The Scheduler Saga

@kavya719

kavya

the innards of
the scheduler

the scheduler

||

the behind-the-scenes orchestrator

of Go programs.

```go
func main() {
  // Create goroutines.
  for _, i := range images {
      go process(i)
  }
  ...
  // Wait.
  <-ch
}
```

```
func main() {
  // Create goroutines.
  for _, i := range images {
      go process(i)
  }
  ...
  // Wait.
  <-ch
}

func process(image) {
  // Create goroutine.
  go reportMetrics()

  complicatedAlgorithm(image)

  // Write to file.
  f, err := os.OpenFile()
  ...
}
```

→ **runs** goroutines created

→ **pauses and resumes** them:
blocking channel operations,
mutex operations.

→ coordinates:
blocking **system calls,** network I/O,
runtime tasks garbage collection.

```go
func main() {
  // Create goroutines.
  for _, i := range images {
      go process(i)
  }
  ...
  // Wait.
  <-ch
}
```

runs goroutines created

pauses and resumes them:
blocking channel operations,
mutex operations.

```go
func process(image) {
  // Create goroutine.
  go reportMetrics()

  complicatedAlgorithm(image)

  // Write to file.
  f, err := os.OpenFile()
  ...
}
```

coordinates:
blocking **system calls,** network I/O,
runtime tasks garbage collection.

…for **hundreds of thousands** of goroutines*

* dependent on workload and hardware, of course.

the **design** of the scheduler, & its scheduling **decisions** impact the **performance** of our programs.

spec it

the important questions.

build!

the big ideas & one sneaky idea.

assess it.

the difficult questions.

spec it

the what, when & why.

why have a scheduler?

why have a scheduler?

goroutines are user-space threads.

- conceptually similar to kernel threads managed by the OS, but **managed entirely by the Go runtime.**

why have a scheduler?

goroutines are user-space threads.

- conceptually similar to kernel threads managed by the OS, but **managed entirely by the Go runtime.**

- **lighter-weight and cheaper** than kernel threads.

  smaller memory footprint:
  initial goroutine stack = 2KB; default thread stack = 8KB.
  state tracking overhead.

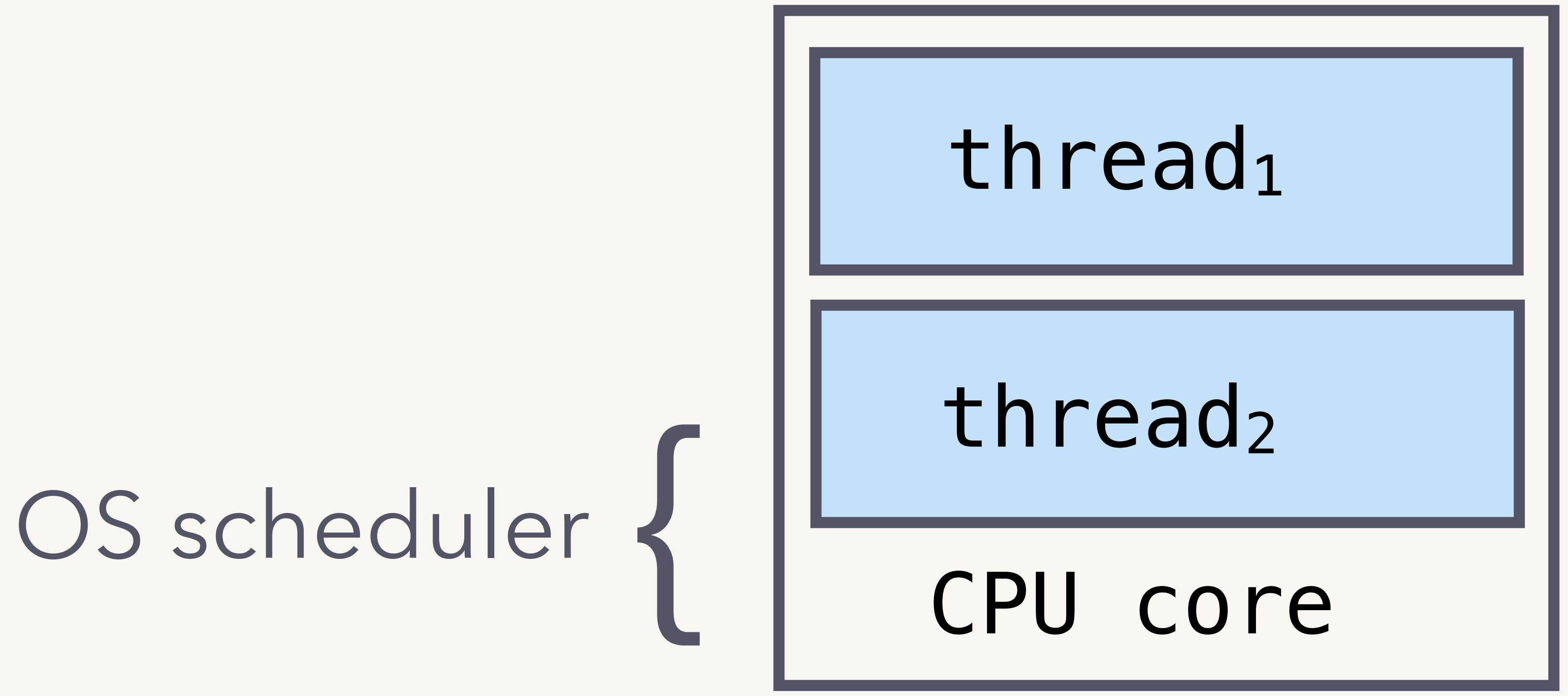
  faster creation, destruction, context switches:
  goroutine switches = ~tens of ns; thread switches = ~a µs.

why have a scheduler?

goroutines are user-space threads.

- conceptually similar to kernel threads managed by the OS, but **managed entirely by the Go runtime.**
- **lighter-weight and cheaper** than kernel threads.

