Go scheduler

Implementing language with lightweight concurrency

Dmitry Vyukov, dvyukov@gmail.com

Hydra conf, July 12 2019

Agenda

- Go specifics
- Scheduler
- Scalability
- Fairness
- Stacks
- Future

What is a **goroutine**?

Logically a **thread** of execution.

Logically same as:

- OS thread
- coroutine
- green thread

Most material is generic

... and to large degree applicable to:

- OS thread schedulers
- Coroutine schedulers
- Thread pools
- Other languages

Go specifics:

1. Current Go design decisions
2. Go requirements and constraints

Go specifics:

1. Current Go design decisions
2. Go requirements and constraints:

- goroutines are lightweight (1M)

Go specifics:

1. Current Go design decisions
2. Go requirements and constraints:

- goroutines are lightweight (1M)
- parallel and scalable

Go specifics:

1. Current Go design decisions
2. Go requirements and constraints:

- goroutines are lightweight (1M)
- parallel and scalable
- minimal API (no hints)

Go specifics:

1. Current Go design decisions
2. Go requirements and constraints:

- goroutines are lightweight (1M)
- parallel and scalable
- minimal API (no hints)
- infinite stack

Go specifics:

1. Current Go design decisions
2. Go requirements and constraints:

- goroutines are lightweight (1M)
- parallel and scalable
- minimal API (no hints)
- infinite stack
- handling of IO, syscalls, C calls

A taste of Go

```go
resultChan := make(chan Result)  // FIFO queue
go func() {                      // start a goroutine
    response := sendRequest()    // blocks on IO
    result := parse(response)
    resultChan <- result         // send the result back
}()
process(<-resultChan)            // receive the result
```

How can we implement this?

Thread per goroutine?

Would work!

But too expensive:

- memory (at least 32K or so)
- performance (syscalls)
- no infinite stacks

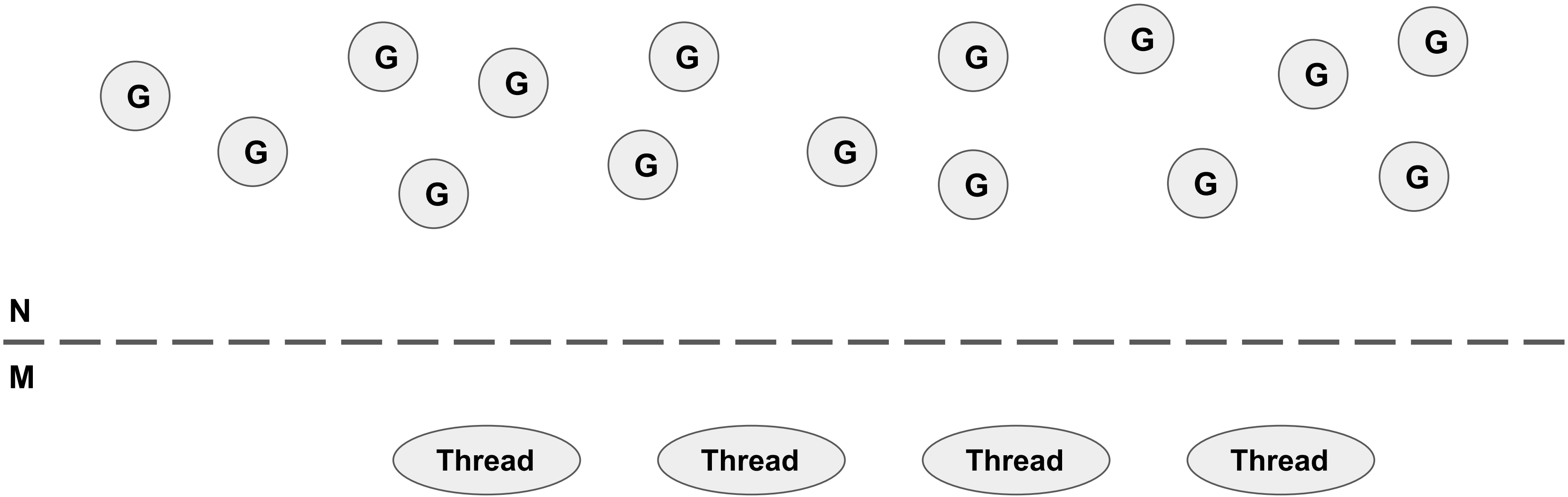
Thread pool?

Only faster goroutine creation.

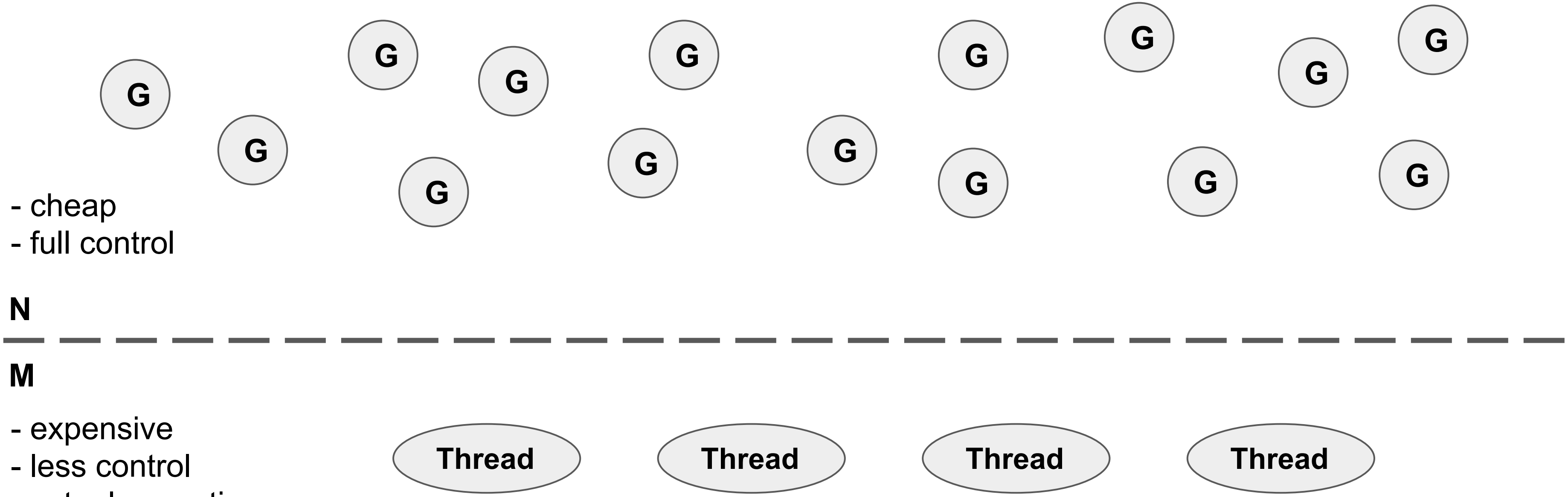
But still:

