iost programs won't care, but a few
// do require certain calls to be made by the main thread.
// Those can arrange for main.main to run in the main thread
// by calling runtime.LockOSThread during initialization
// to preserve the lock.
lockOSThread()
```

- sysmon starts a new thread to run the system monitor.
- Also, the main thread is blocked by the go runtime and hence the Go scheduler has to start a new thread.
- Other threads are needed for running the GC, timing etc.
- Quoting Go runtime

> *The GOMAXPROCS variable limits the number of operating system threads that can execute user-level Go code simultaneously. There is no limit to the number of threads that can be blocked in system calls on behalf of Go code; those do not count against the GOMAXPROCS limit.*

```go
package main

import (
  "fmt"
  "runtime"
  "runtime/pprof"
)

func main() {
  var threadProfile = pprof.Lookup("
 ↳ threadcreate")
  fmt.Printf("Number of logical CPUs %d\n
 ↳ ", runtime.NumCPU())
  fmt.Printf("Number of OS threads %d\n
 ↳ ", threadProfile.Count())
  fmt.Printf("Number of goroutines %d\n
 ↳ ", runtime.NumGoroutine())
  fmt.Println("Hello world!")
}
```

```
Number of logical CPUs 8
Number of OS threads 5
Number of goroutines 1
Hello world!
```

Diving into the Go scheduler

The scheduler's job is to distribute ready-to-run goroutines over worker threads.

P	M	G
processor, resource that is required to execute Go code.	worker thread, or machine.	goroutine

LRQ	GRQ
Local Run Queue, each P has its own to manage coroutines	for goroutines that have not been assigned to a P yet

Scheduling paradigms

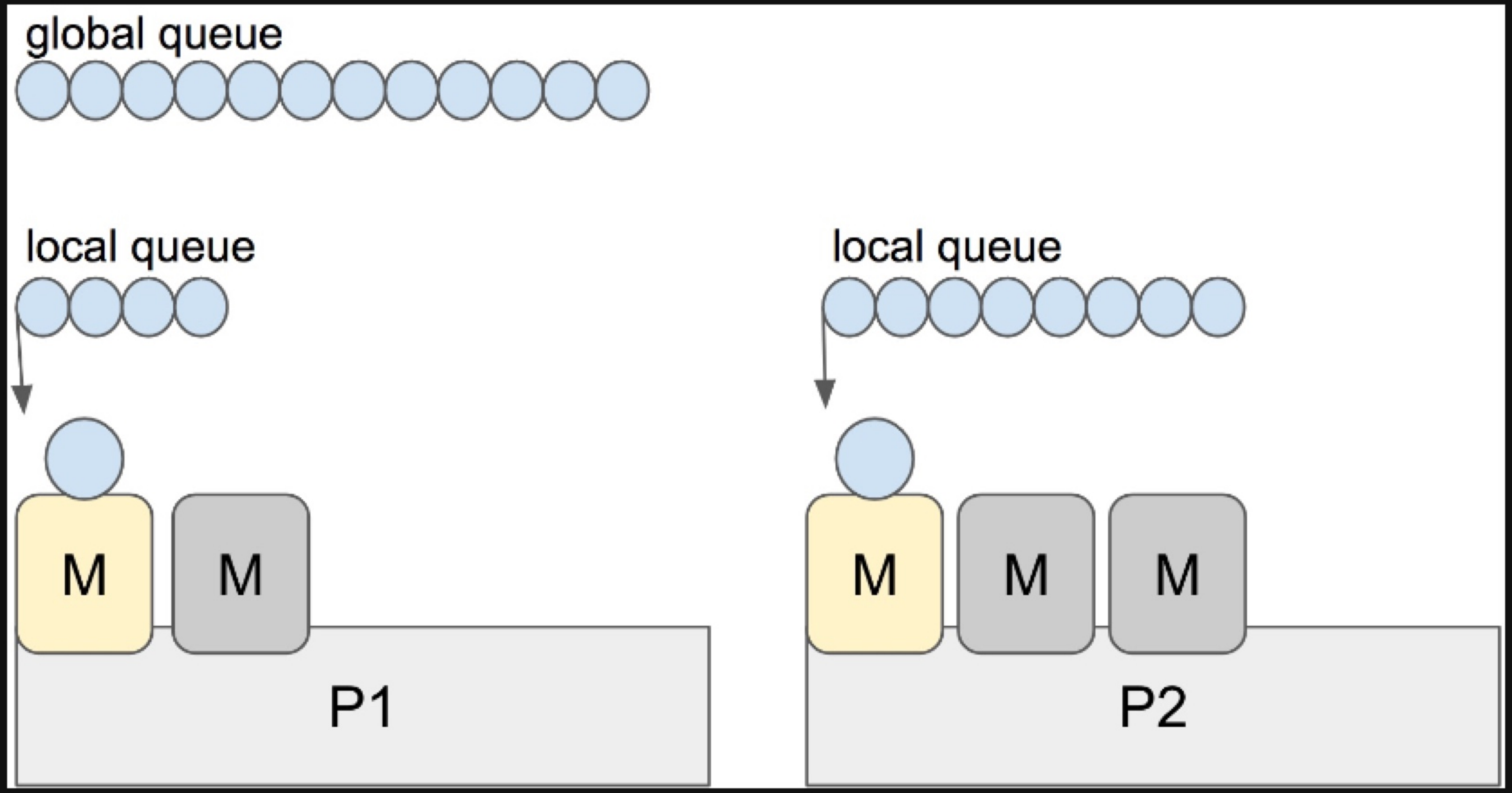
Work stealing

An underutilized processor actively looks for other processor's threads and "steal" some.

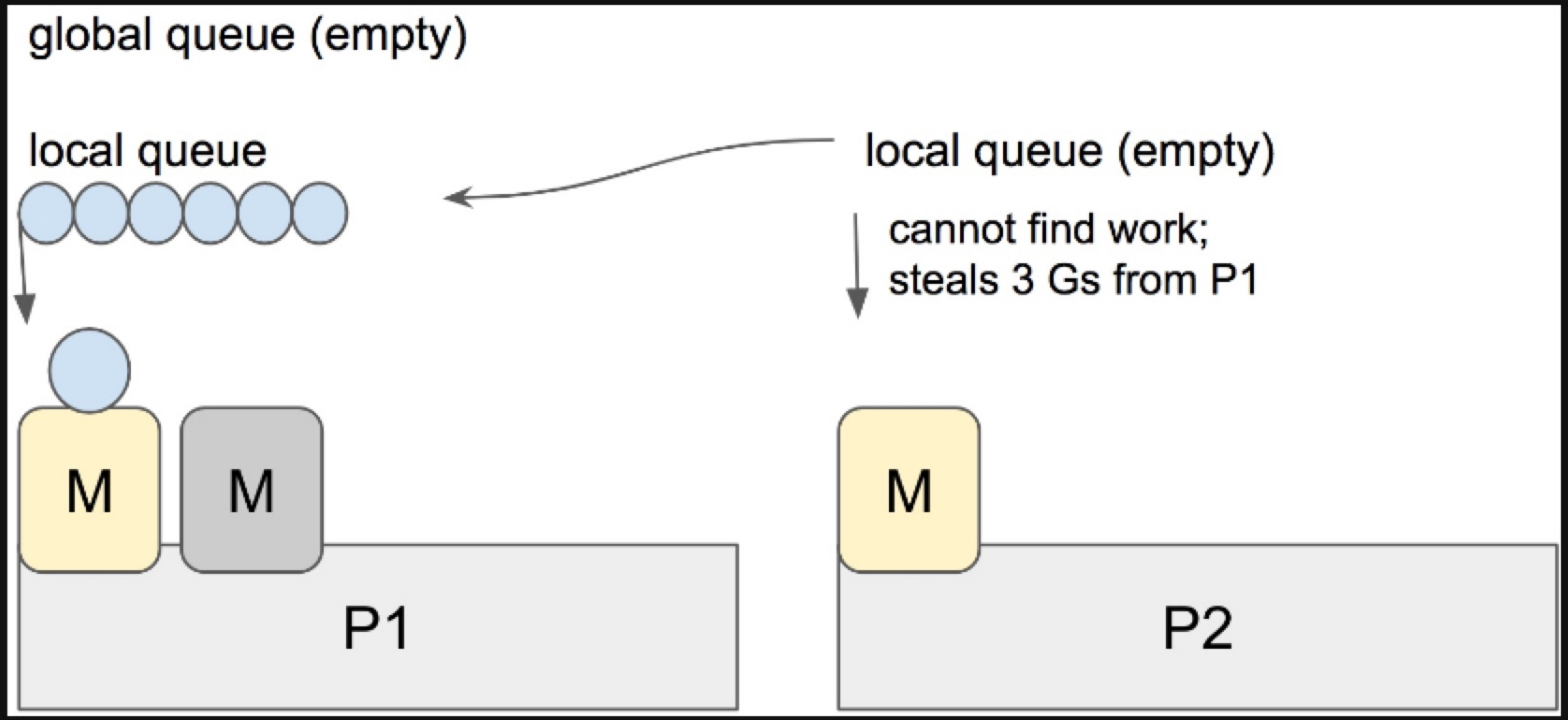
Work sharing

When a processor generates new threads, it attempts to migrate some of them to the other processors with the hopes of them being utilized by the idle/underutilized processors.