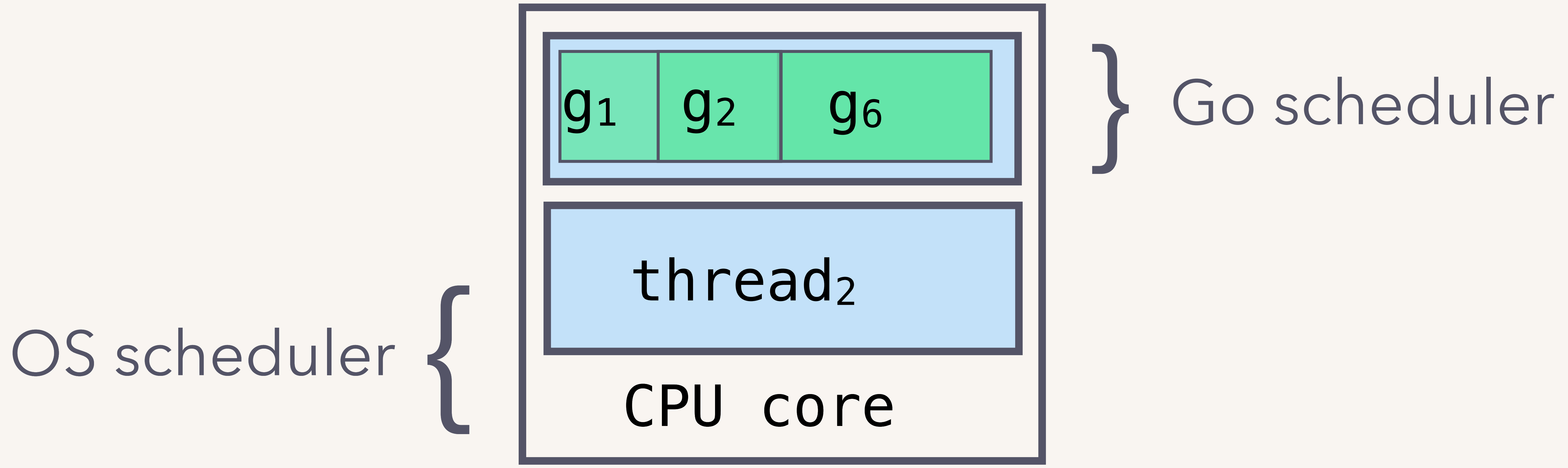
...but how are they *run*?



why have a scheduler?

goroutines are user-space threads.

- conceptually similar to kernel threads managed by the OS, but **managed entirely by the Go runtime.**
- **lighter-weight and cheaper** than kernel threads.
- **multiplexed** onto kernel threads.



when does it schedule?

when does it schedule?

At operations that <u>should or would</u> affect <u>goroutine execution.</u>

```
func main() {
  // Create goroutines.
  for _, i := range images {
      go process(i)
  }
  ...
  // Wait.
  <-ch
}

func process(image) {
  // Create goroutine.
  go reportMetrics()

  complicatedAlgorithm(image)

  // Write to file.
  f, err := os.OpenFile()
  ...
}
```

→ goroutine creation

run new goroutines soon,
continue this for now.

→ goroutine blocking

pause this one immediately.

→ blocking system call

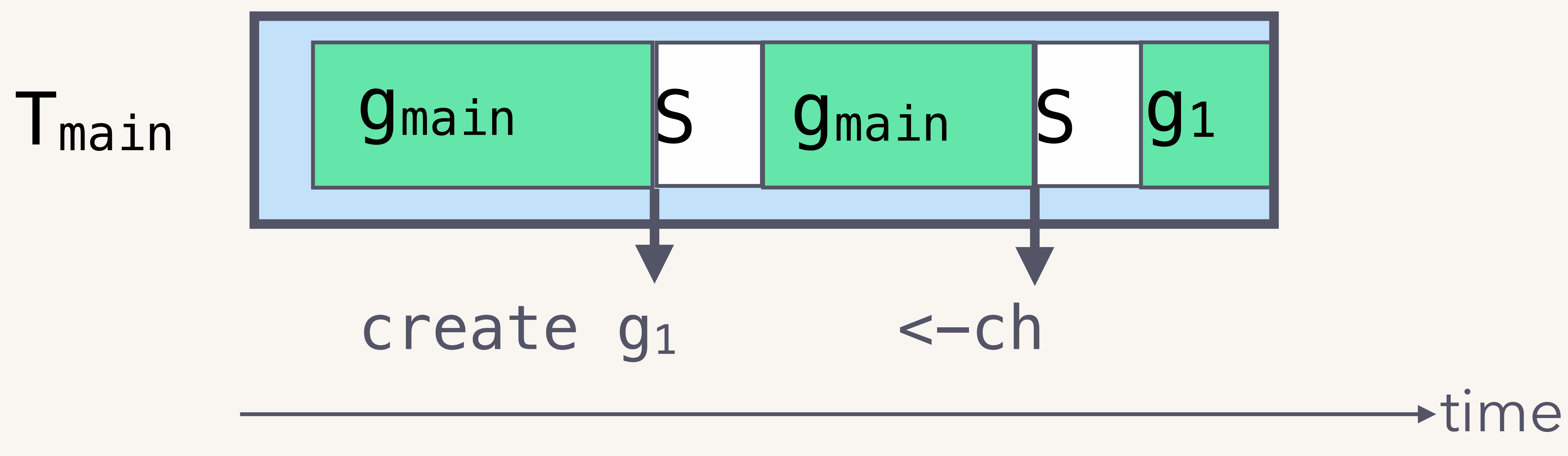
the <u>thread itself blocks</u> too!

when does it schedule?

At operations that should or would affect goroutine execution.

The runtime causes a switch into the scheduler under-the-hood,
and the scheduler may schedule a different goroutine on this thread.

T: threads
g: goroutines
S: scheduler



#schedgoals

for scheduling **goroutines** onto **kernel threads**.

- ☐ use a **small number of kernel threads**.