- memory consumption
- performance
- no infinite stacks

M:N Threading



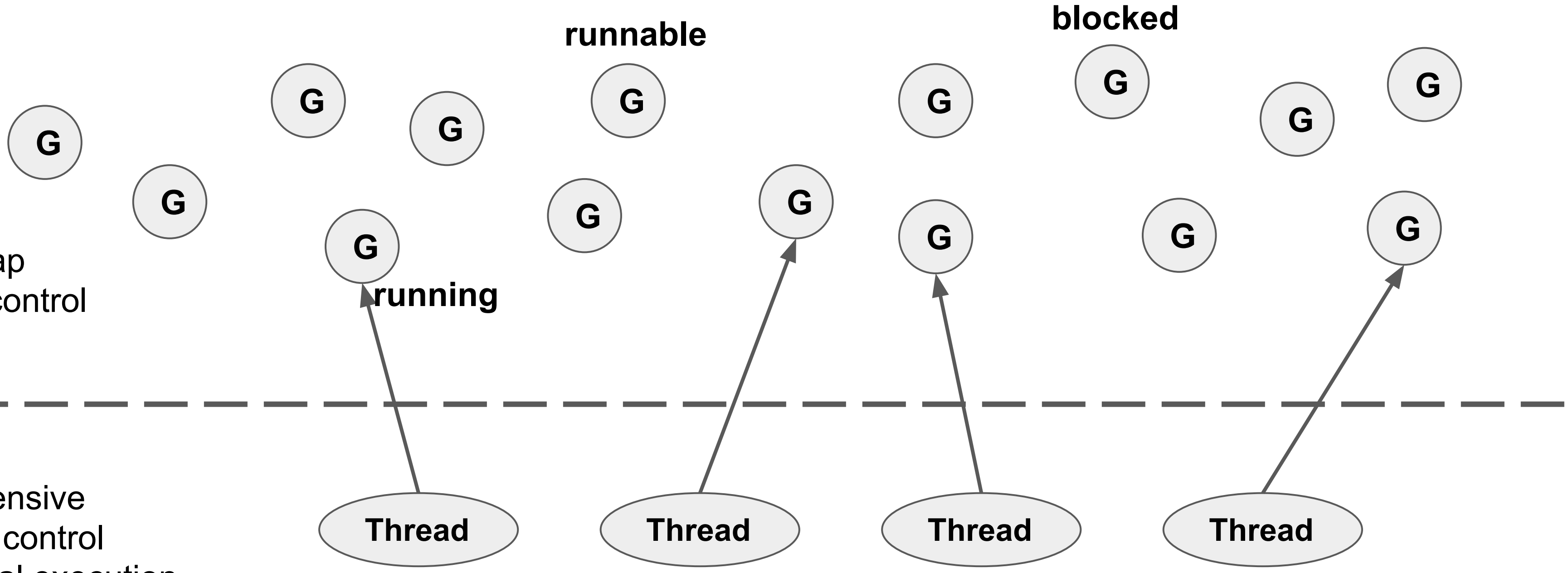
M:N Threading



- cheap
- full control

- expensive
- less control
- actual execution
- parallelism

Goroutine States



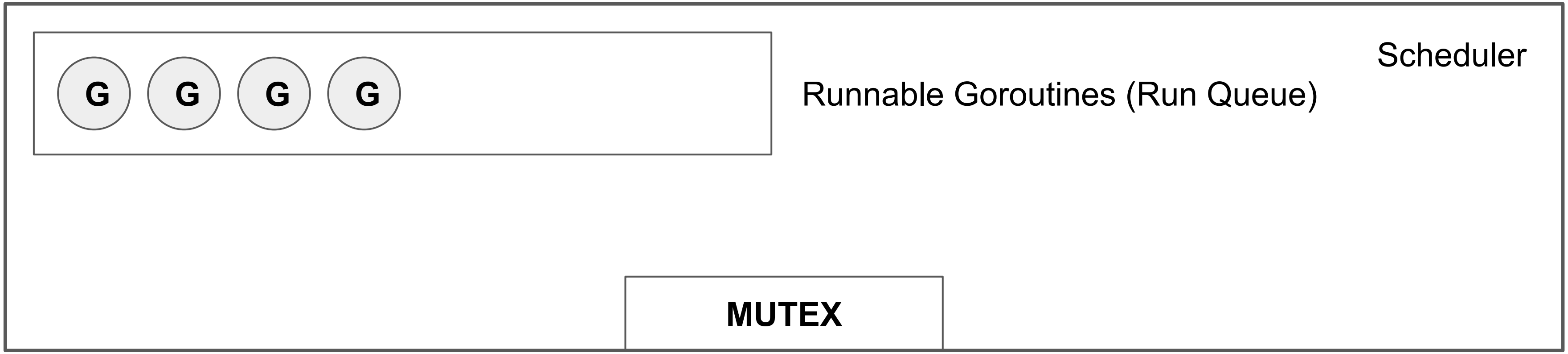
- cheap
- full control

N

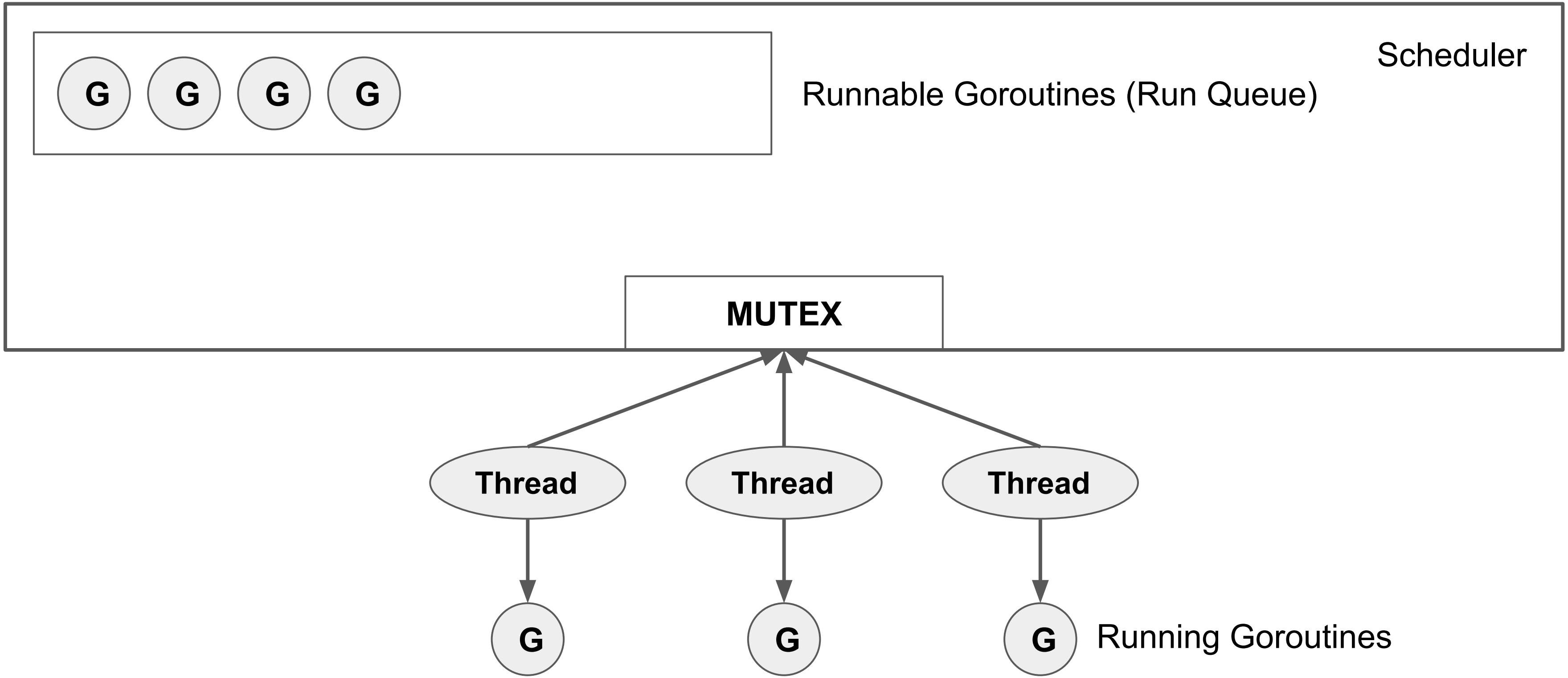
M

- expensive
- less control
- actual execution
- parallelism

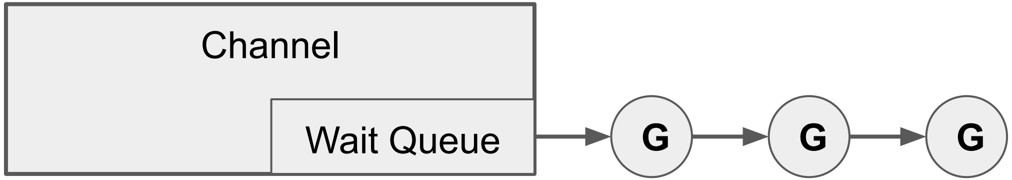
Simple M:N Scheduler



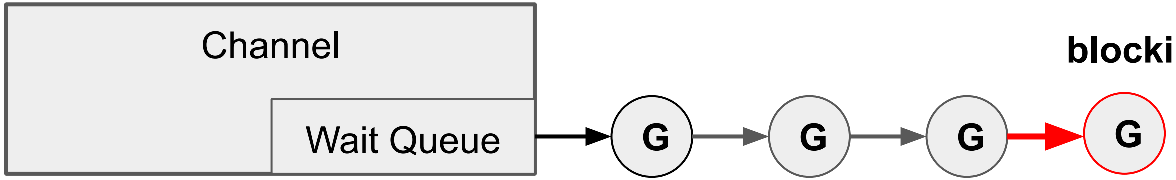
Simple M:N Scheduler



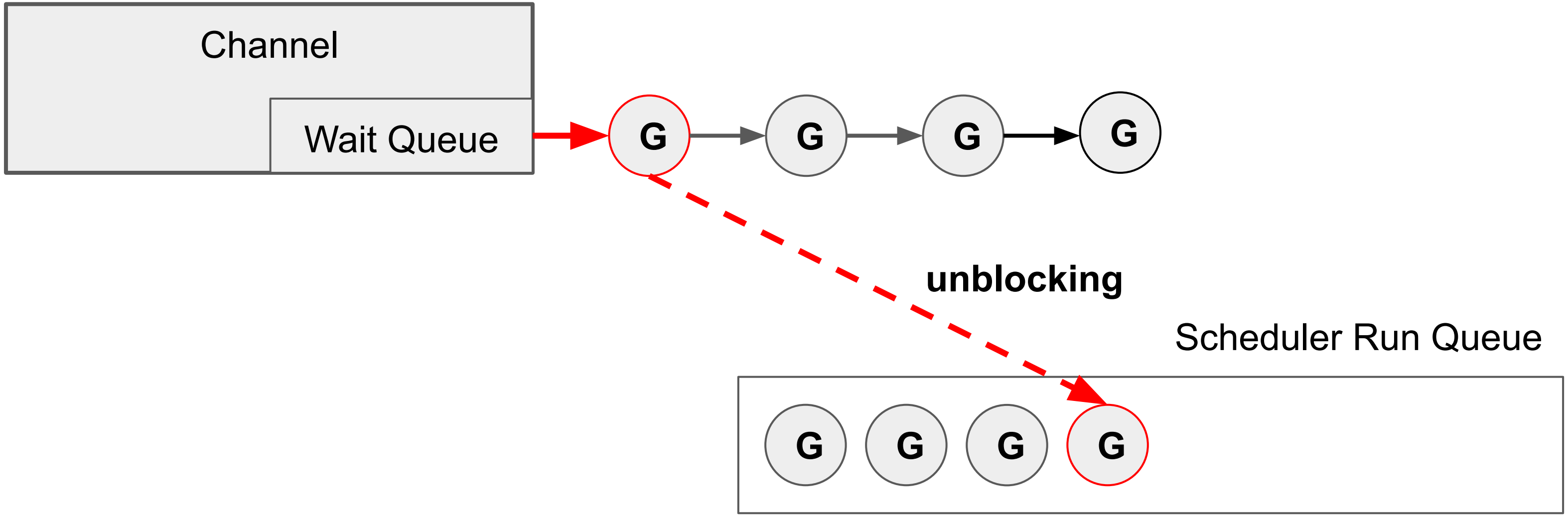
Blocked Goroutines



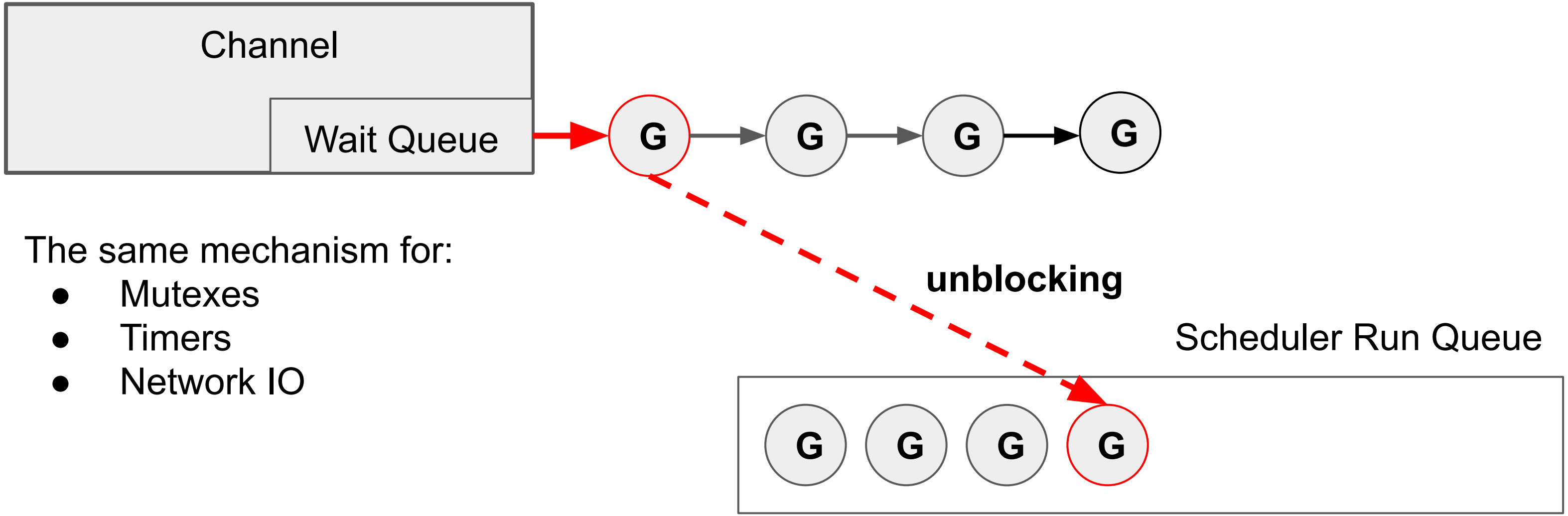
Blocked Goroutines



Blocked Goroutines



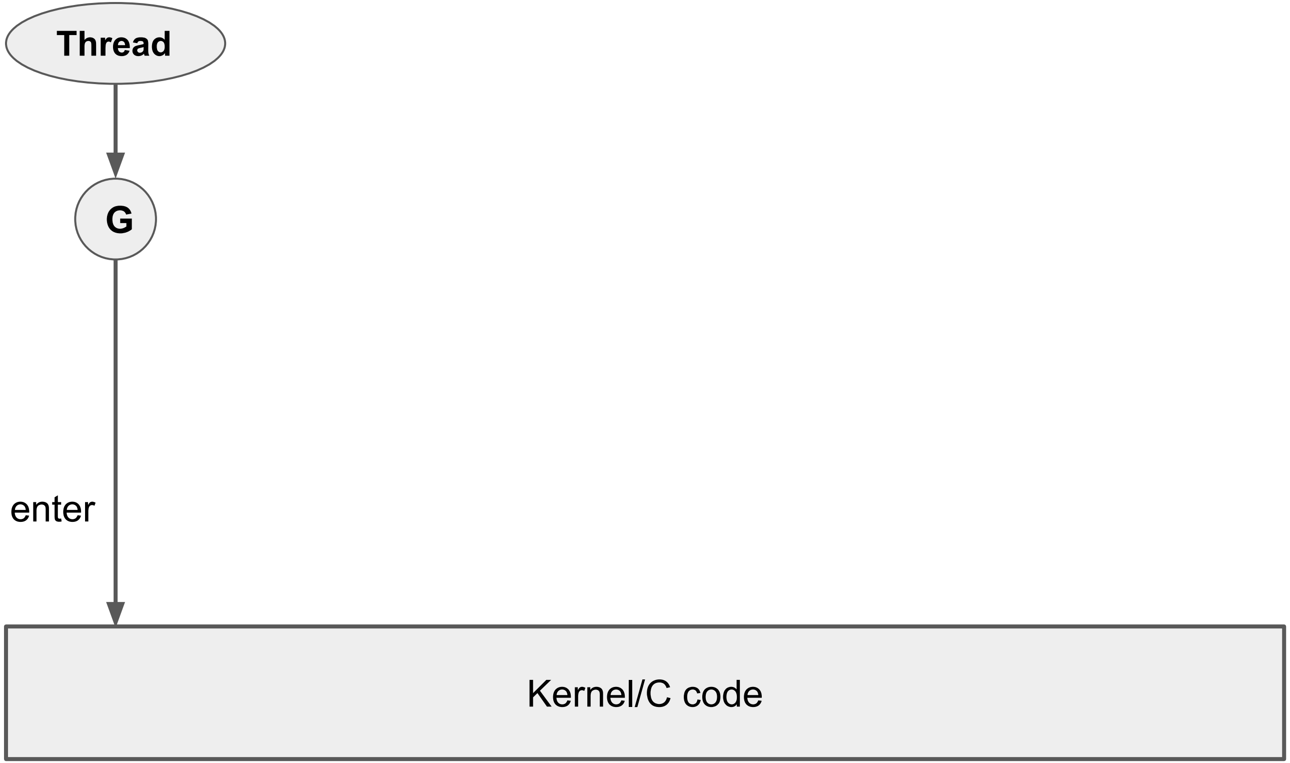
Blocked Goroutines



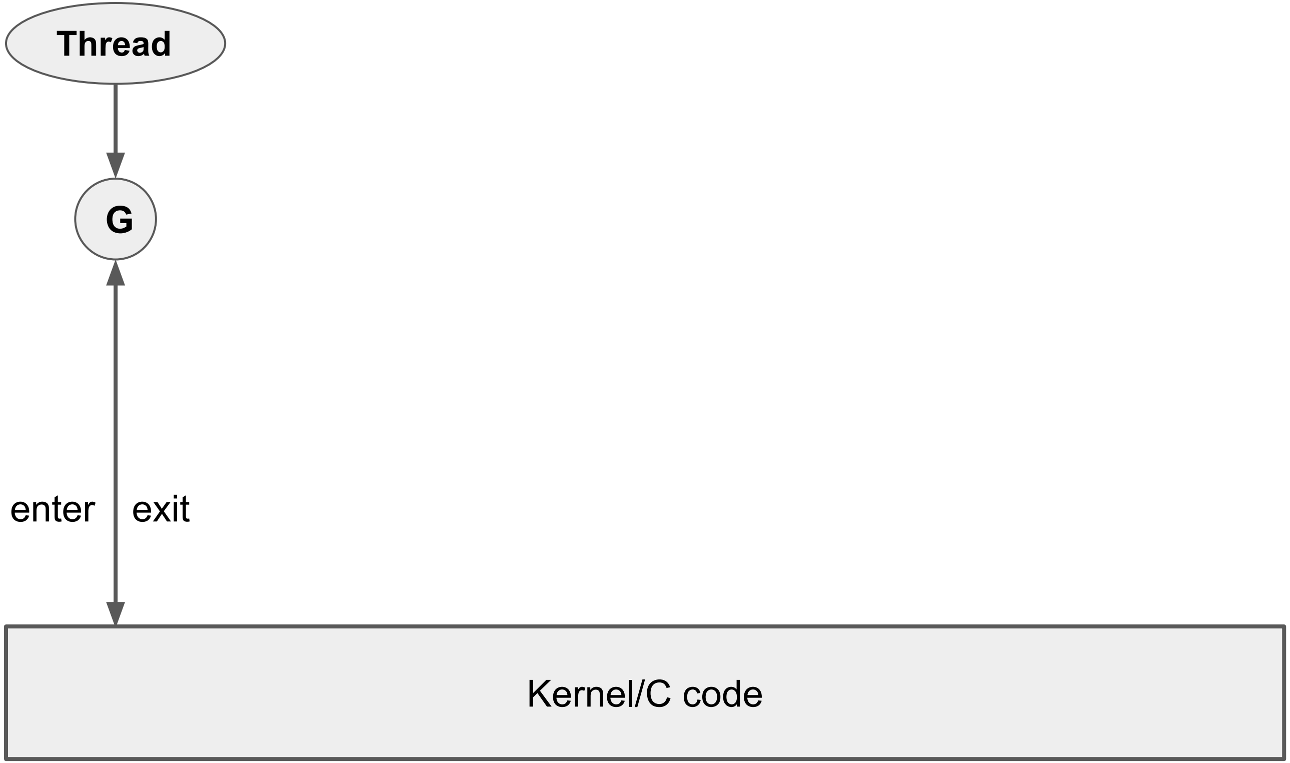
The same mechanism for:

- Mutexes
- Timers
- Network IO

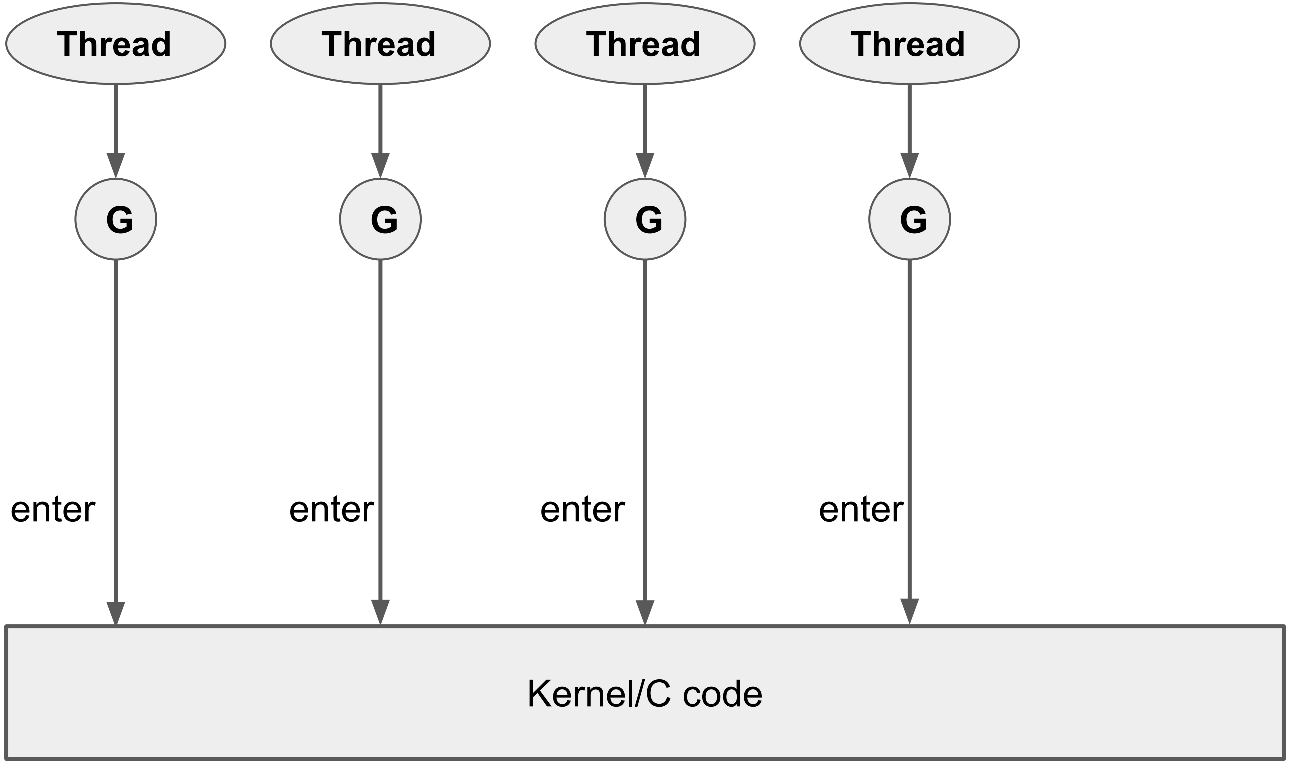
System Calls



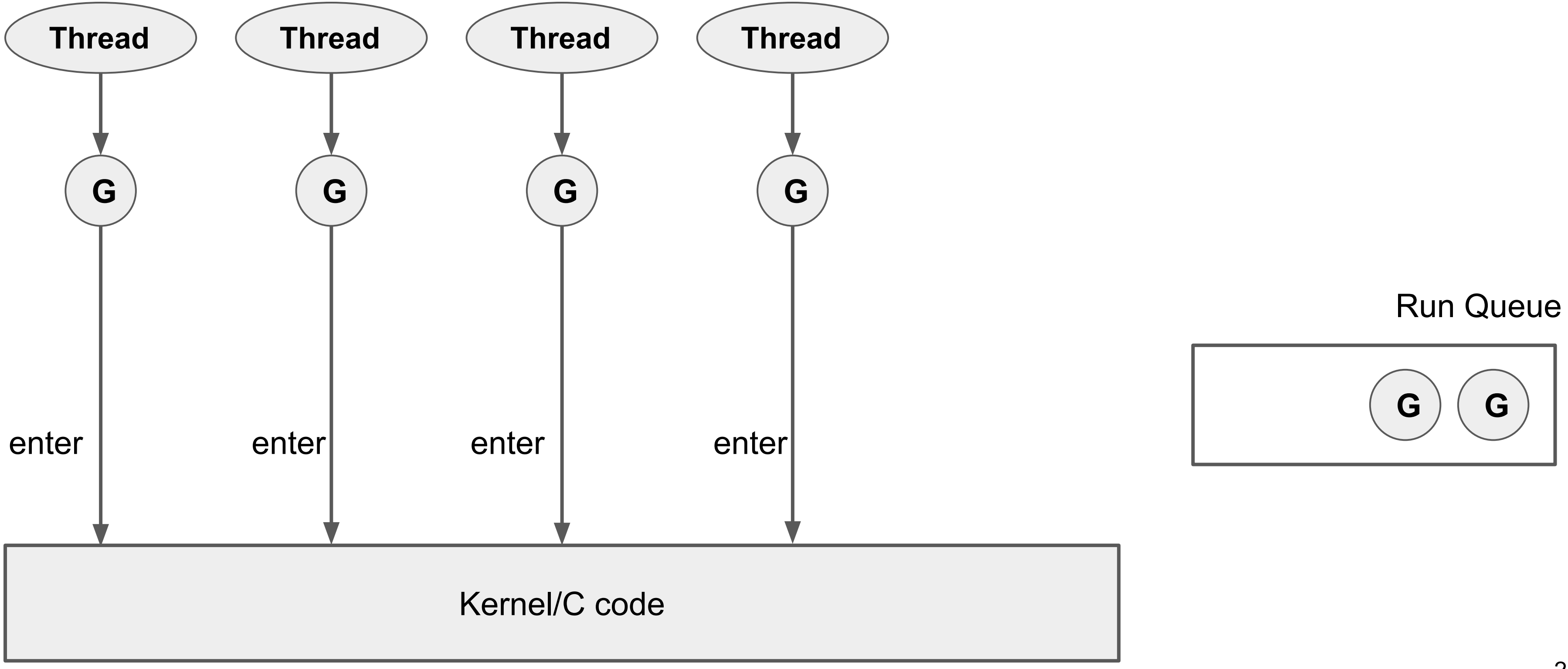
System Calls



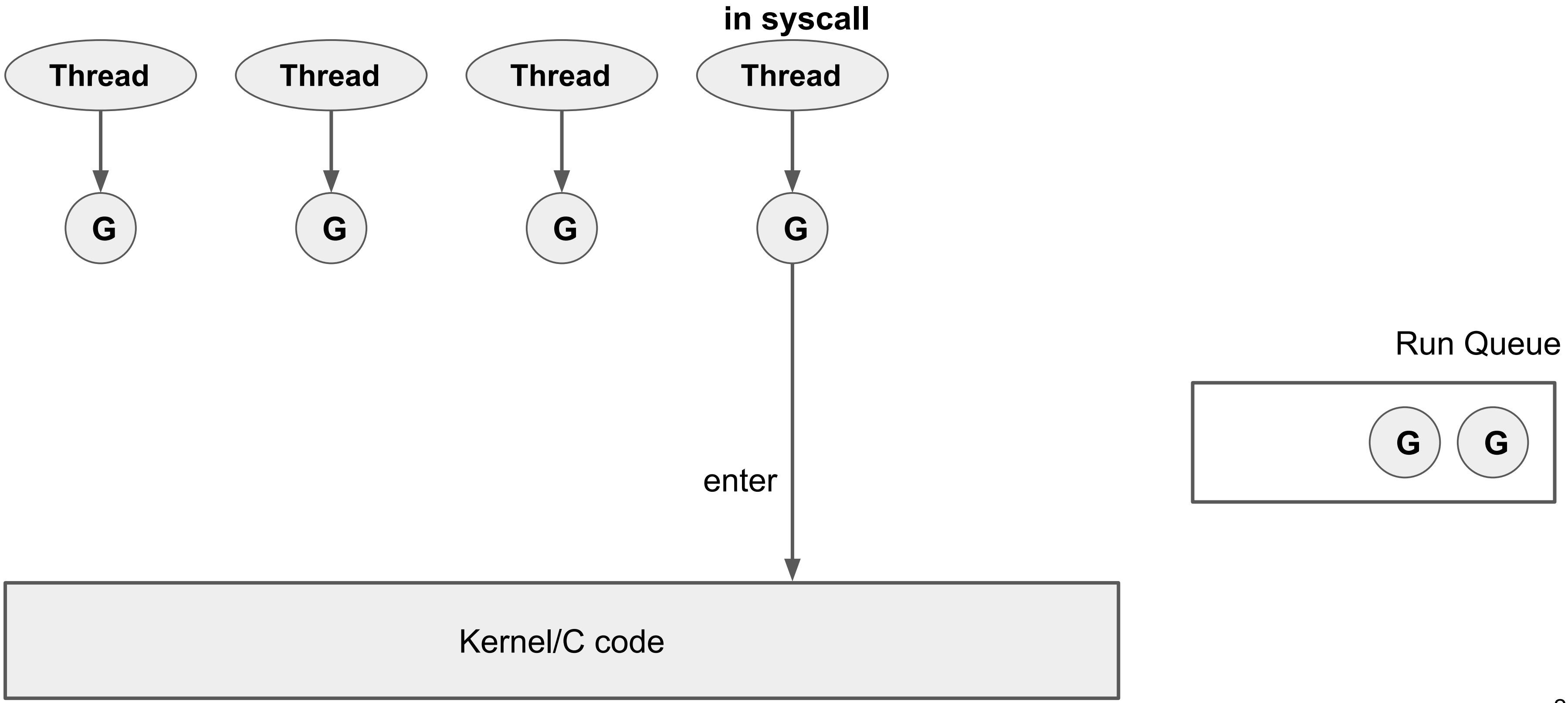
System Calls



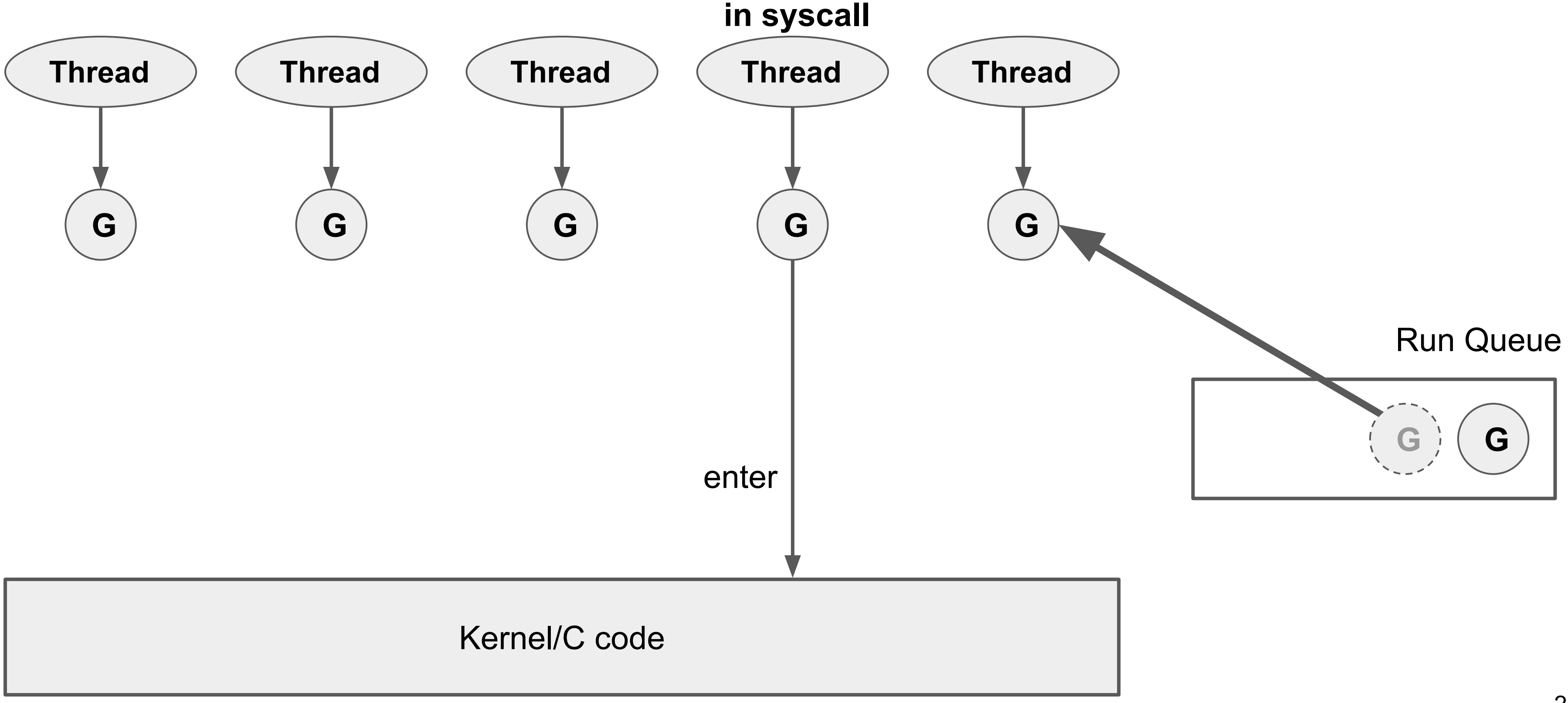
System Calls



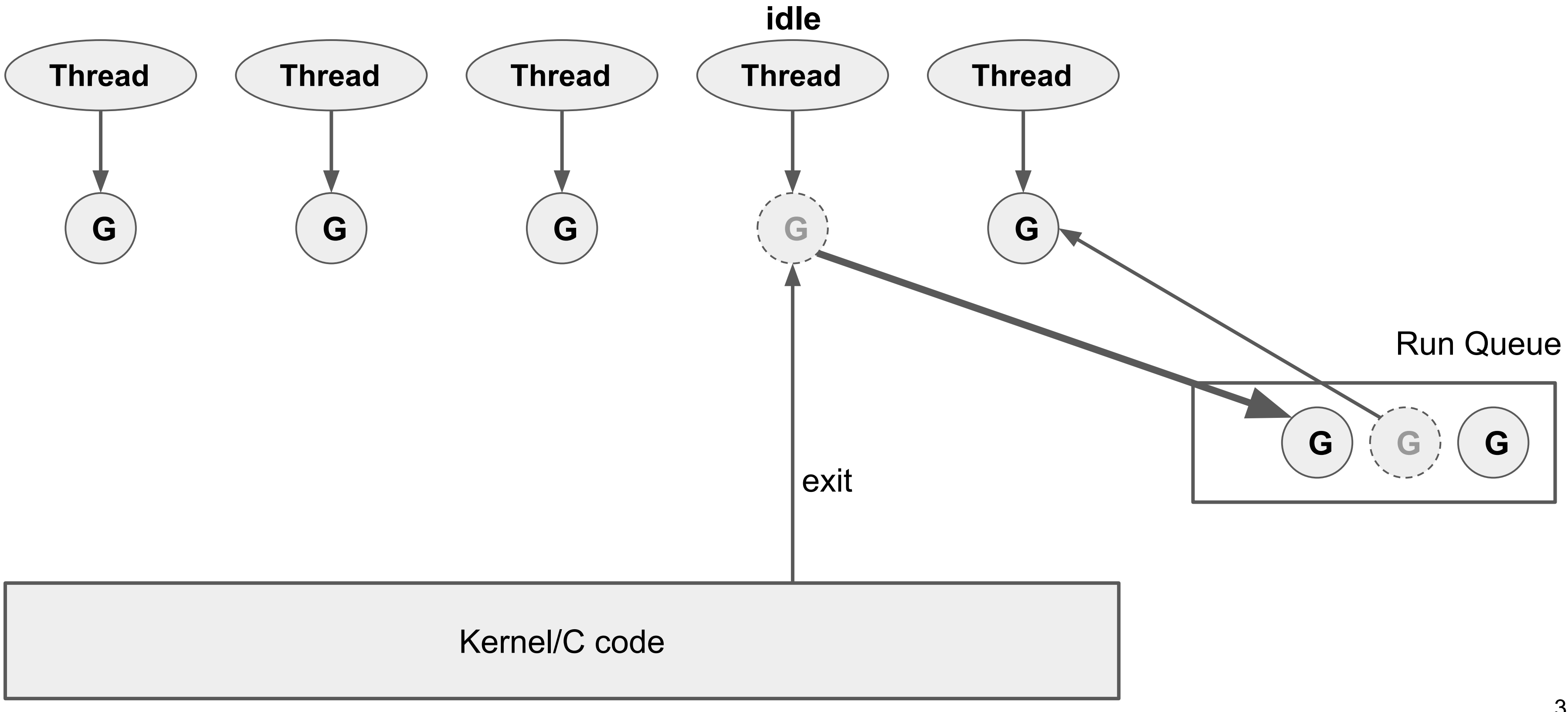
System Calls



System Calls



System Calls



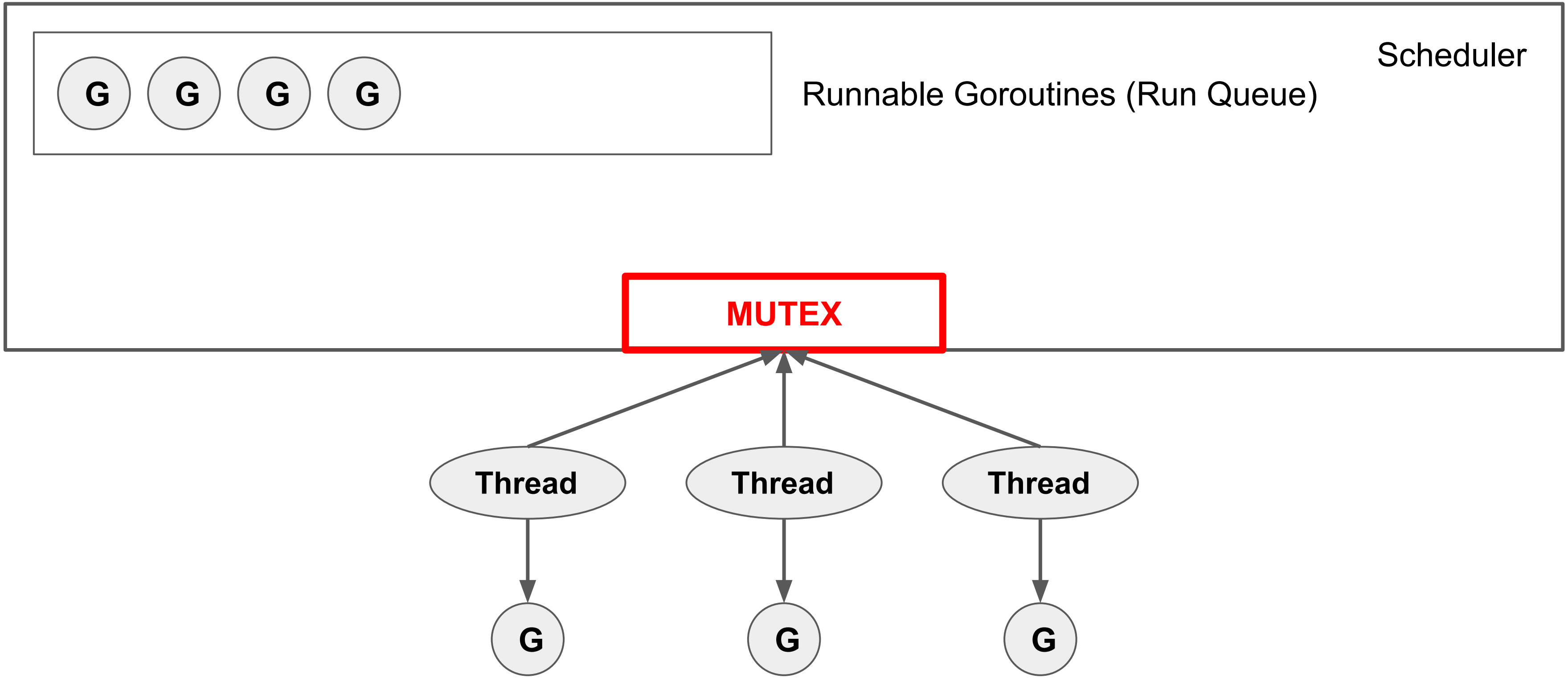
#Threads **>** #Cores

√ lightweight goroutines

√ handling of IO and syscalls

√ parallel

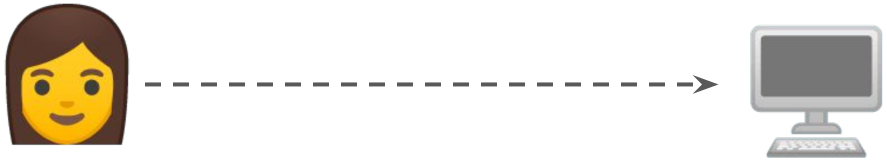
Not Scalable!

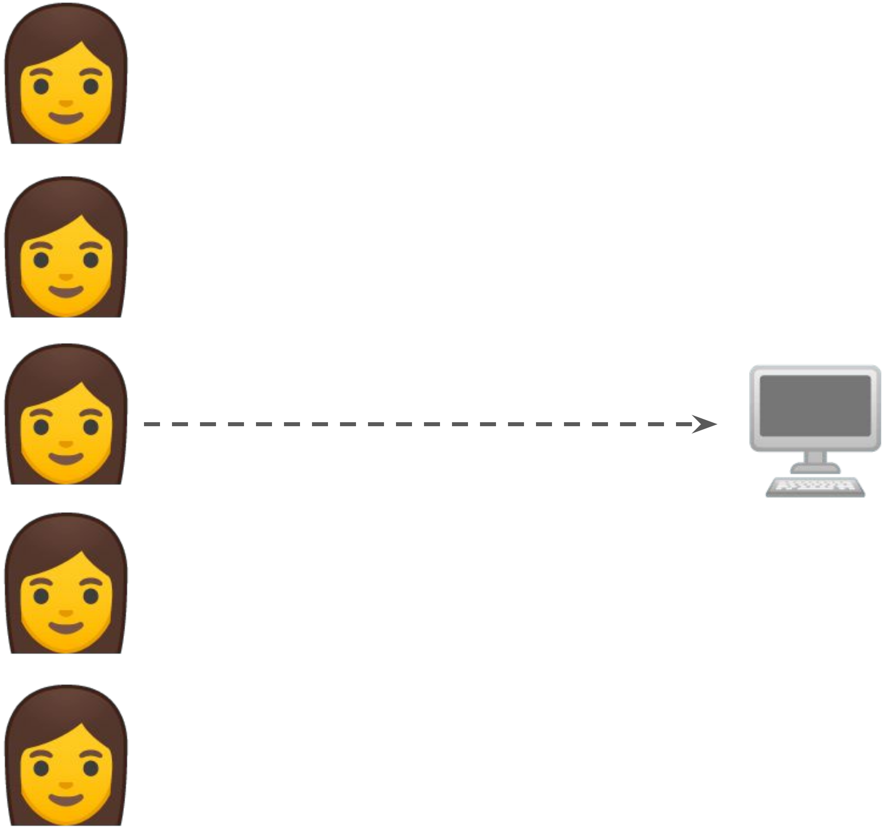


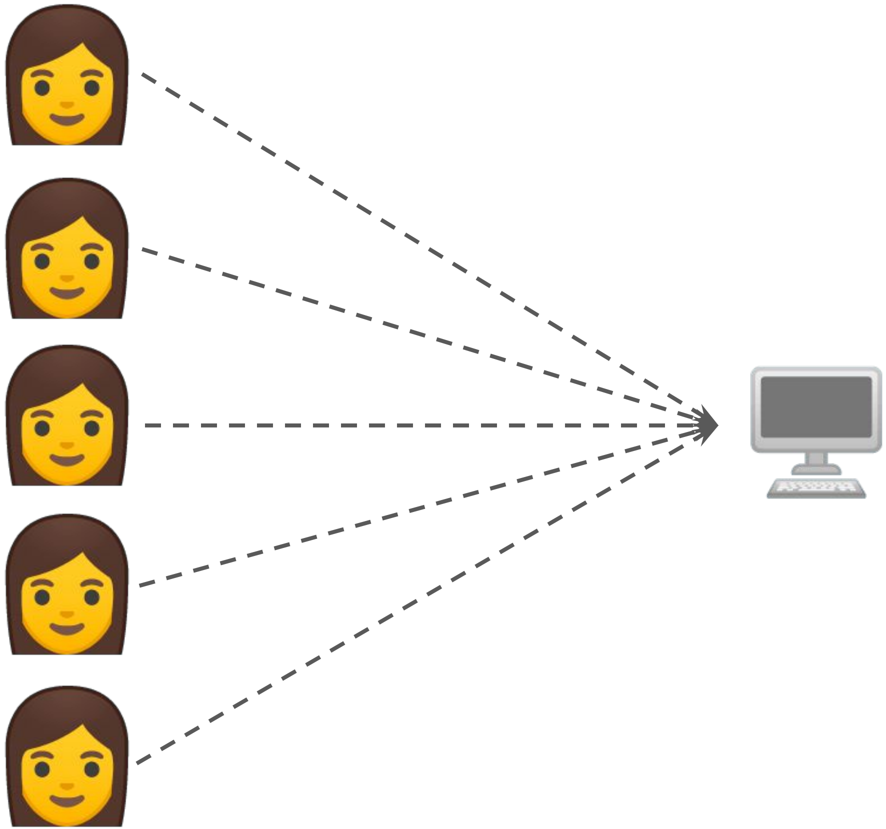
lock-free?

lock-free?