```go
↳ // One round of scheduler: find a runnable
// goroutine and execute it.
// Never returns.
func schedule() {
```



```go
// Steal half of elements from local
// runnable queue of p2 and put onto
// local runnable queue of p.
// Returns one of the stolen elements
// (or nil if failed).
func runqsteal(_p_, p2 *p, stealRunNextG
 ↳ bool) *g {
```



Does this really happen?

```
env GODEBUG=scheddetail=1,schedtrace=2 GOMAXPROCS=2 go run main.go
```

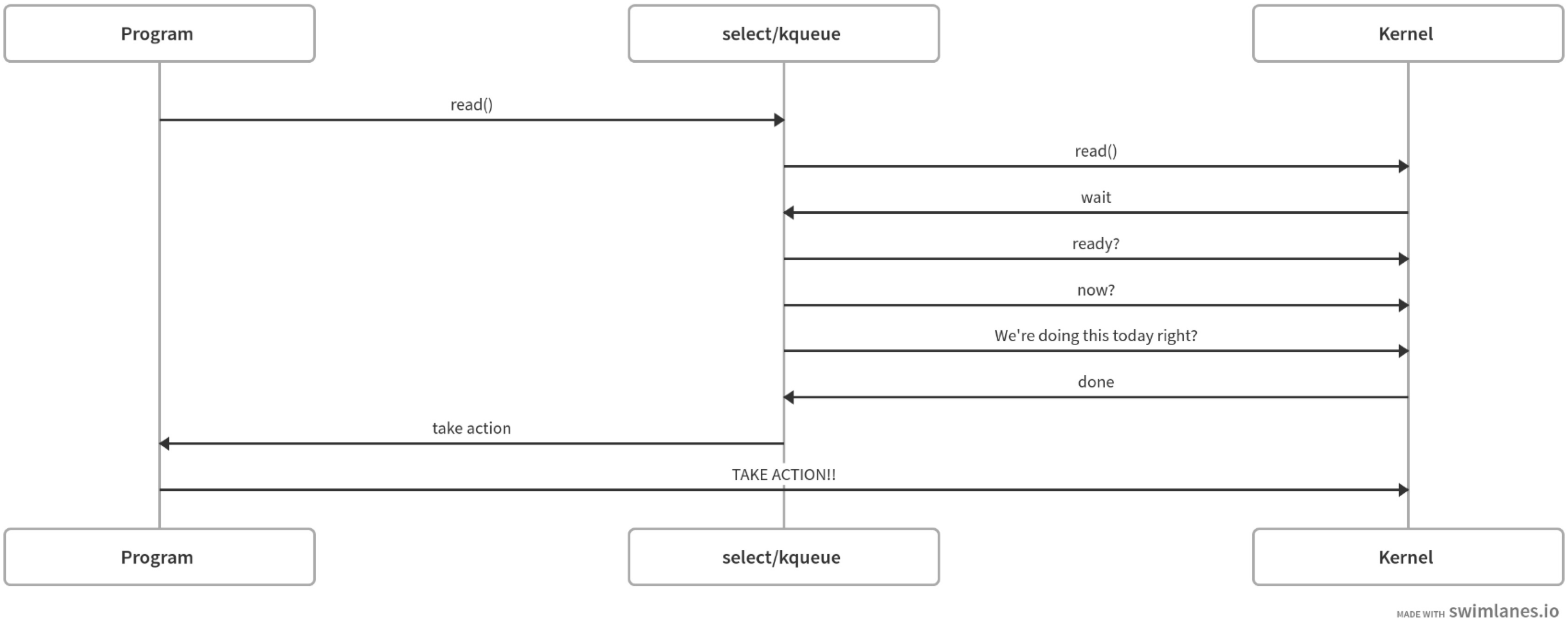
```
SCHED
 ↳ 0ms: gomaxprocs=2 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0 runqueue=0 gcwaiting=0 nmidlelocke
  P0: status=1 schedtick=0 syscalltick=0 m=4 runqsize=0 gfreecnt=0 timerslen=0
  P1: status=1 schedtick=2 syscalltick=0 m=3 runqsize=0 gfreecnt=0 timerslen=0
  M4:
 ↳ p=0 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 spinning=true blocked=false lockedg=-1
  M3:
 ↳ p=1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 spinning=false blocked=false lockedg=-1
  M2:
 ↳ p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=2 dying=0 spinning=false blocked=false lockedg=-1
M1:
 ↳ p=-1 curg=17 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 spinning=false blocked=false lockedg=17
  M0:
 ↳ p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 spinning=false blocked=false lockedg=1
  G1: status=1(chan receive) m=-1 lockedm=0
  G17: status=6() m=1 lockedm=1
  G2: status=4(force gc (idle)) m=-1 lockedm=-1
  G3: status=4(GC sweep wait) m=-1 lockedm=-1
```

What does the future of syscalls look like?

What happens right now?

- All UNIX IO syscalls are synchronous and blocking
- For example, a program calls `read()`, goes to sleep until the descriptor is ready
- `select()` and `kqueue` wake processes so that they can go and perform an action

Traditional async



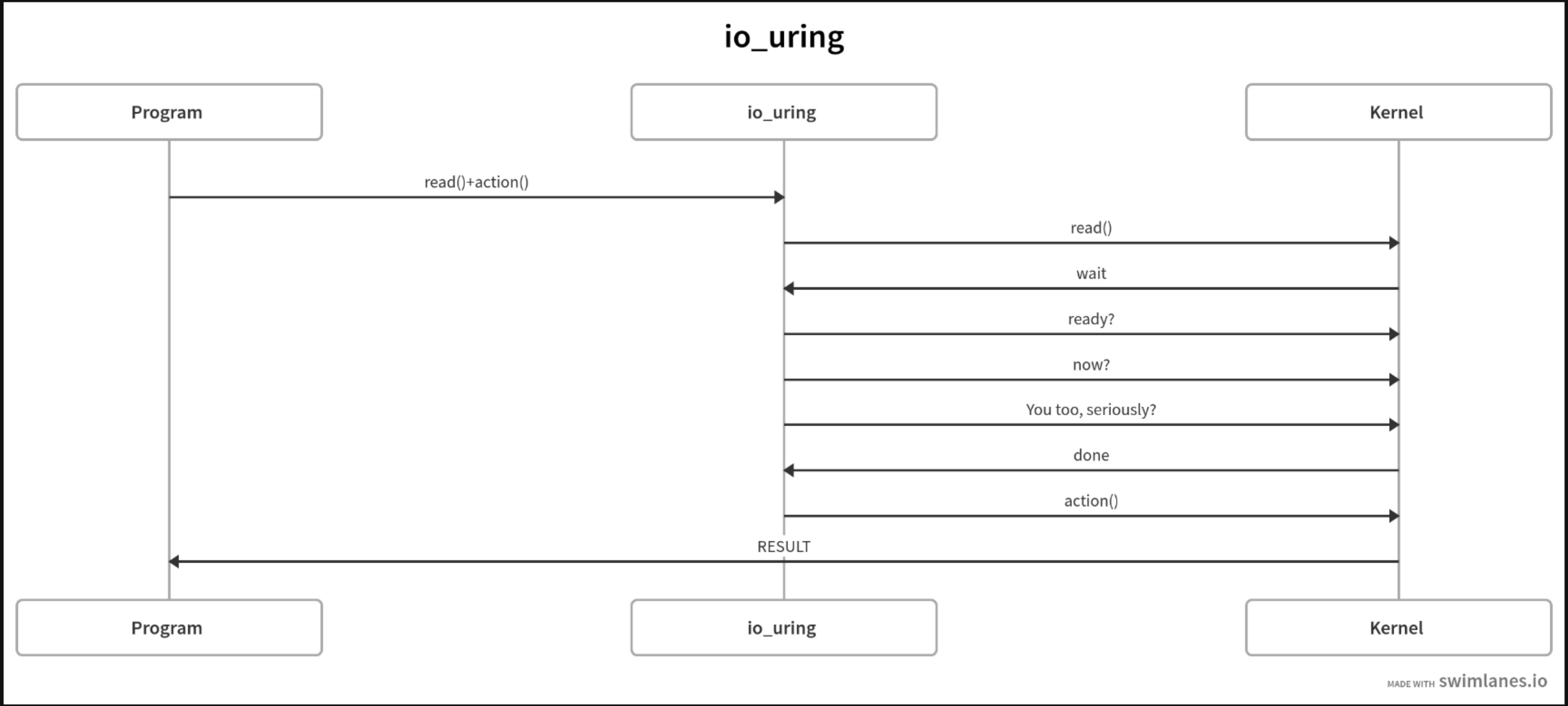
io_uring

- io_uring subsystem released in mainline kernel in 2019

- Solves for inherently synchronous Unix I/O

- Built around a ring buffer in memory shared between user space and kernel

- Allows submission of operations and collection of results asynchronously.


io_uring
Program
io_uring
Kernel
read()+action()
read()
wait
ready?
now?
You too, seriously?
done
action()
RESULT
Program
io_uring
Kernel
MADE WITH swimlanes.io

Questions?

Thank you!