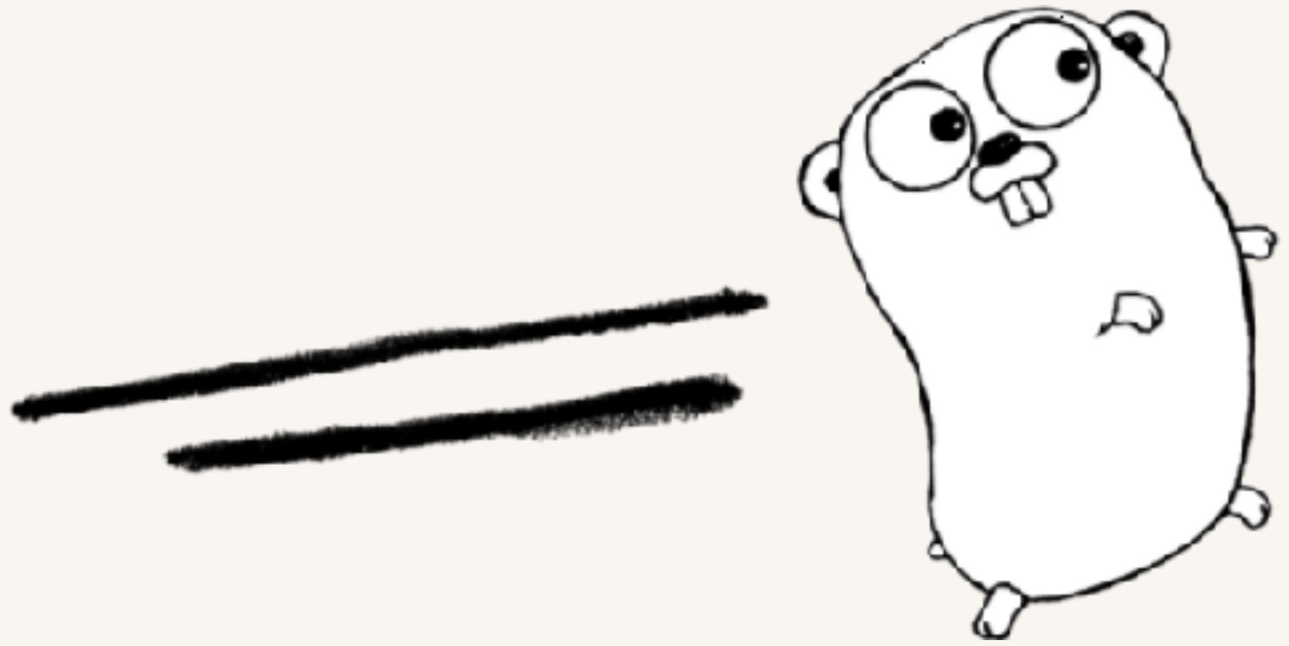
  kernel threads are expensive to create.

- ☐ support high **concurrency.**

  Go programs should be able to run lots and lots of goroutines.

- ☐ leverage **parallelism i.e. scale to N cores.**

  On an N-core machine, Go programs should be able to run N goroutines in parallel.*



* depending on the program structure, of course.

build it!

the big ideas & neat details.

how to multiplex goroutines onto kernel threads?

when to **create threads**?

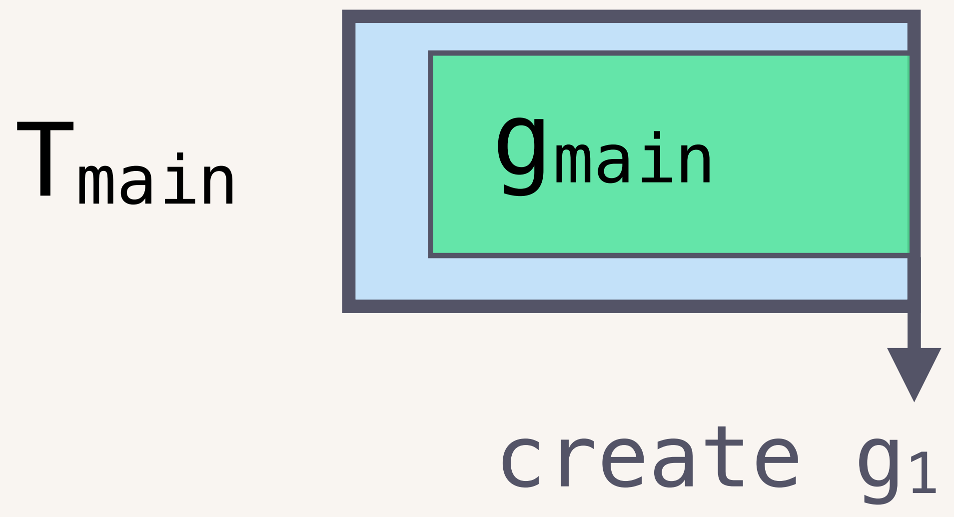
how to **distribute goroutines** across threads?