⛔





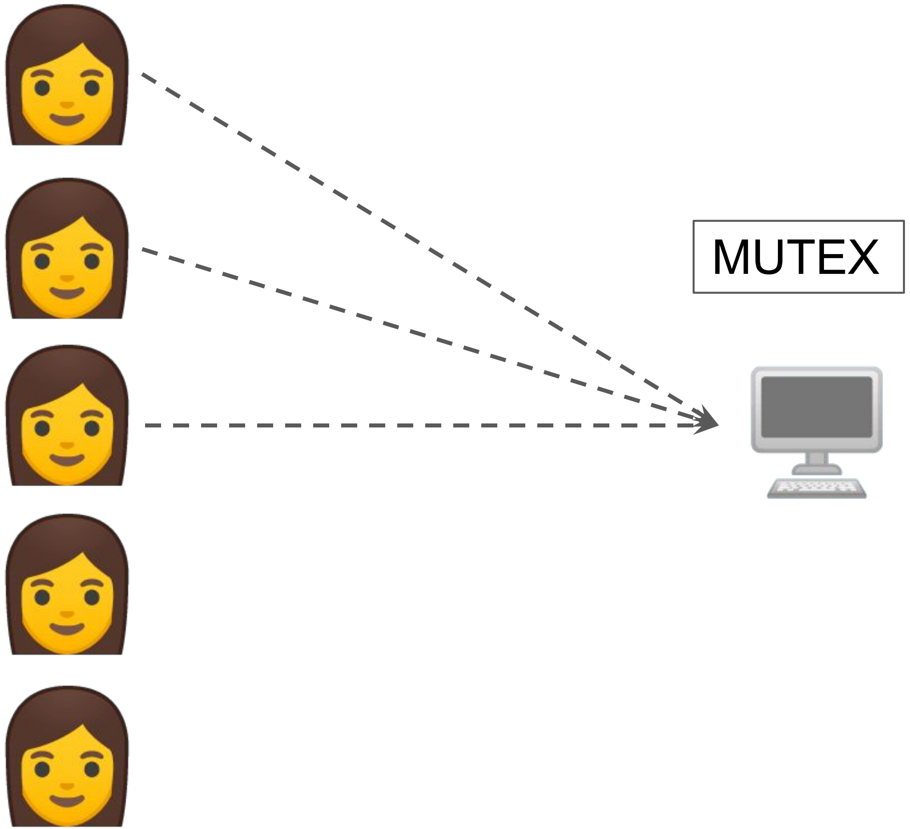


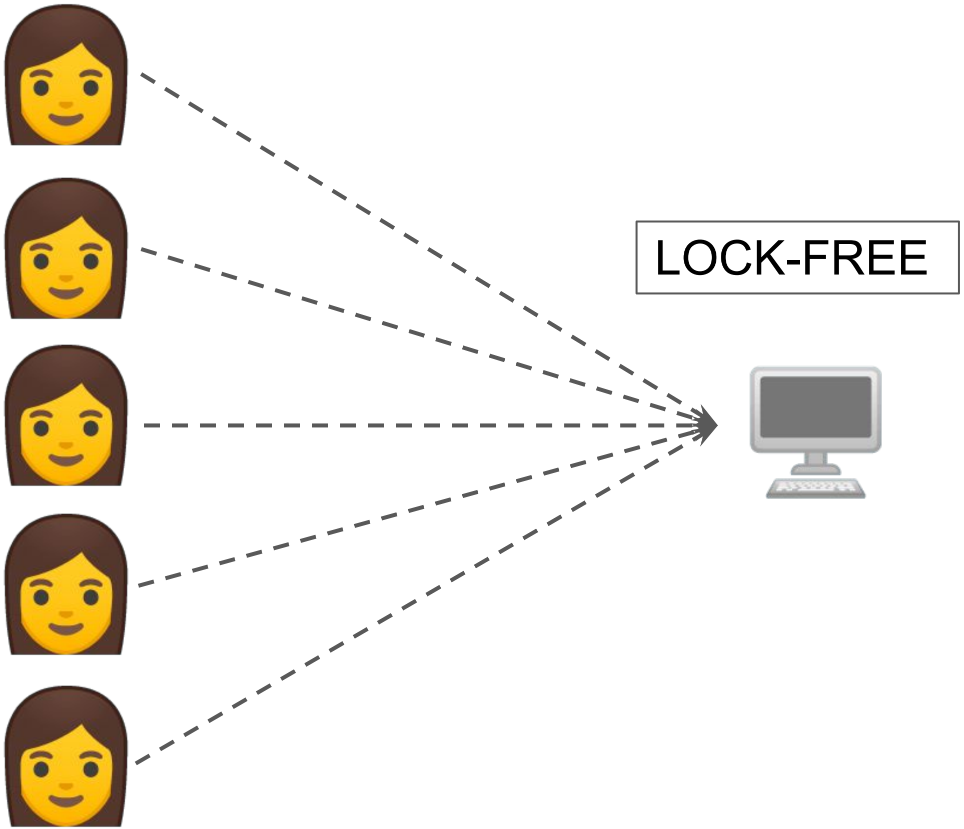
Shifts:

8:00 - 16:00

16:00 - 24:00

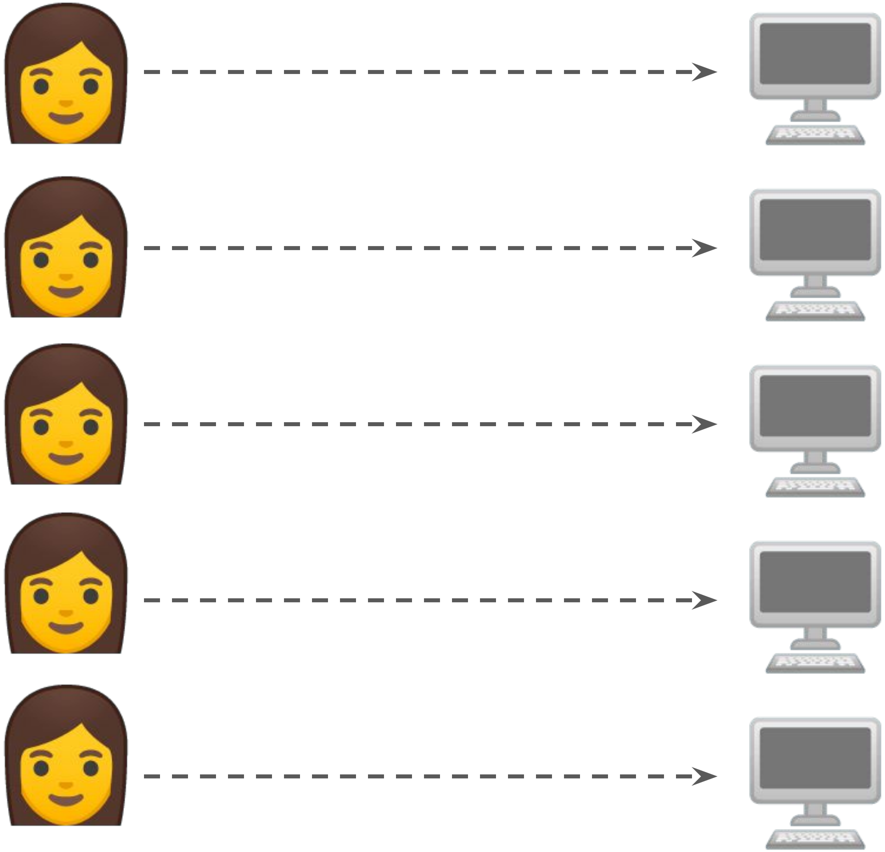
0:00 - 8:00



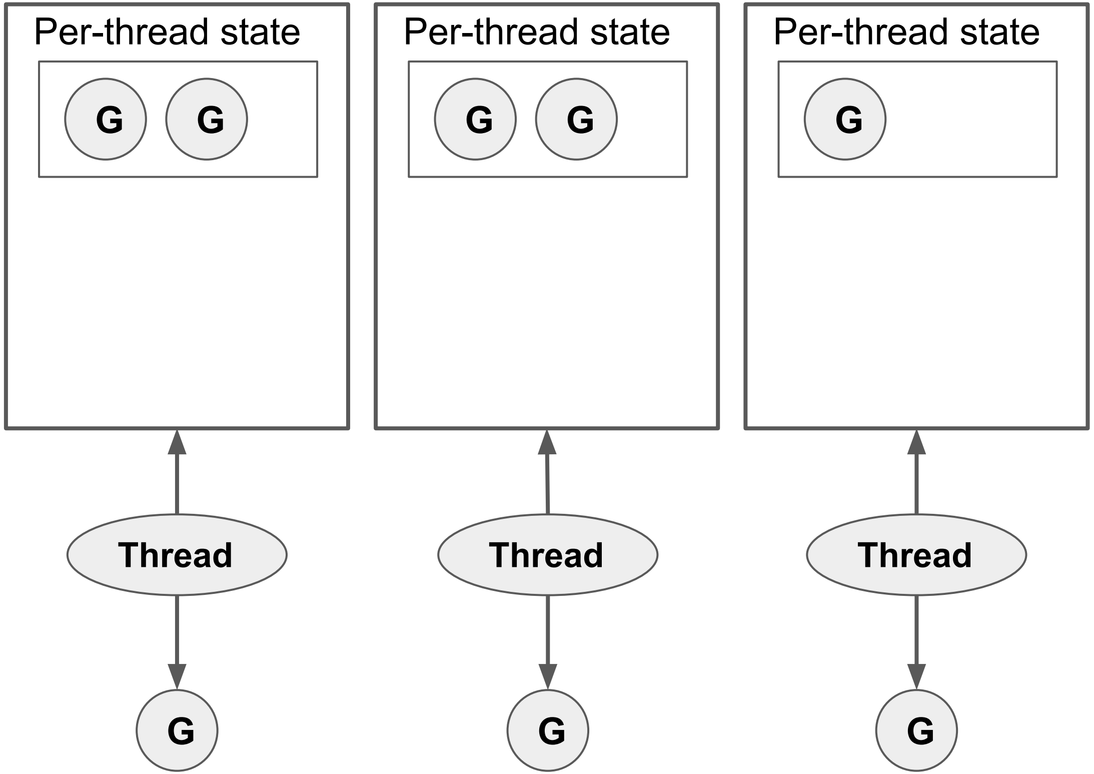


LOCK-FREE

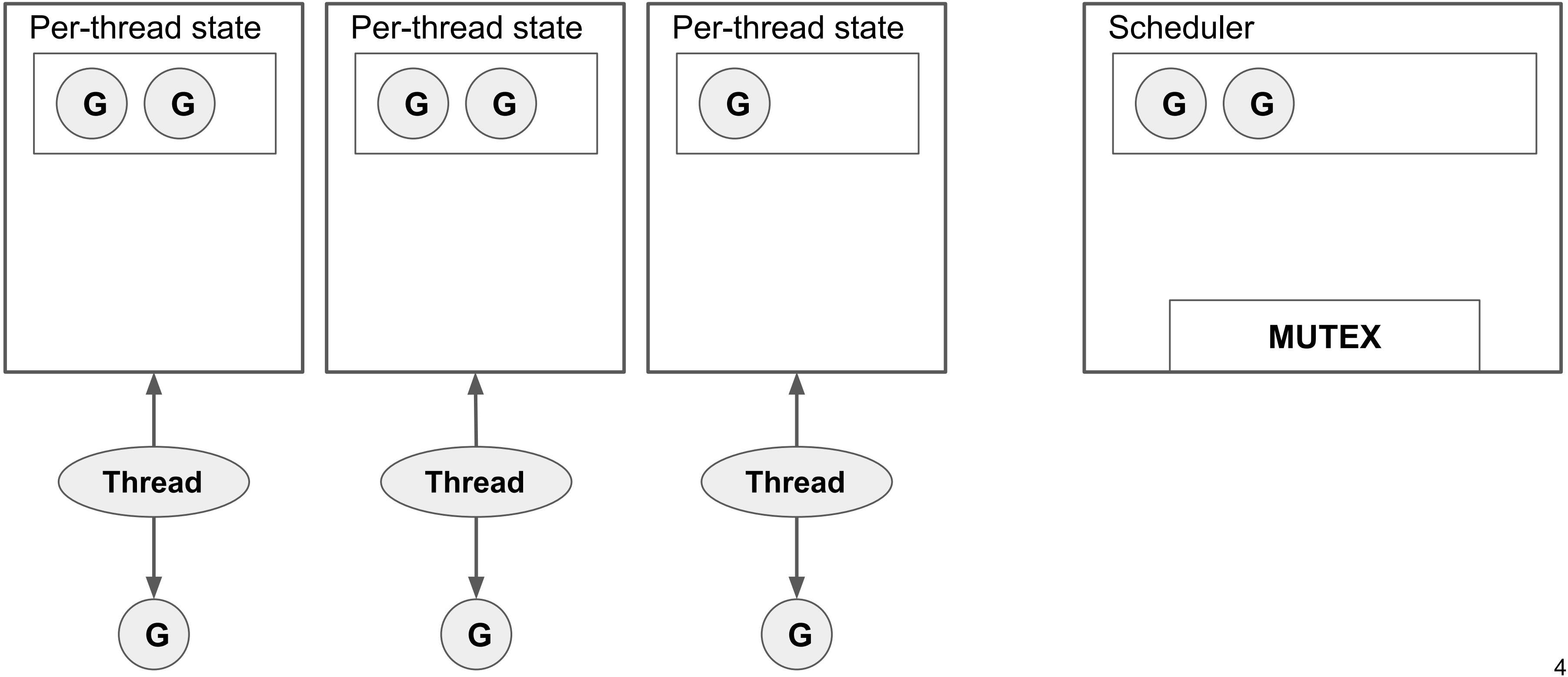
DISTRIBUTED



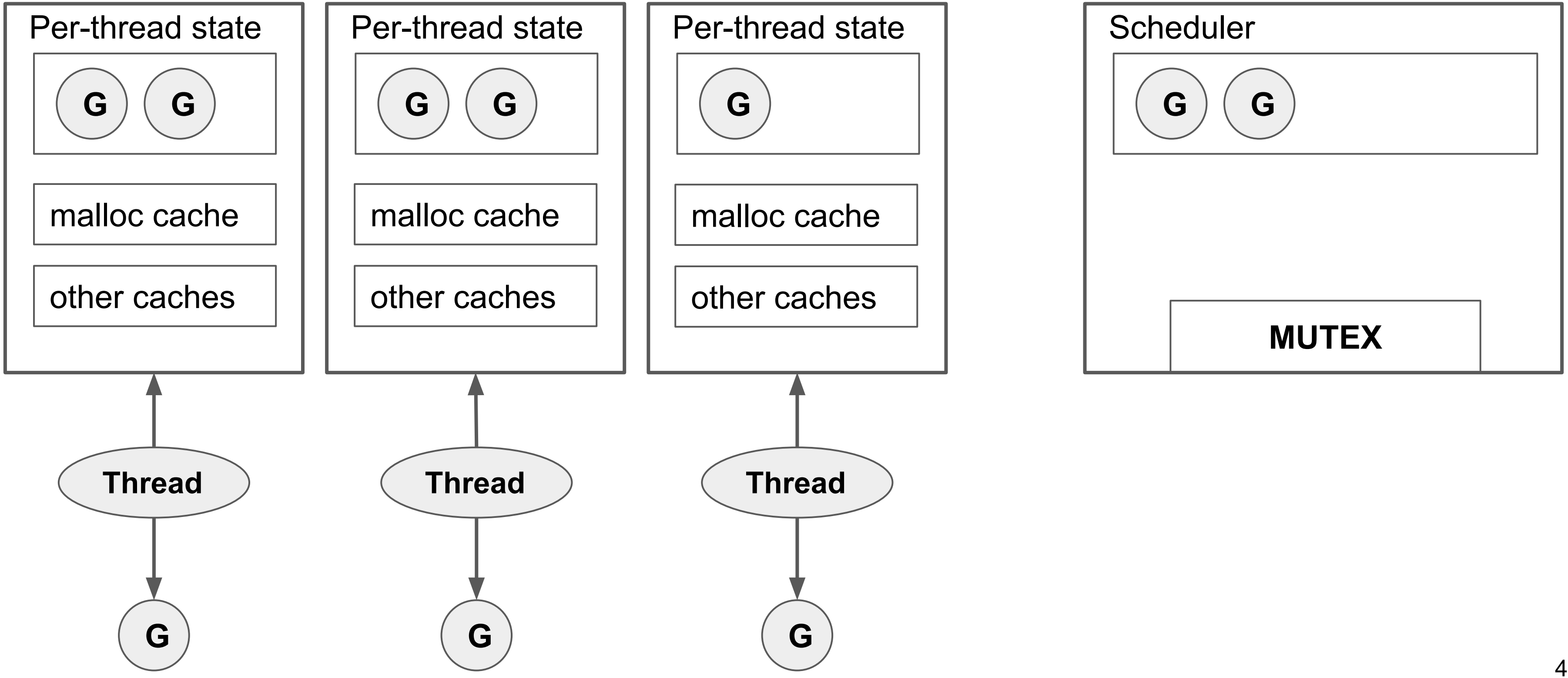
Distributed Scheduler



Distributed Scheduler



Distributed Scheduler

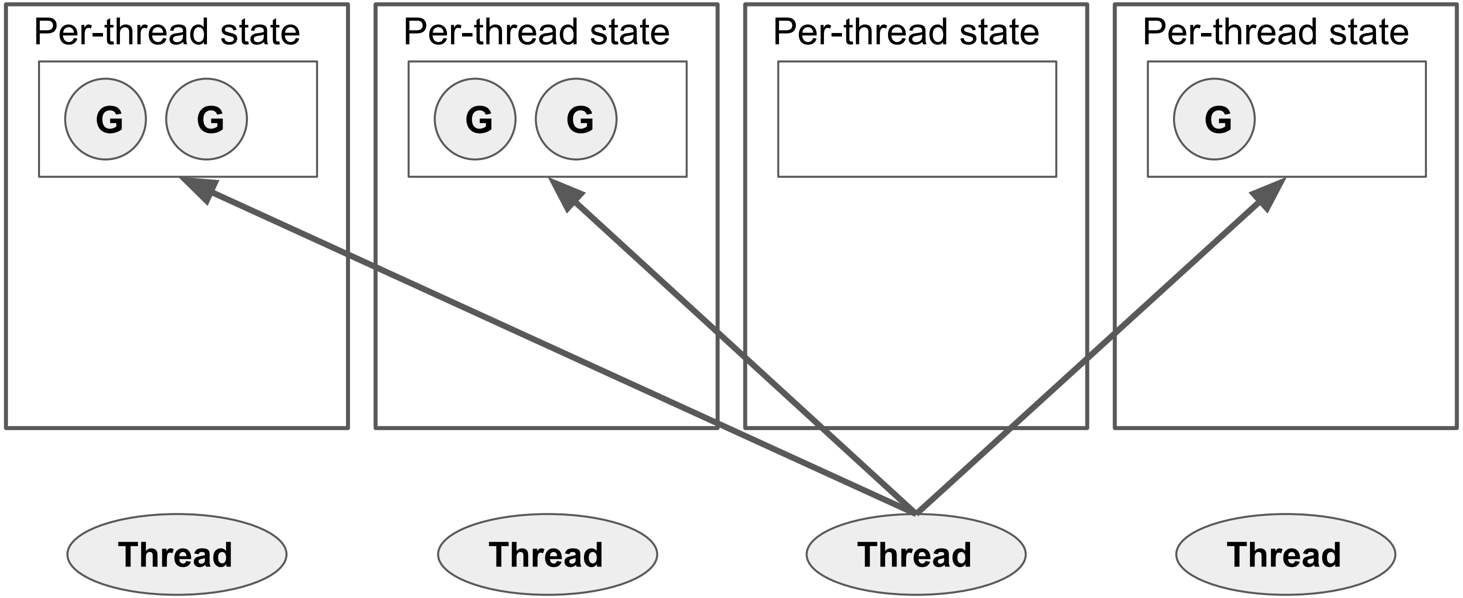


Poll Order

Main question: what is the **next goroutine** to run?

1. Local Run Queue
2. Global Run Queue
3. Network Poller
4. Work Stealing

Work Stealing



√ lightweight goroutines

√ handling of IO and syscalls

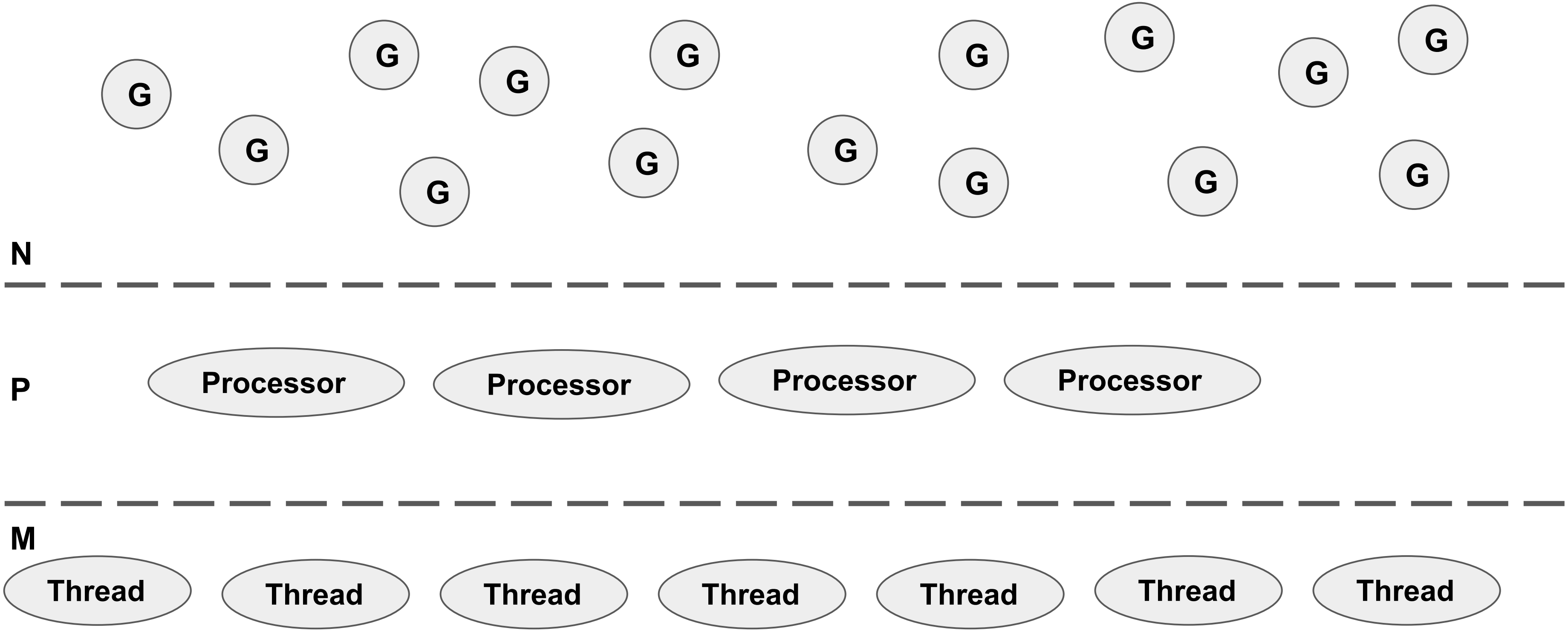
√ parallel

√ **scalable**

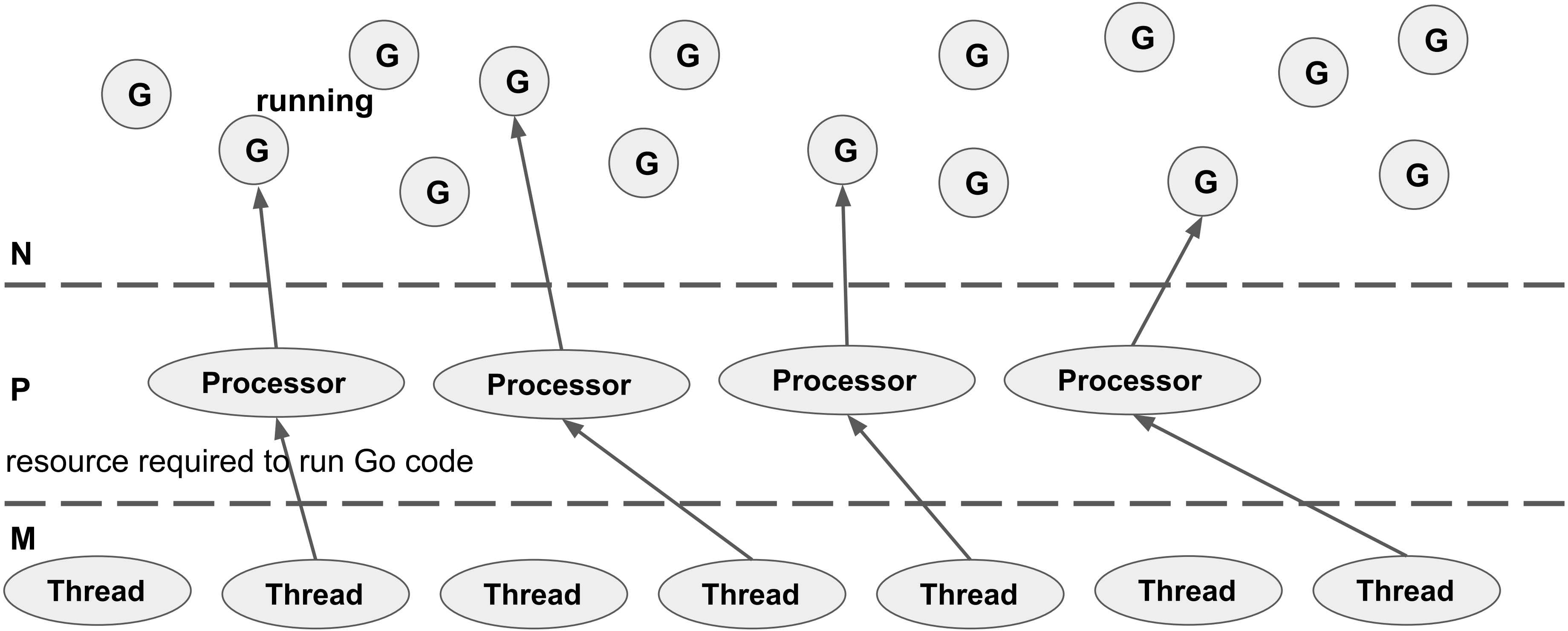
Threads in syscalls :(

(#threads > #cores)