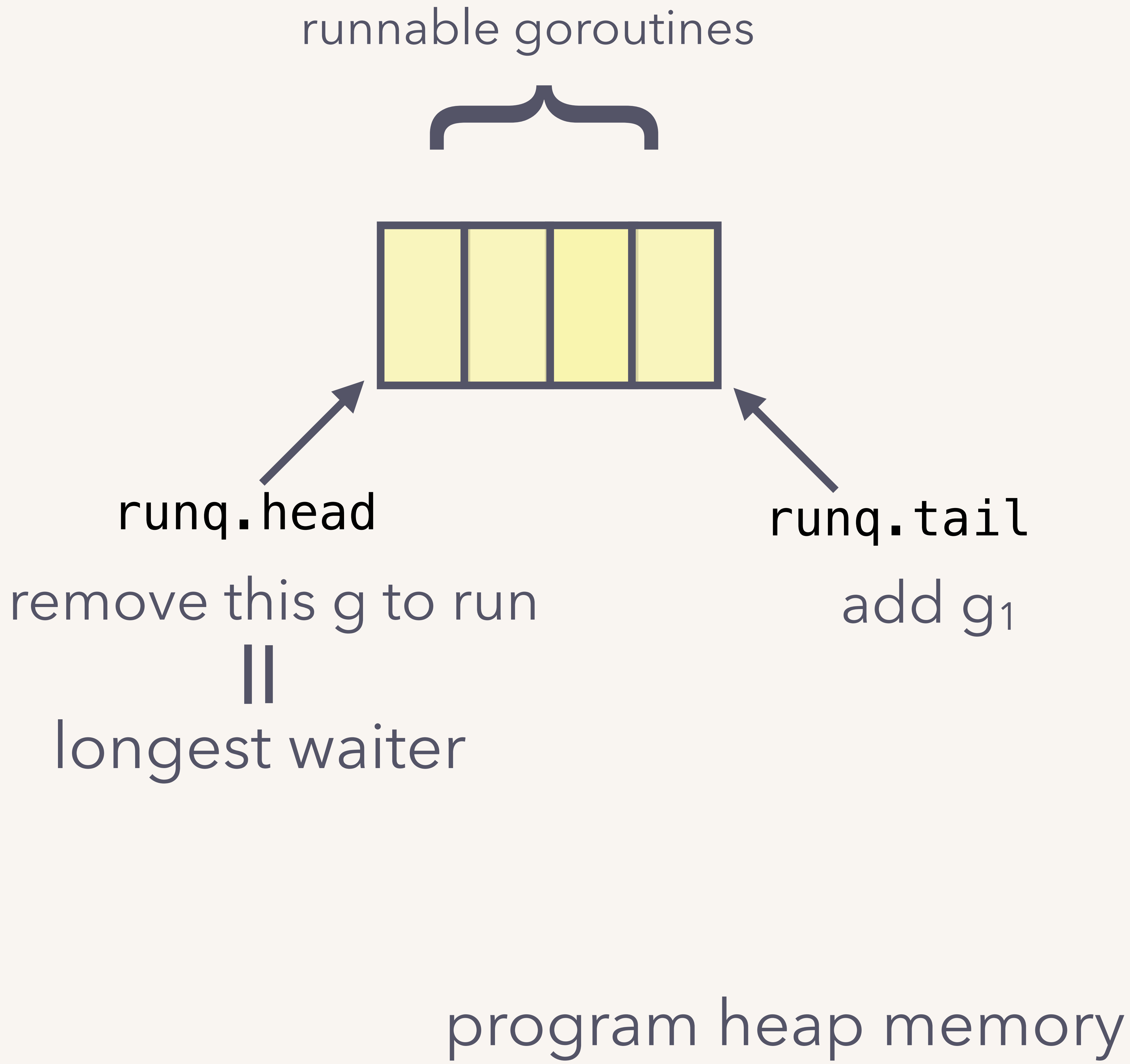
prelude: runqueues

Goroutines that ready-to-run and need to be scheduled are tracked in heap-allocated FIFO runqueues.

prelude: runqueues

Goroutines that ready-to-run and need to be scheduled are tracked in heap-allocated FIFO <u>runqueues</u>.





g_{main}

```
func main() {
  // Create goroutines.
  for _, i := range images {
      go process(i)
  }
  ...
  // Wait.
  <-ch
}
```

creates g_1,
g_2

goroutine blocks

g_1

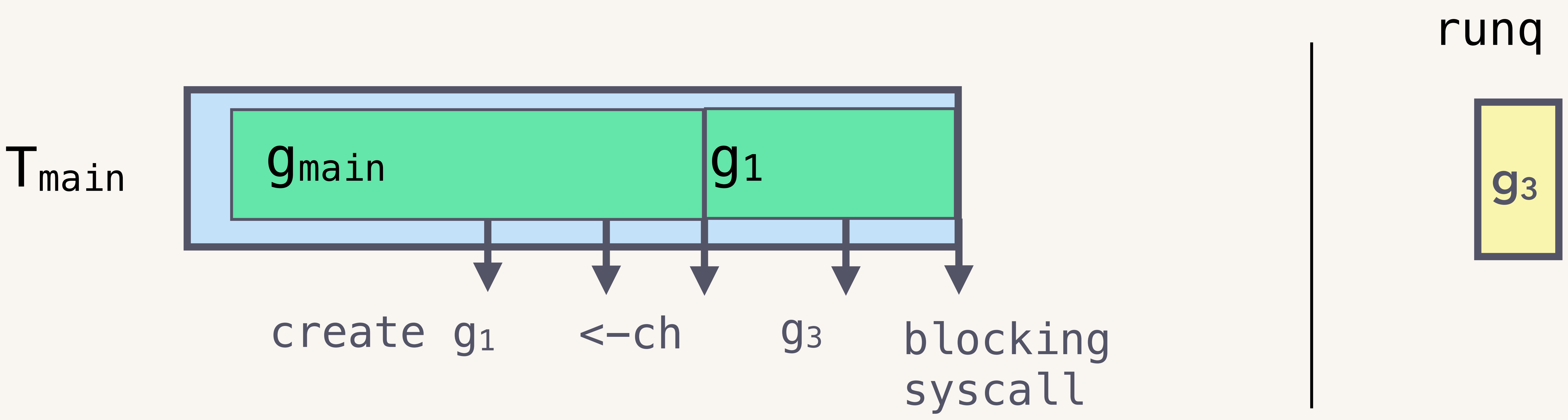
```
func process(image) {

   ...

}
```

assume: running on a box with 2 CPU cores.

first, the non-ideas

first, the non-ideas

I. Multiplex all goroutines on a single thread.



– no concurrency!

if a goroutine blocks the thread, no other goroutines run either.

– no parallelism possible:

can only use a single CPU core, even if more are available.

first, the non-ideas

II. Create & destroy a thread per-goroutine.

– **defeats the purpose of using goroutines**

threads are heavyweight, and expensive to create and destroy.

okay, here's an idea…

idea I: reuse threads

Create threads when needed

there're goroutines to run,
but all threads are busy.

idea I: reuse threads

Create threads when needed; keep them around for reuse.

there're goroutines to run,
but all threads are busy.

"**thread parking**" i.e.
put them to sleep;
no longer uses a CPU core.

track idle threads in a list.

idea I: reuse threads

Create threads <u>when needed</u>; <u>keep them around</u> for reuse.

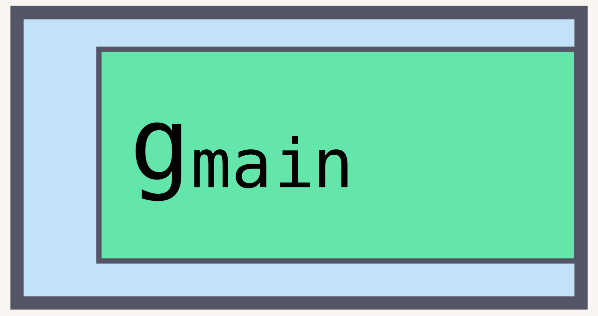
The threads get goroutines to run from a runqueue.