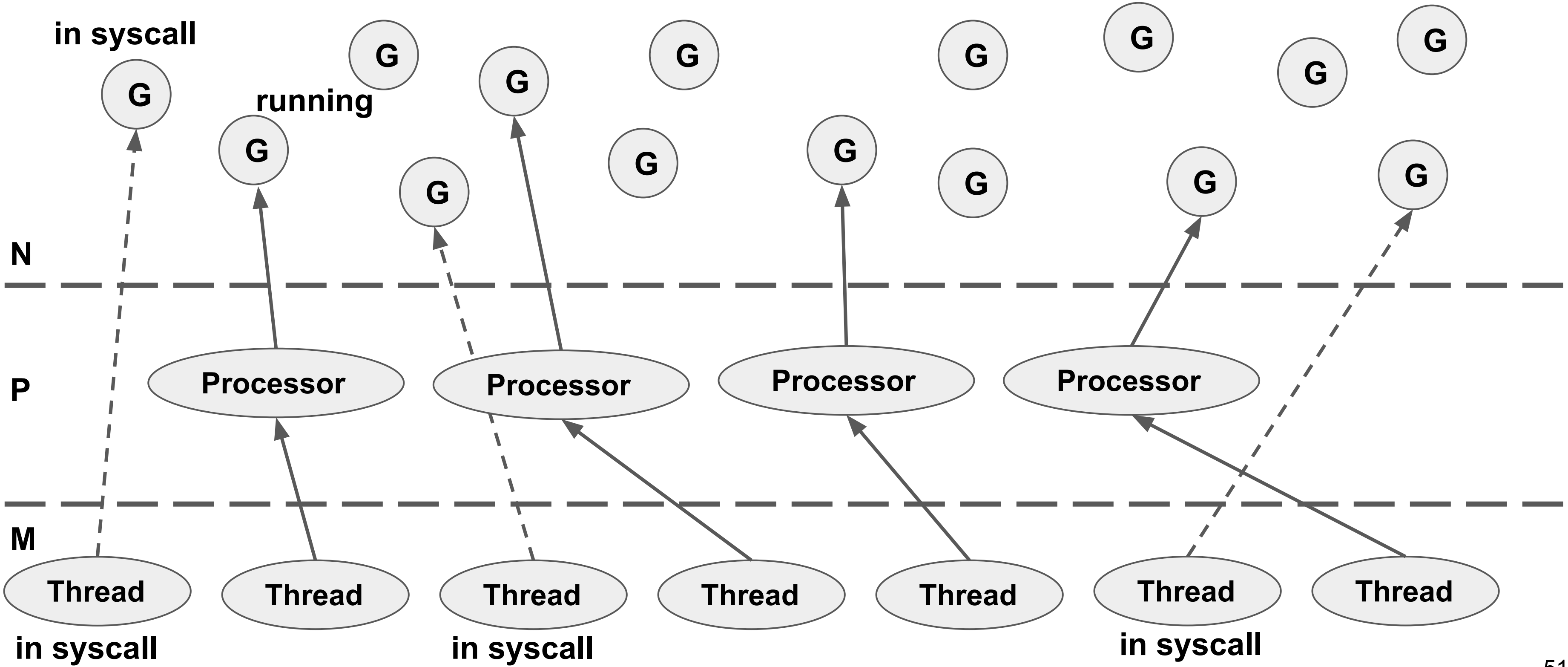
M:P:N Threading



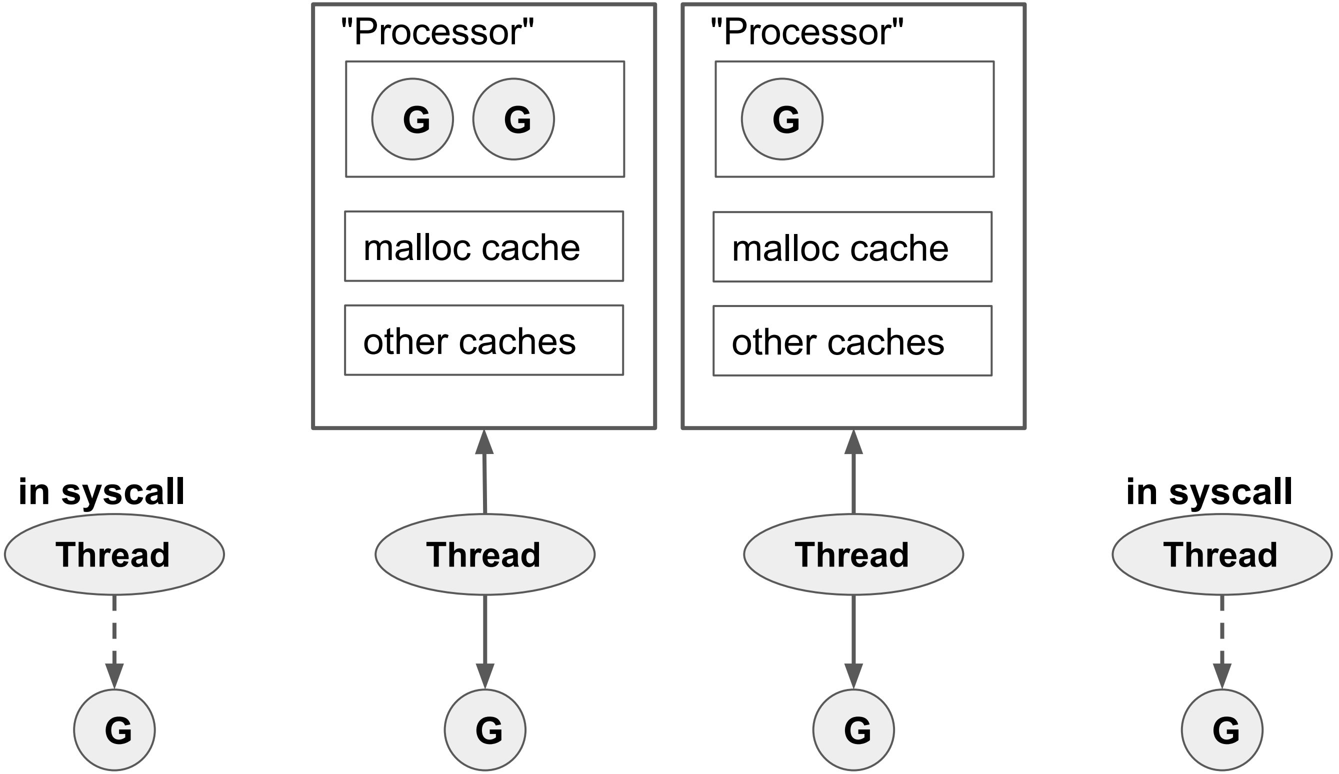
M:P:N Threading



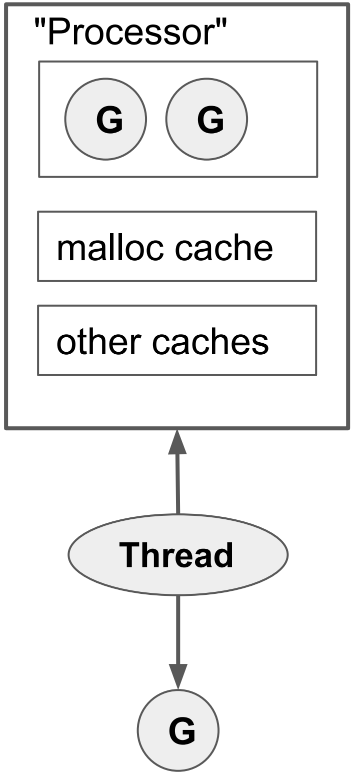
M:P:N Threading



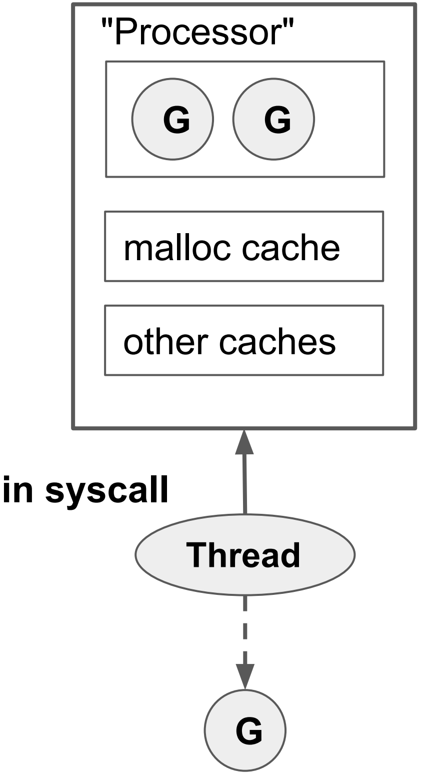
Distributed 3-Level Scheduler



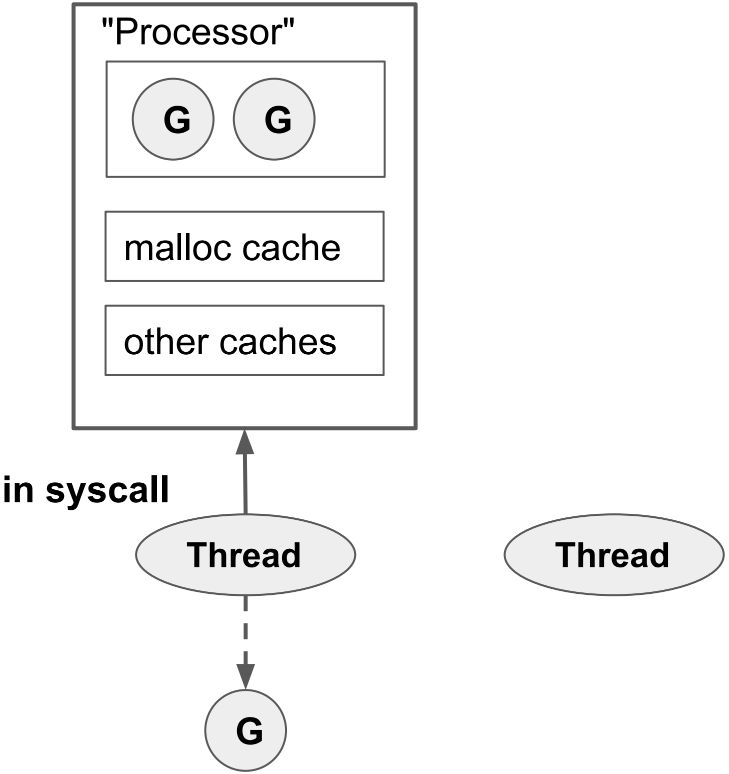
Syscall handling: Handoff



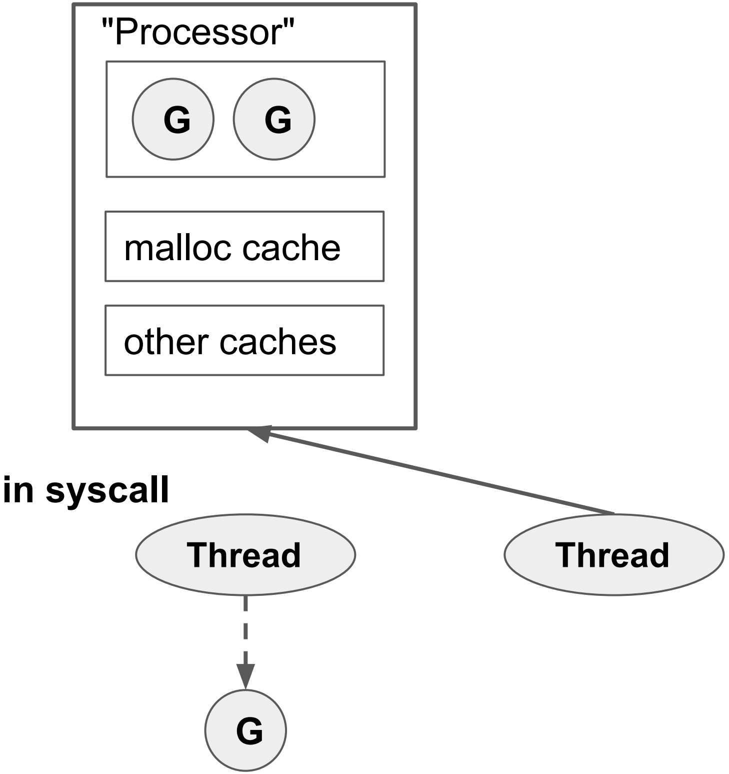
Syscall handling: Handoff



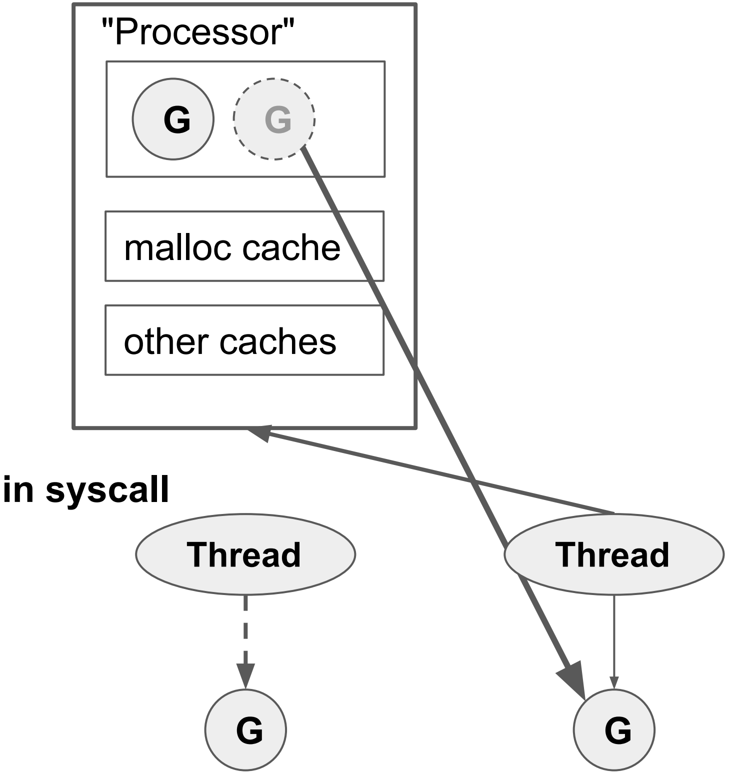
Syscall handling: Handoff



Syscall handling: Handoff



Syscall handling: Handoff



√ lightweight goroutines

√ handling of IO and syscalls

√ parallel

√ scalable

√ **efficient**

Fairness

Fairness

What: if a goroutine is runnable, it will run eventually.

Why:

- bad tail latencies
- livelocks
- pathological behaviors

Fairness

What: if a goroutine is runnable, it will run eventually.

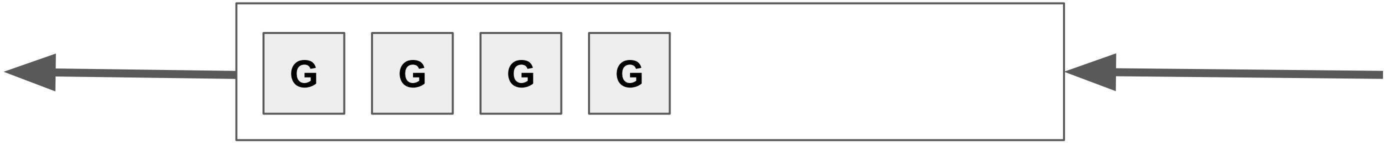
Why:

- bad tail latencies
- livelocks
- pathological behaviors

Fairness is like **Oxygen**

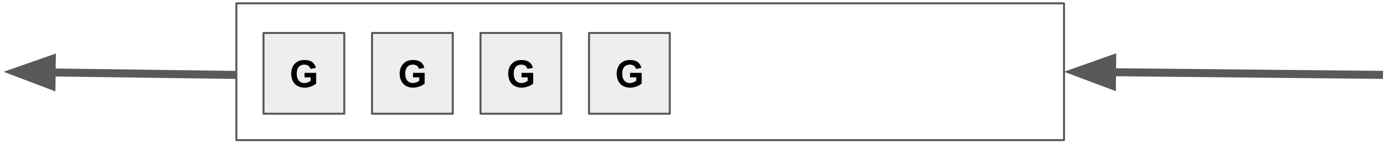
Fair Scheduling

Fair: FIFO Run Queue

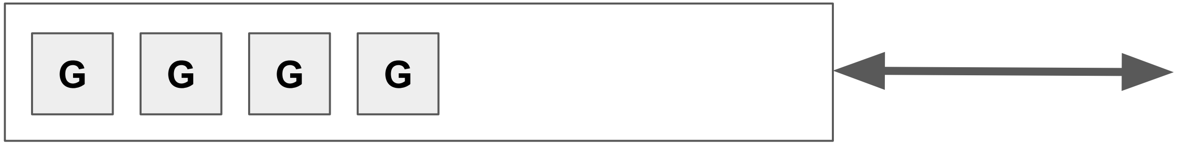


Fair Scheduling

Fair: FIFO Run Queue



Not Fair: LIFO Run Queue



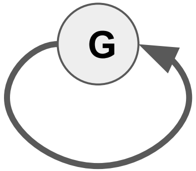
Fairness/Performance Tradeoff

- Single Run Queue does not scale
- FIFO bad for locality

Want a **minimal** amount of fairness!

Infinite Loops

in infinite loop

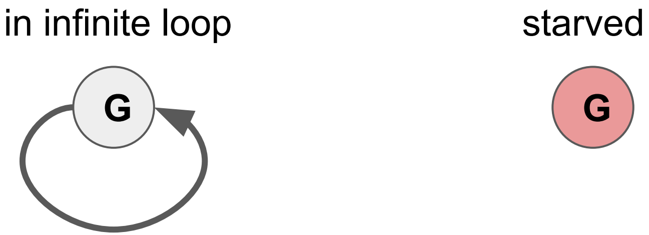


Infinite Loops

in infinite loop

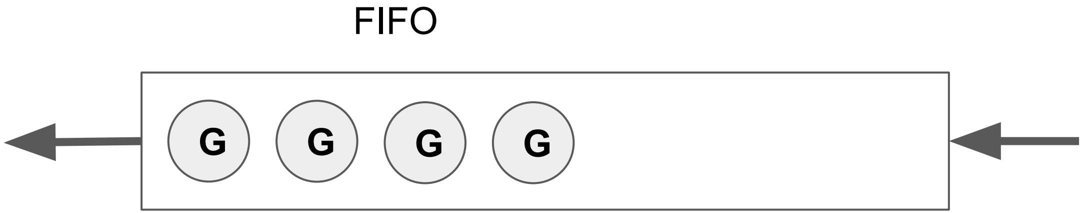
starved

Infinite Loops

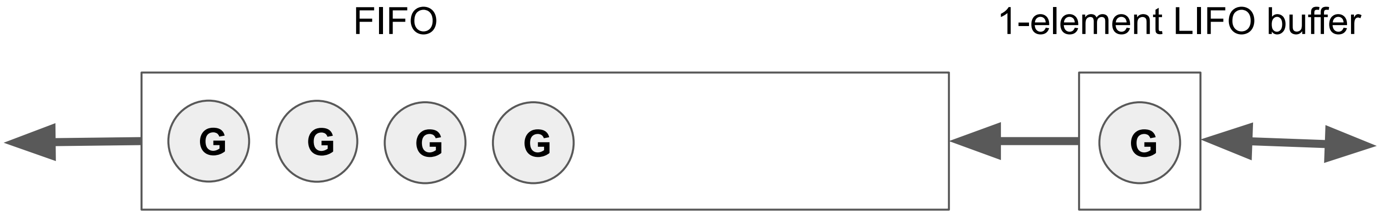


Solution: **preemption** (~10ms)

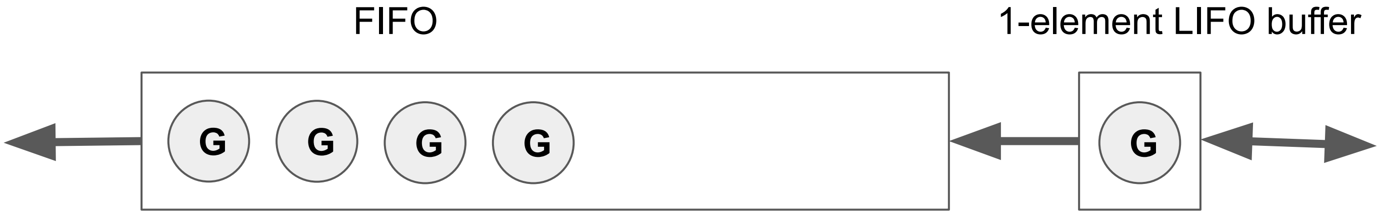
Local Run Queue



Local Run Queue

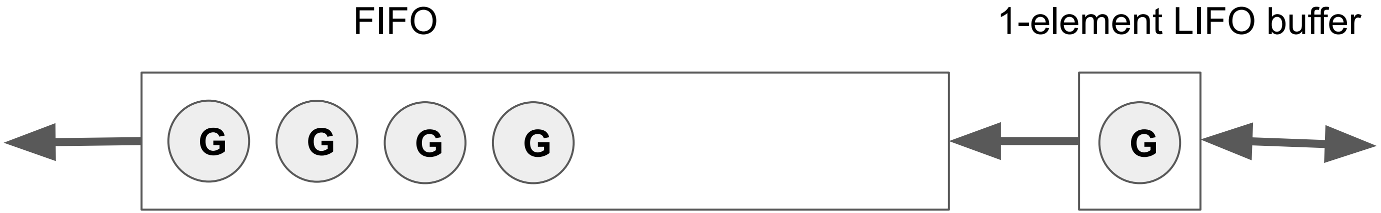


Local Run Queue



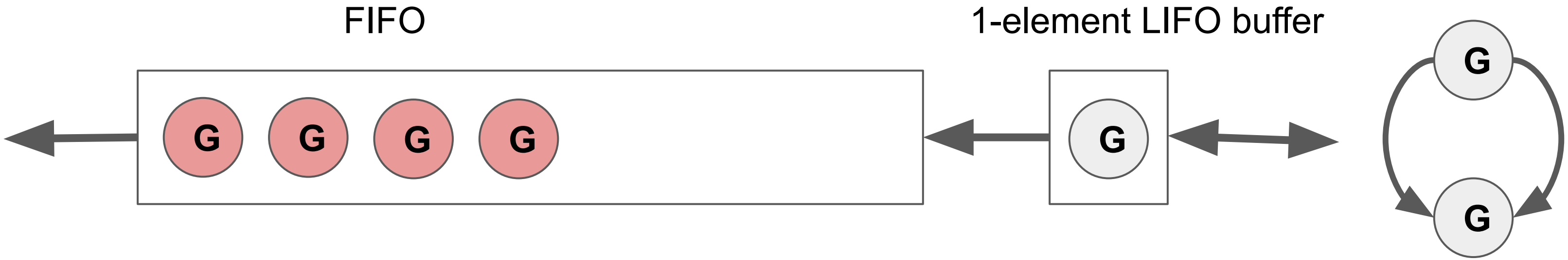
- better locality

Local Run Queue

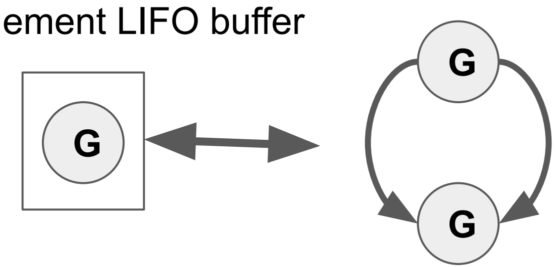


- better locality
- restricts stealing (3us)