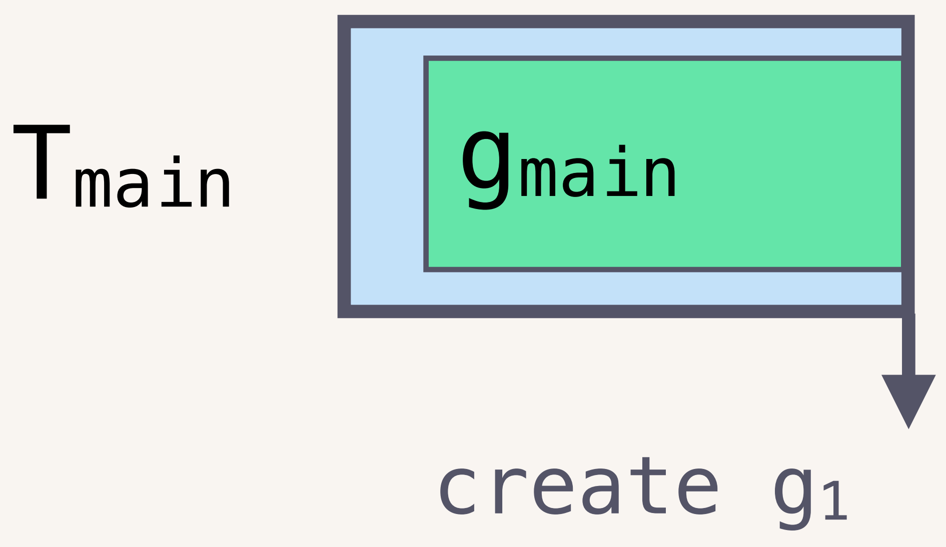
program | scheduler

runq

idle threads

T_{main}

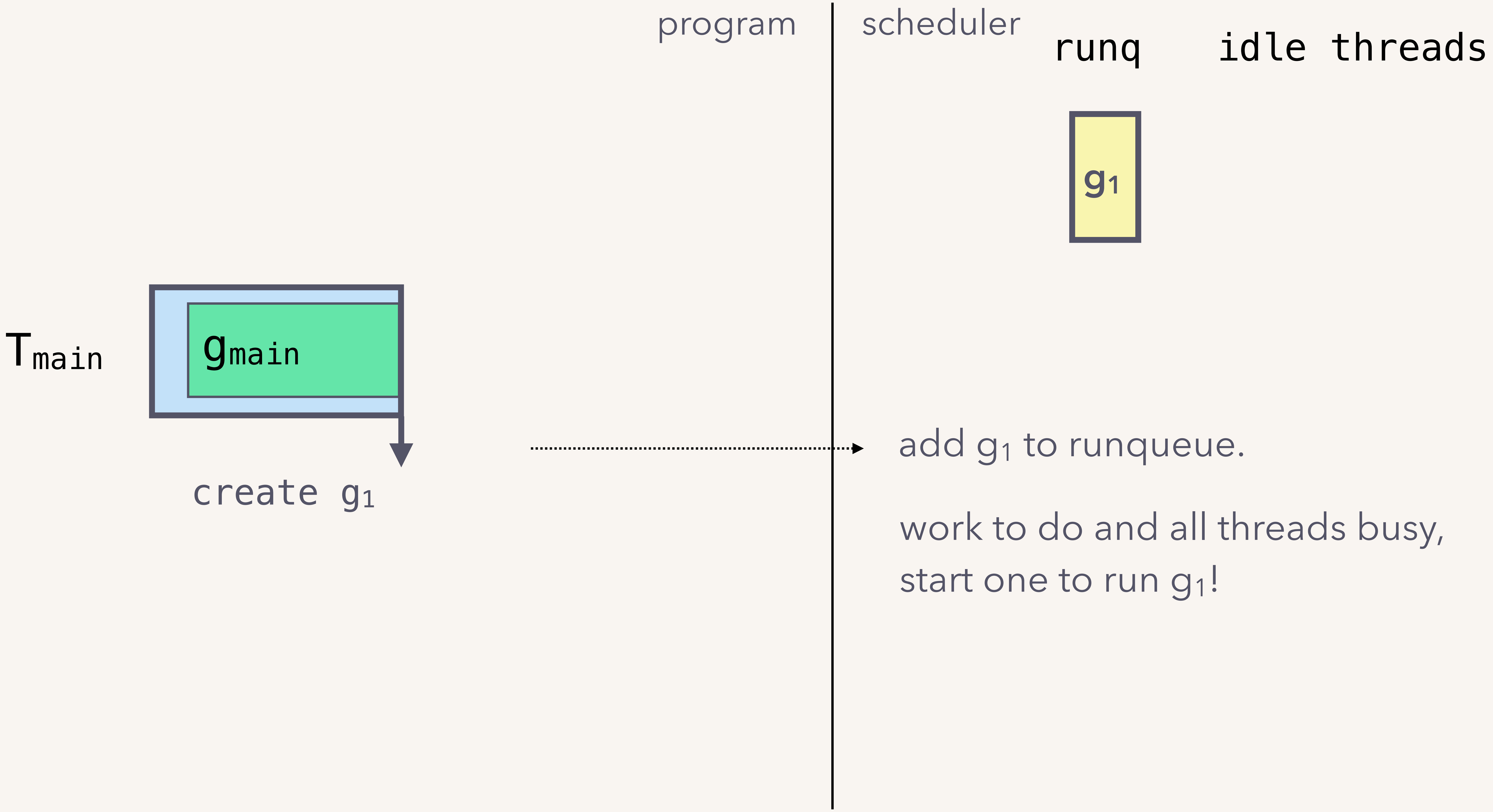


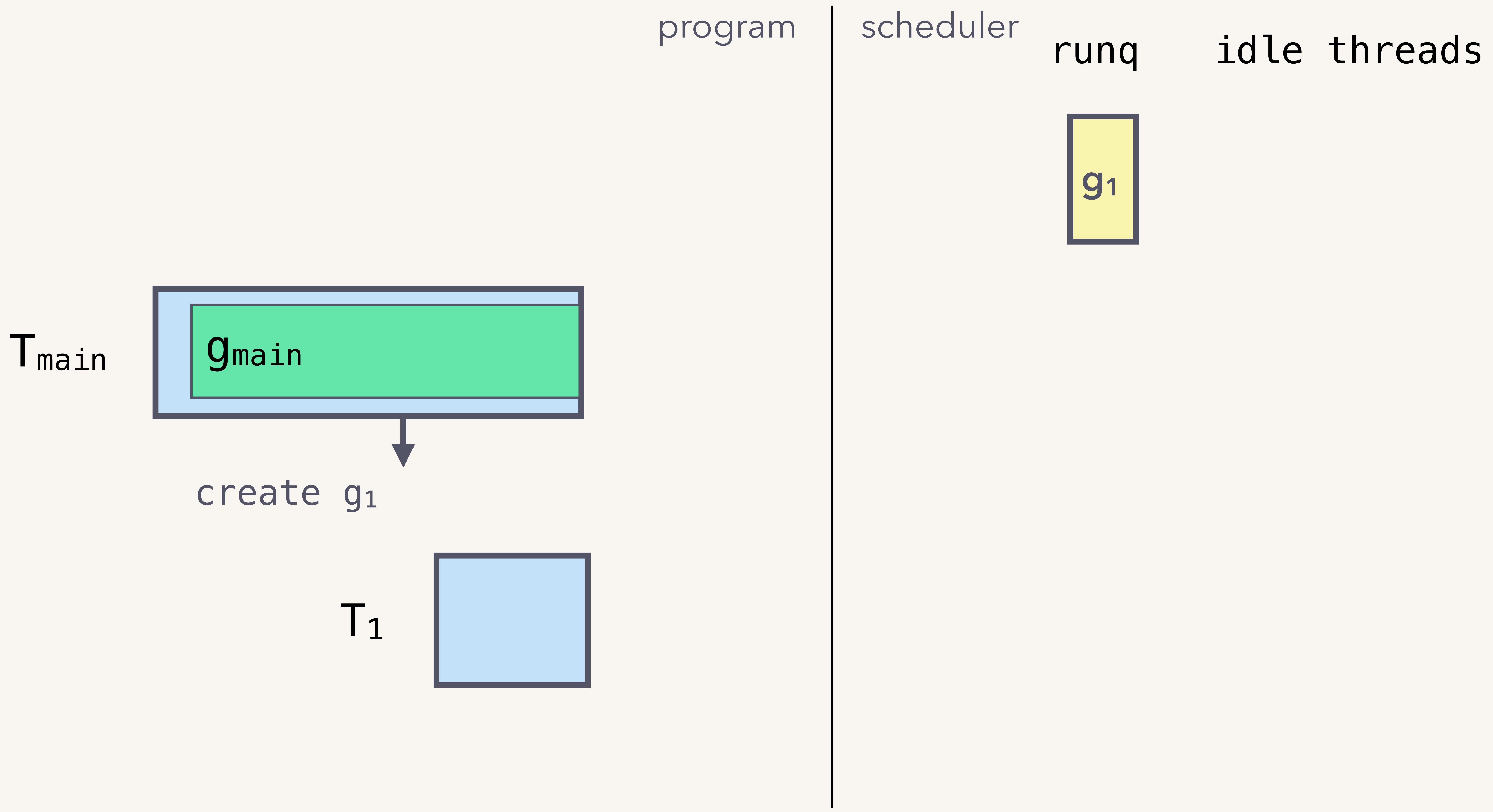


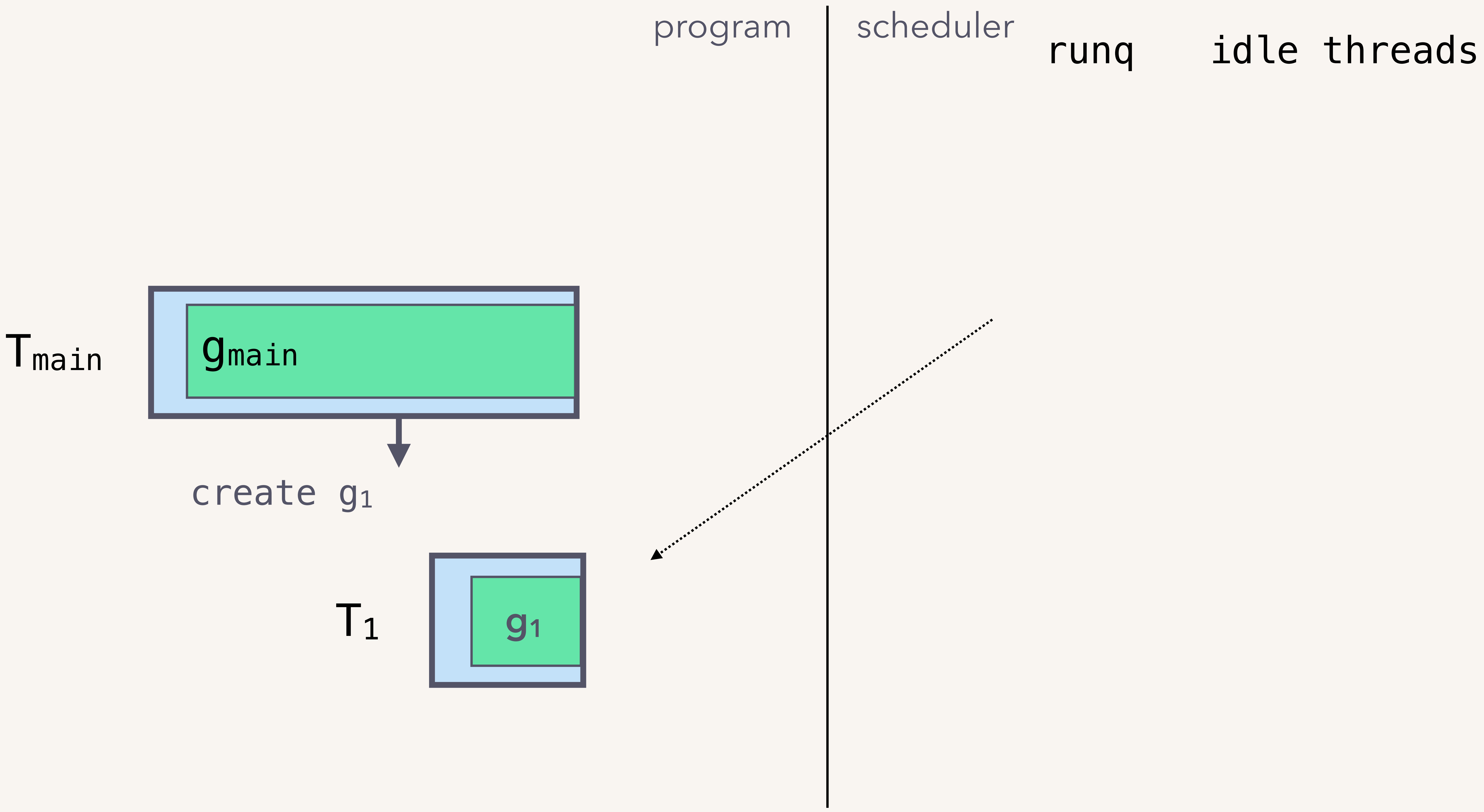
program | scheduler

runq

idle threads


program
scheduler
runq
idle threads
g1
Tmain
gmain
create g1
add g1 to runqueue.
work to do and all threads busy,
start one to run g1!