Local Run Queue Starvation

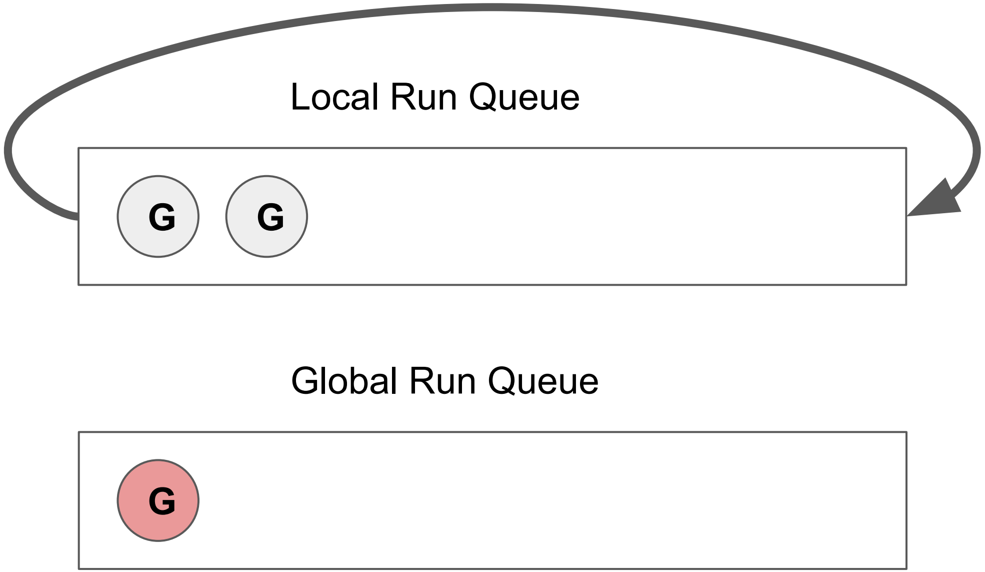


Time Slice Inheritance


1-element LIFO buffer

Solution: **inherit time slice** -> looks like infinite loop -> preemption (~10ms)

Global Run Queue Starvation



Global Run Queue Starvation

```
g = pollLocalRunQueue()
if g != nil {
    return g
}
return pollGlobalRunQueue()
```

Global Run Queue Starvation

```go
schedTick++
if schedTick%61 == 0 {
    g = pollGlobalRunQueue()
    if g != nil {
        return g
    }
}
g = pollLocalRunQueue()
if g != nil {
    return g
}
return pollGlobalRunQueue()
```

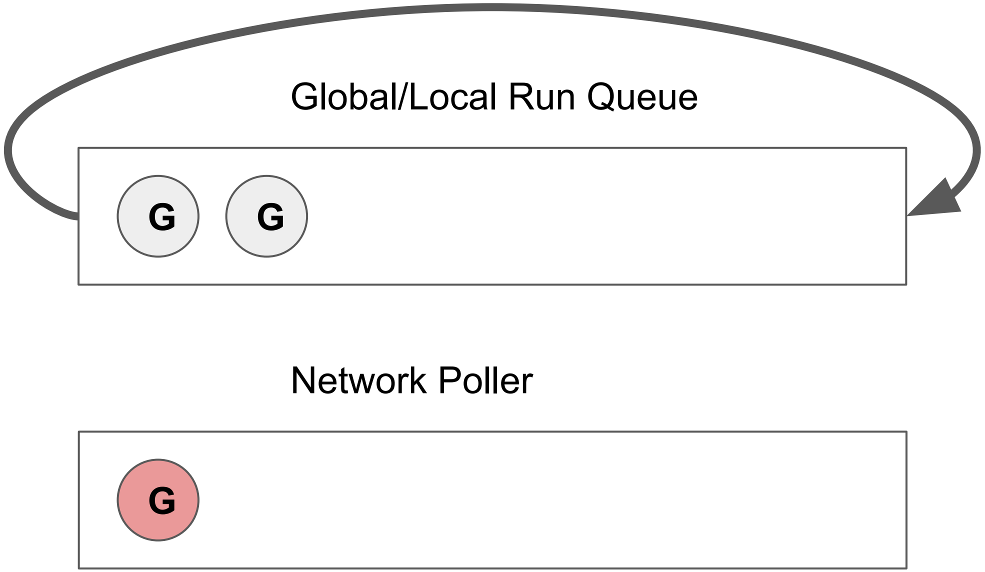
Why 61?

It is not even 42! ¯_(ツ)_/¯

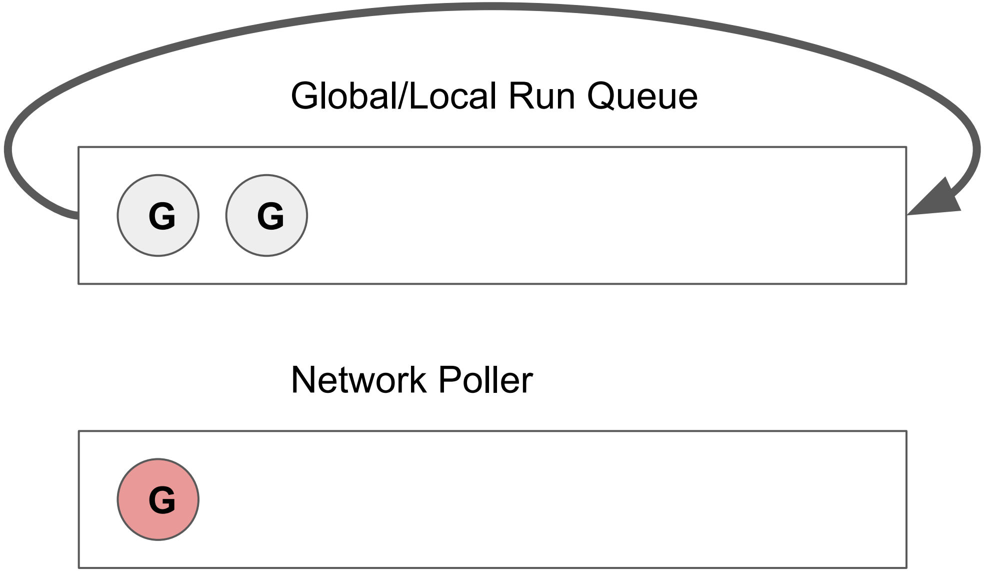
Want something:

- not too small
- not too large
- prime to break any patterns

Network Poller Starvation



Network Poller Starvation



Solution: **background thread** poll network occasionally

Fairness Hierarchy

Goroutine - preemption

Local Run Queue - time slice inheritance

Global Run Queue - check once in a while

Network Poller - background thread

= minimal fairness at minimal cost

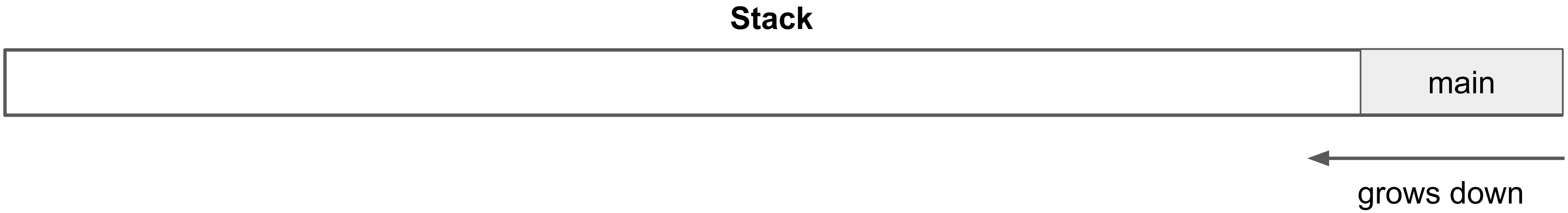
Stacks

Function Frame

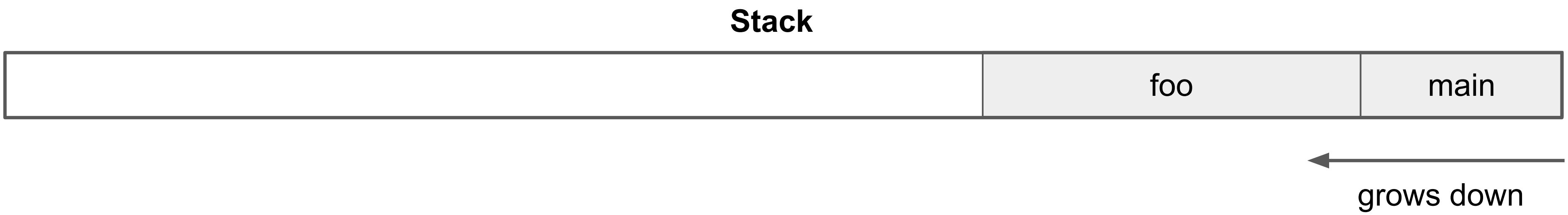
```
void foo()
{
    ...
    int x = 42;
    ...
    return;
}
```

- local variables
- return address
- previous frame pointer

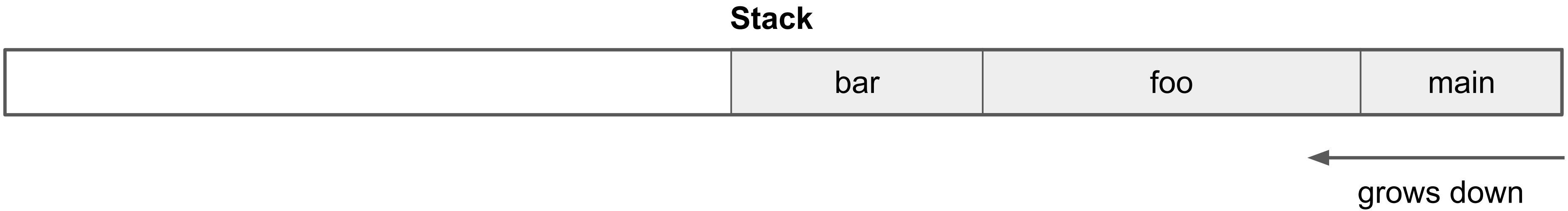
Thread Stack



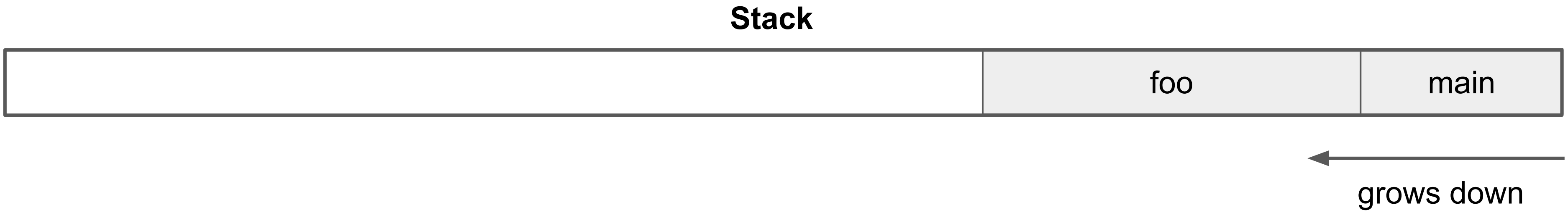
Thread Stack



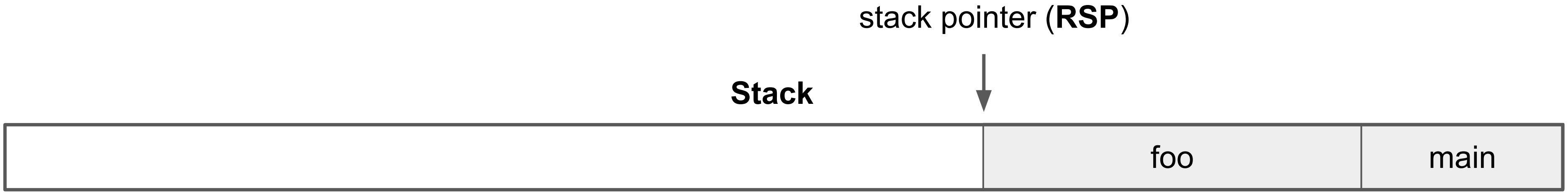
Thread Stack



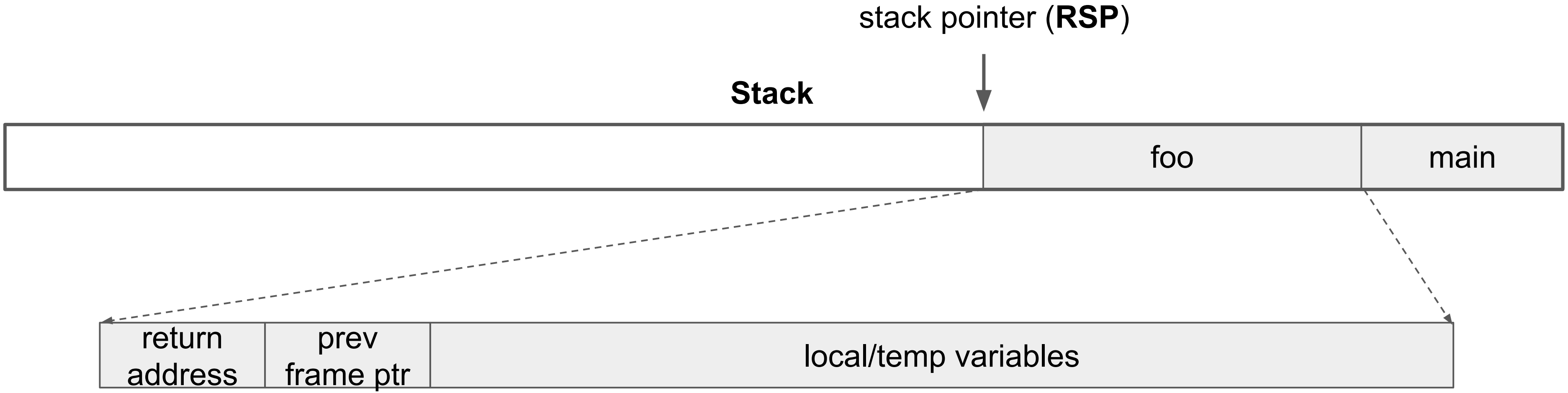
Thread Stack



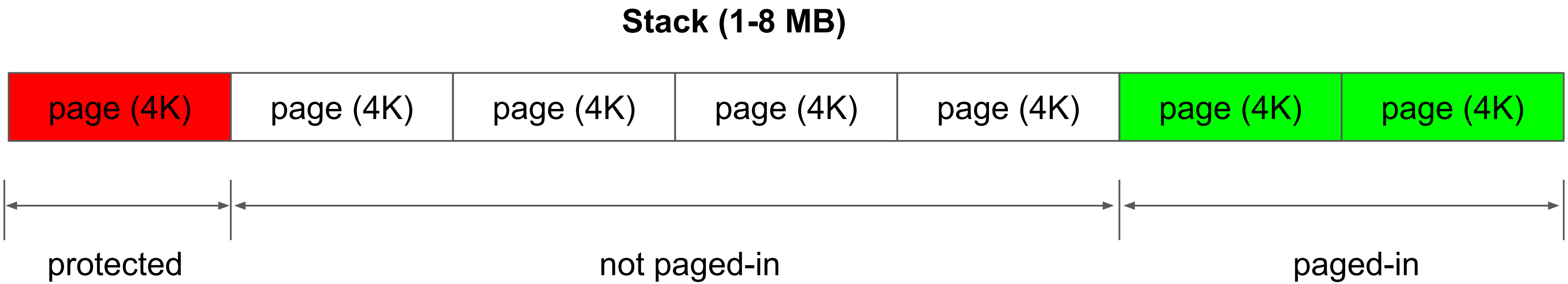
Thread Stack



Thread Stack



Stack Implementation



Stack (1-8 MB)

page (4K) page (4K) page (4K) page (4K) page (4K) page (4K) page (4K)

protected

not paged-in

paged-in

Stack is cheap!

```
foo:
sub    $64, %RSP          // allocate stack frame of size 64
...
mov    %RAX, 16(%RSP)     // store to a local var
...
add    $64, %RSP          // deallocate stack frame
retq
```

Paging-based infinite stacks?

- Lazy page-in
- 64-bit Virtual Address Space

Can we build "infinite" stacks based on this?

What is infinite?