program
scheduler
runq
idle threads
g1
Tmain
gmain
create g1
T1



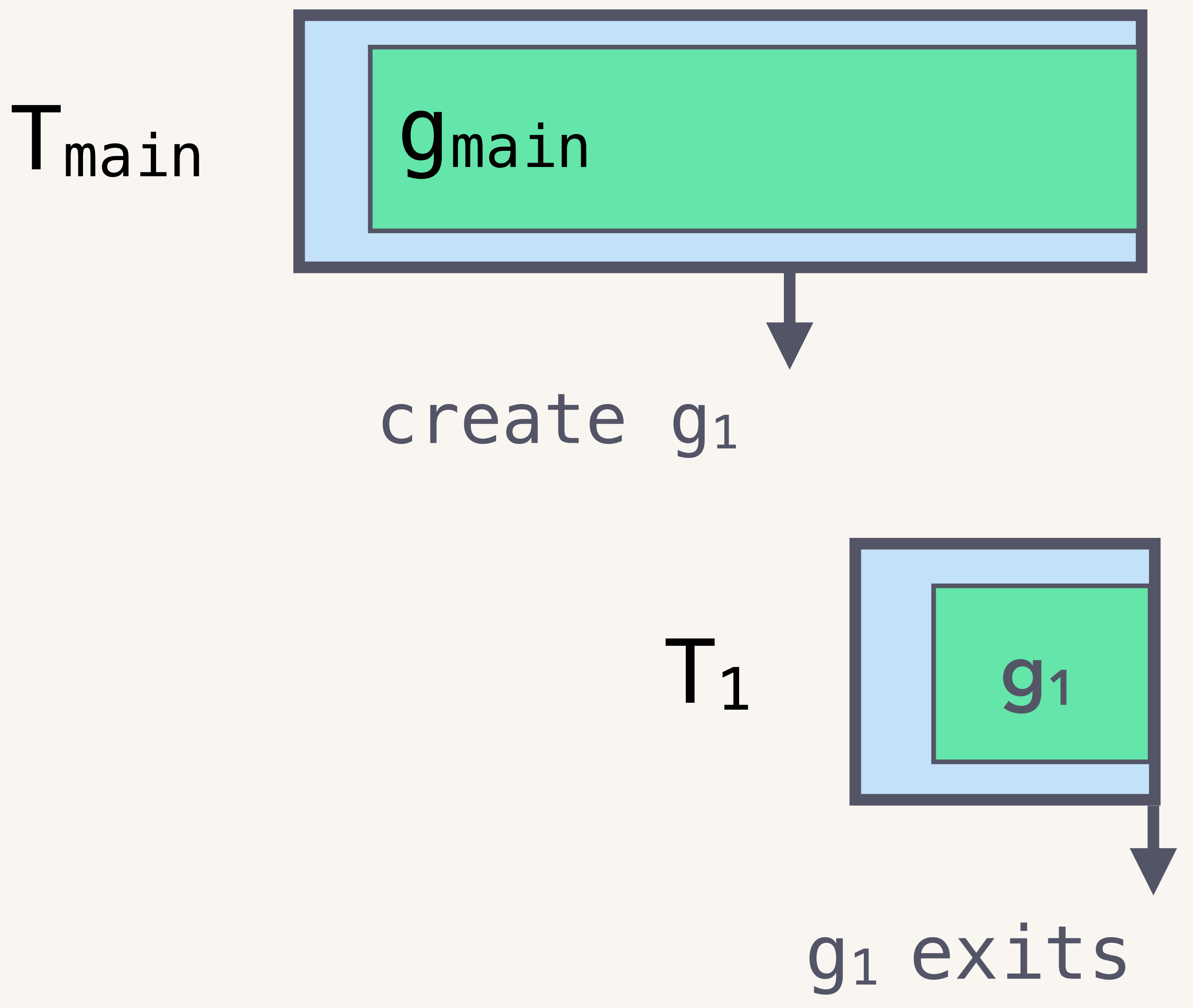
program
scheduler
runq
idle threads
T_{main}
g_{main}
create g_1
T_1
g_1