1GB is "infinite" enough

Paging won't work :(

- Not enough Address Space
    - 48 bits address space
    - 1 bit for kernel = 47 bits = 128TB
    - max 128K stacks

Paging won't work :(

- Not enough Address Space
  - 48 bits address space
  - 1 bit for kernel = 47 bits = 128TB
  - max 128K stacks
- Bad granularity
  - 4KB x 1M = 4GB

Paging won't work :(

- Not enough Address Space
  - 48 bits address space
  - 1 bit for kernel = 47 bits = 128TB
  - max 128K stacks
- Bad granularity
  - 4KB x 1M = 4GB
- Slow "page-out"

Paging won't work :(

- Not enough Address Space
  - 48 bits address space
  - 1 bit for kernel = 47 bits = 128TB
  - max 128K stacks
- Bad granularity
  - 4KB x 1M = 4GB
- Slow "page-out"
- No huge pages (2MB, 1GB)

Paging won't work :(

- Not enough Address Space
  - 48 bits address space
  - 1 bit for kernel = 47 bits = 128TB
  - max 128K stacks
- Bad granularity
  - 4KB x 1M = 4GB
- Slow "page-out"
- No huge pages (2MB, 1GB)
- 32-bit systems
  - ARM

Normal stack again

```
foo:




sub     $64, %RSP
...
mov     %RAX, 16(%RSP)
...
add     $64, %RSP
retq
```

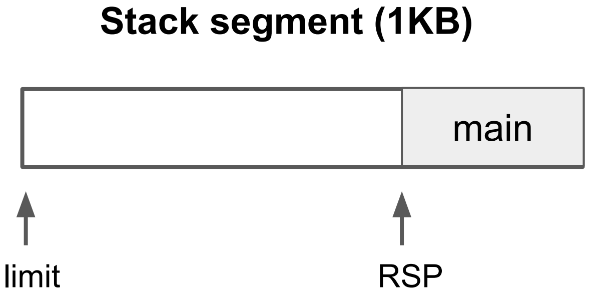
Goroutine stacks

```
foo:
mov    %fs:-8, %RCX       // load G descriptor from TLS
cmp    16(%RCX), %RSP     // compare the stack limit and RSP
jbe    morestack          // jump to slow-path if not enough stack
sub    $64, %RSP
...
mov    %RAX, 16(%RSP)
...
add    $64, %RSP
retq
...
morestack:                // call runtime to allocate more stack
callq  <runtime.morestack>
```

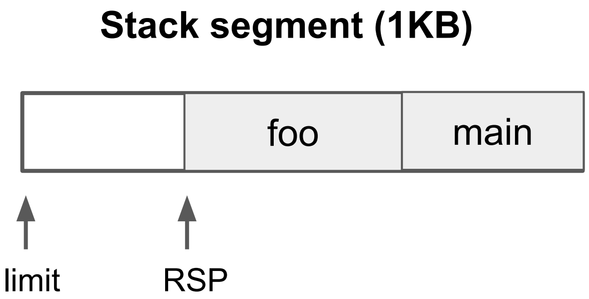
Function Prologue

```
void foo()
{
  if (RSP < TLS_G->stack_limit)
    morestack();
  ...
}
```

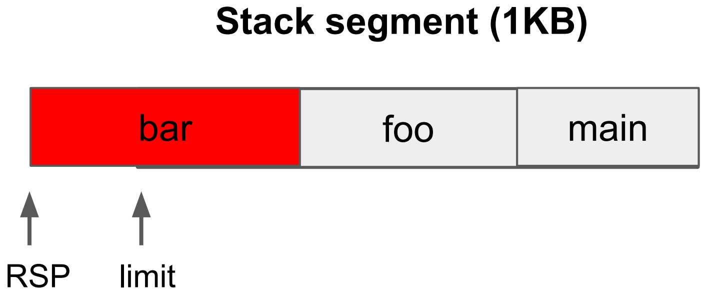
Split Stack



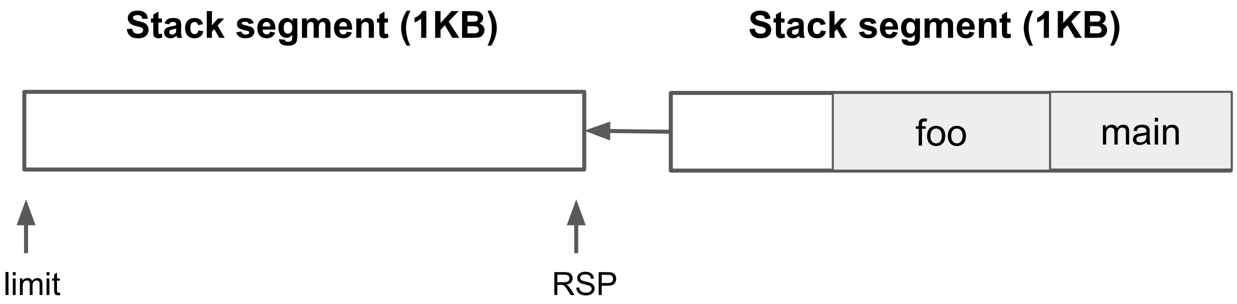
Split Stack



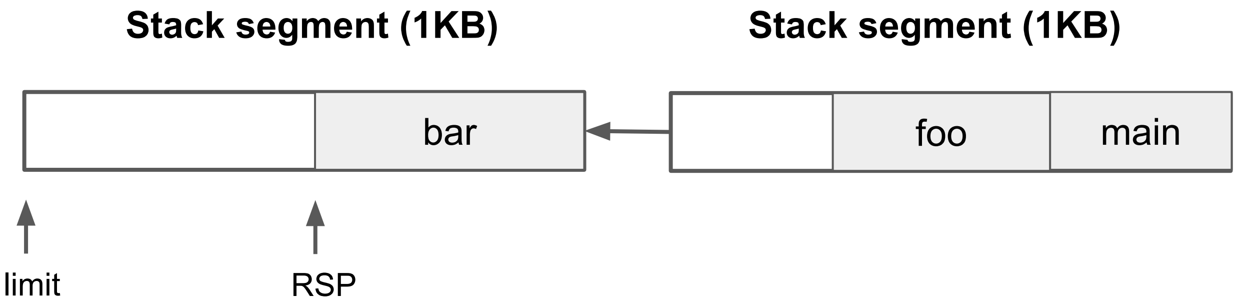
Split Stack



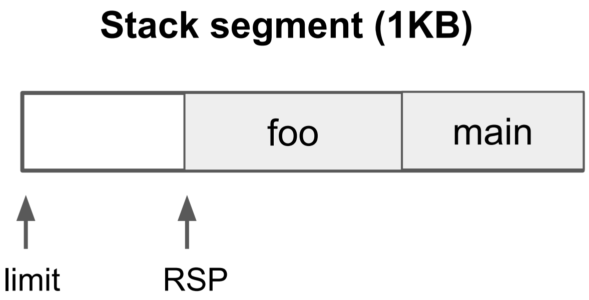
Split Stack



Split Stack



Split Stack



Split Stack Benefits

- 1M goroutines
- works on 32-bits
- good granularity
- cheap "page-out"
- huge pages

"Hop Split" Problem :(

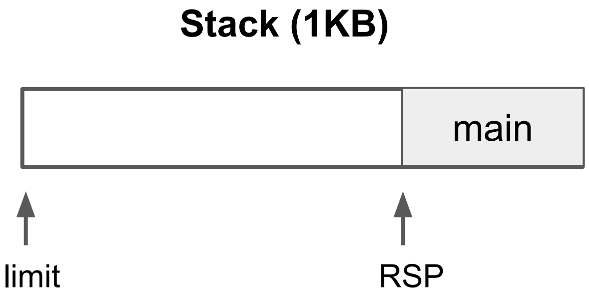
```
for ... {    // hot loop
    foo()    // causes stack split
}
```

Important Performance Characteristics

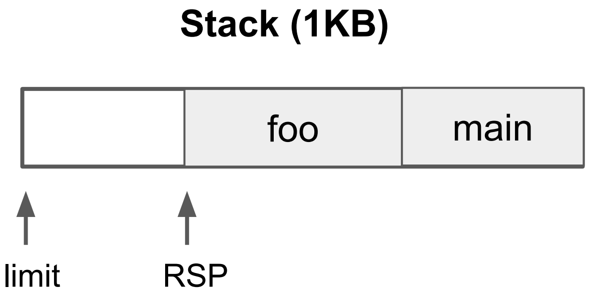
1. Transparent
2. Stable

"Hot Split" problem fail both.

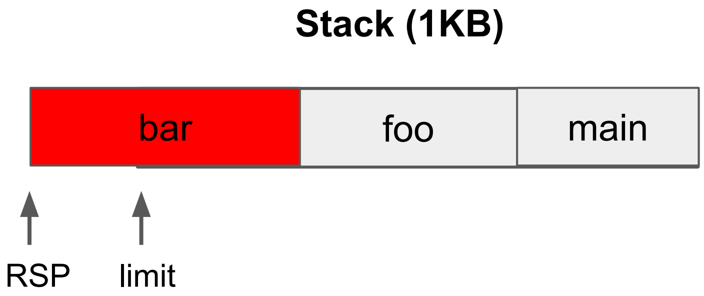
Growable Stack



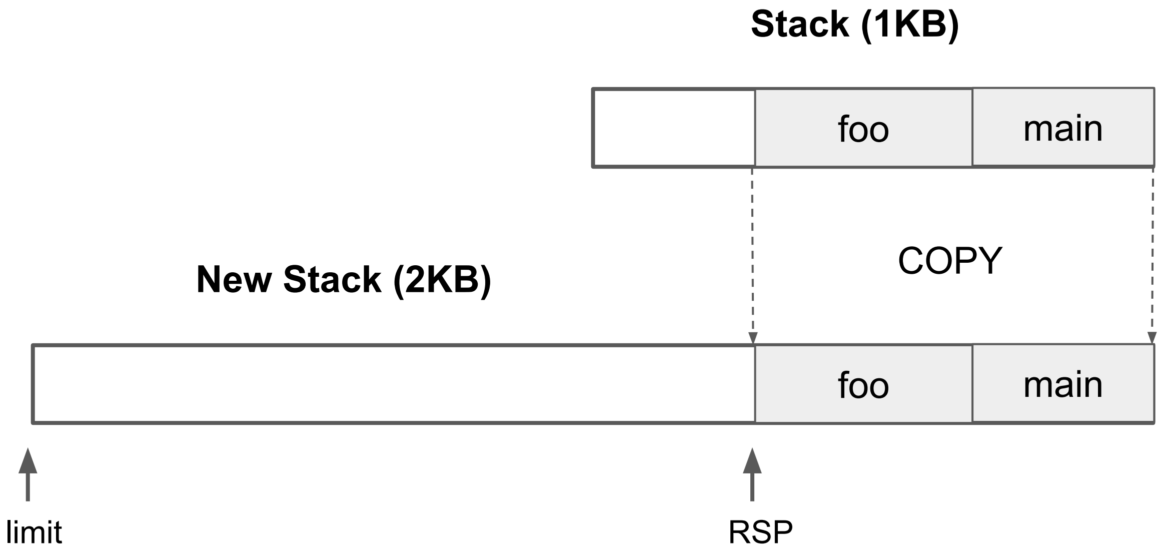
Growable Stack



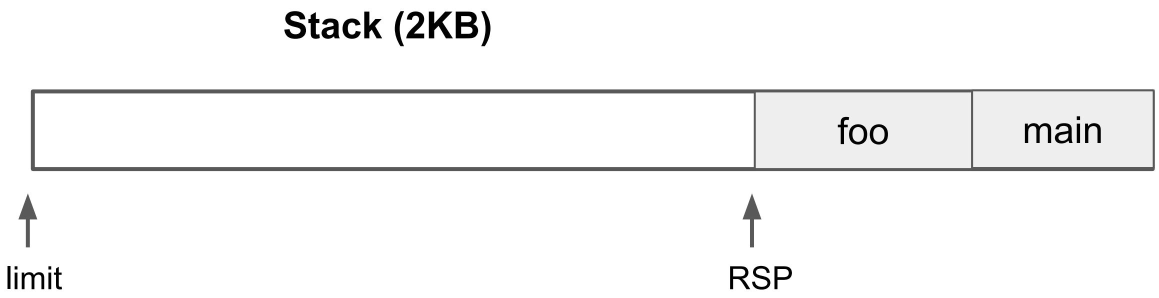
Growable Stack



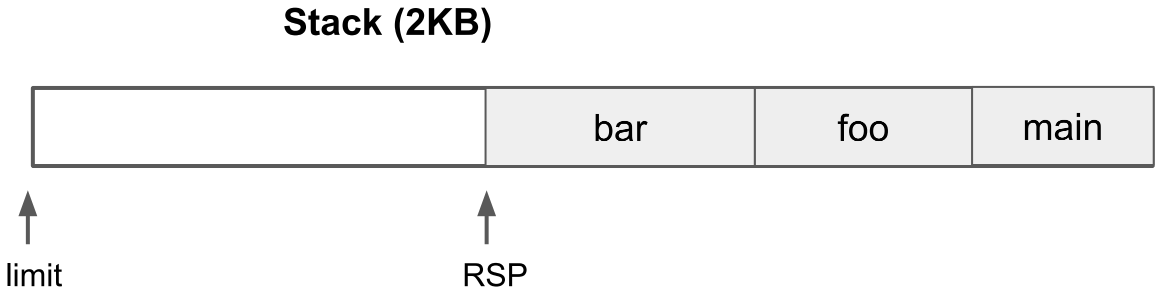
Growable Stack



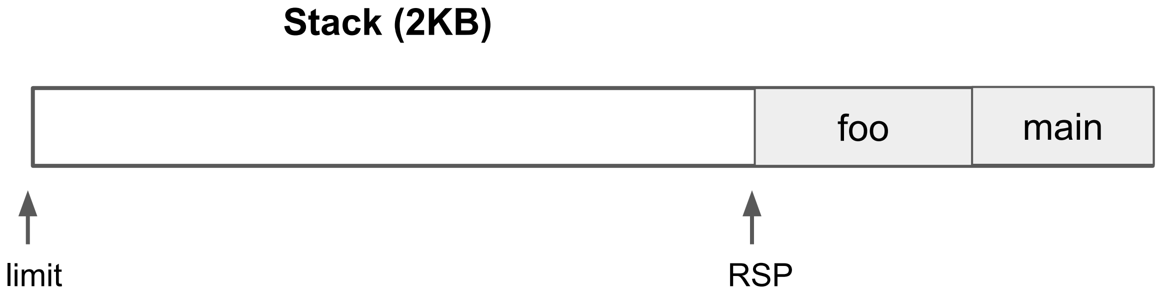
Growable Stack



Growable Stack



Growable Stack



Stack Performance

Split Stack

- **O(1)** cost per function call
- **repeated**

Worst case: stack split in hot loop

Stack Performance

Split Stack

- **O(1)** cost per function call
- **repeated**

Worst case: stack split in hot loop

Growable Stack

- **O(N)** cost per function call
- **amortized**

Worst case: growing stack for short goroutine

Stack Performance

Split Stack

- **O(1)** cost per function call
- **repeated**

Worst case: stack split in hot loop

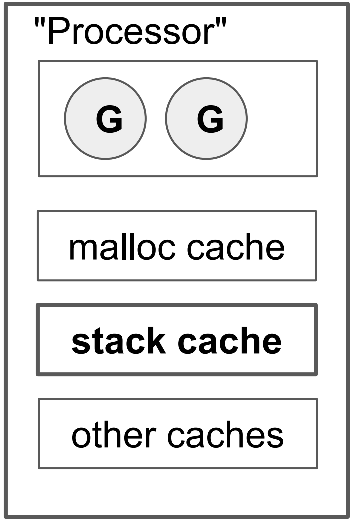
Growable Stack

- **O(N)** cost per function call
- **amortized**

Worst case: growing stack for short goroutine

Penalizing **cheap** operation a **bit** < Penalizing **expensive** operation **significantly**

Stack Cache



Interesting Fact

Split stacks are in gcc:

```
$ gcc -fsplit-stack prog.c
```

Preemption

What: Asynchronously asking a goroutine to yield.

Why:

- multiplexing multiple goroutines
- auxiliary functions (GC, crashes)

Preemption

What: Asynchronously asking a goroutine to yield.

Why:

- multiplexing multiple goroutines
- auxiliary functions (GC, crashes)

Preemption is also like **Oxygen**

Implementation strategy

Signals:

+ Fast

Implementation strategy

Signals:

+ Fast

- OS-dependent

- non-preemptible regions

- GC stack/register maps

⛔

Implementation strategy

Signals:

+ Fast
- OS-dependent
- non-preemptible regions
- GC stack/register maps

⛔

Cooperative checks:

+ OS-independent
+ non-preemptible regions
+ GC stack/register maps

Implementation strategy

Signals:

- + Fast
- − OS-dependent
- − non-preemptible regions
- − GC stack/register maps

⛔

Cooperative checks:

- + OS-independent
- + non-preemptible regions
- + GC stack/register maps
- − Slow (1-10%)

⛔

Function Prologue

```
foo:
mov    %fs:-8, %RCX      // load G descriptor from TLS
cmp    16(%RCX), %RSP    // compare the stack limit and RSP
jbe    morestack         // jump to slow-path if not enough stack
...
```

Spoof stack limit!

```
G->stackLimit = 0xfffffffffffffade
```

Function Prologue

```
foo:
mov     %fs:-8, %RCX

cmp     16(%RCX), %RSP    // guaranteed to fail!

jbe     morestack

...
```

Advantages

- ＋ fast
- ＋ portable
- ＋ simple
- ＋ GC-friendly

Advantages

+ fast
+ portable
+ simple
+ GC-friendly
- loops

Recap

√ lightweight goroutines

√ handling of IO and syscalls

√ parallel

√ scalable

√ efficient

√ fair

√ infinite stacks

√ preemptible*

Thank you!

Q&A

Dmitry Vyukov, dvyukov@google.com