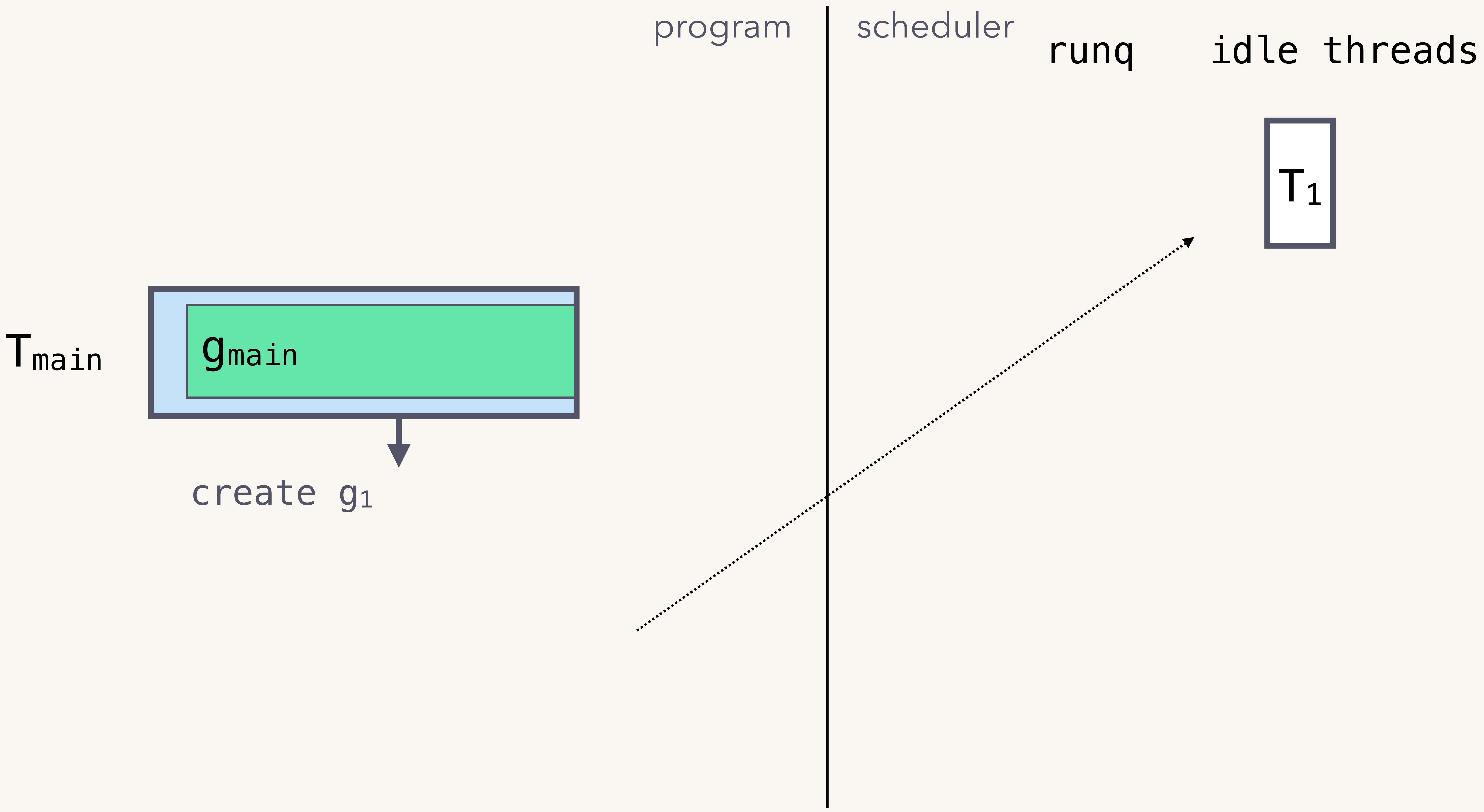
program

scheduler

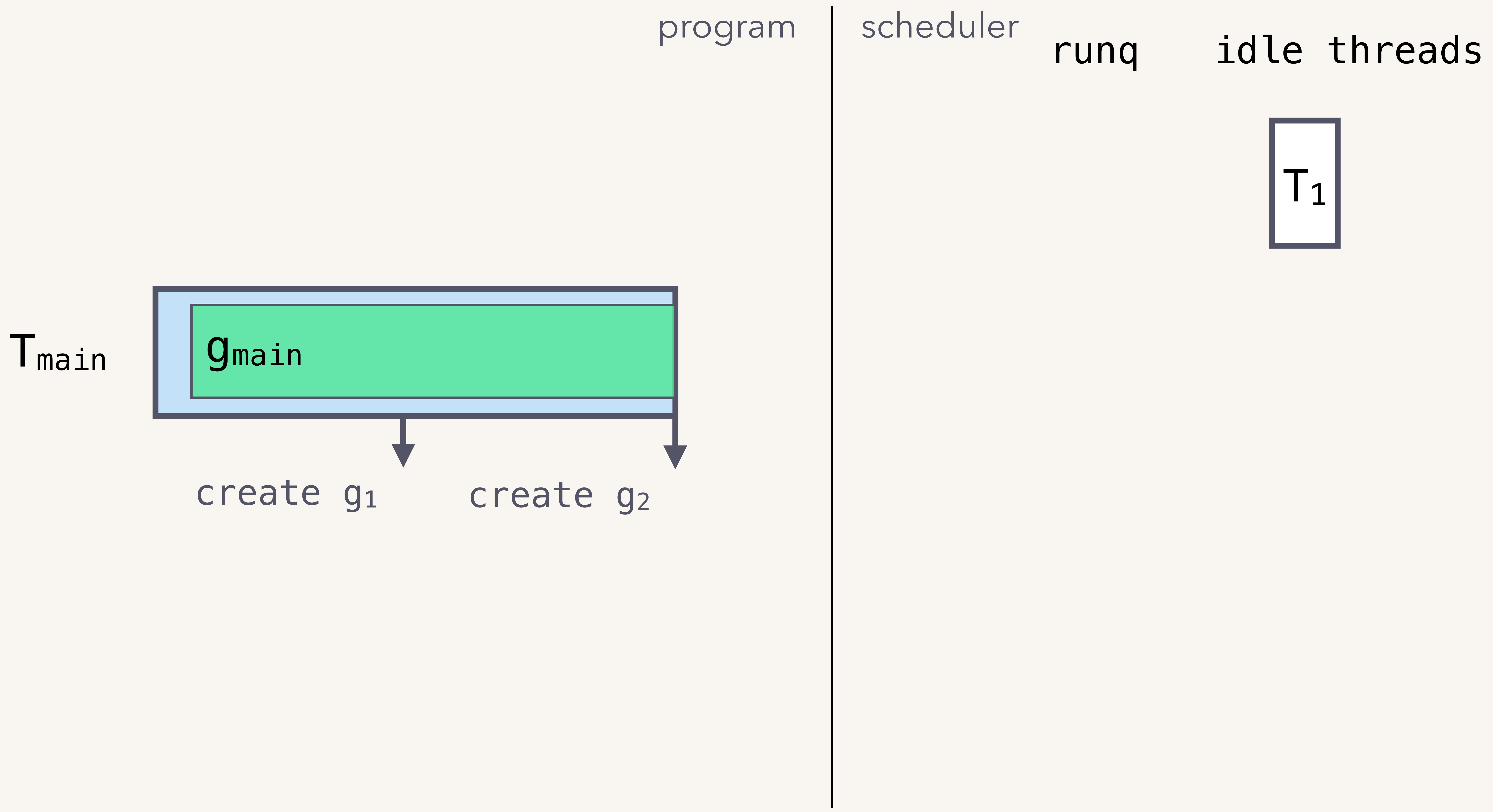
runq

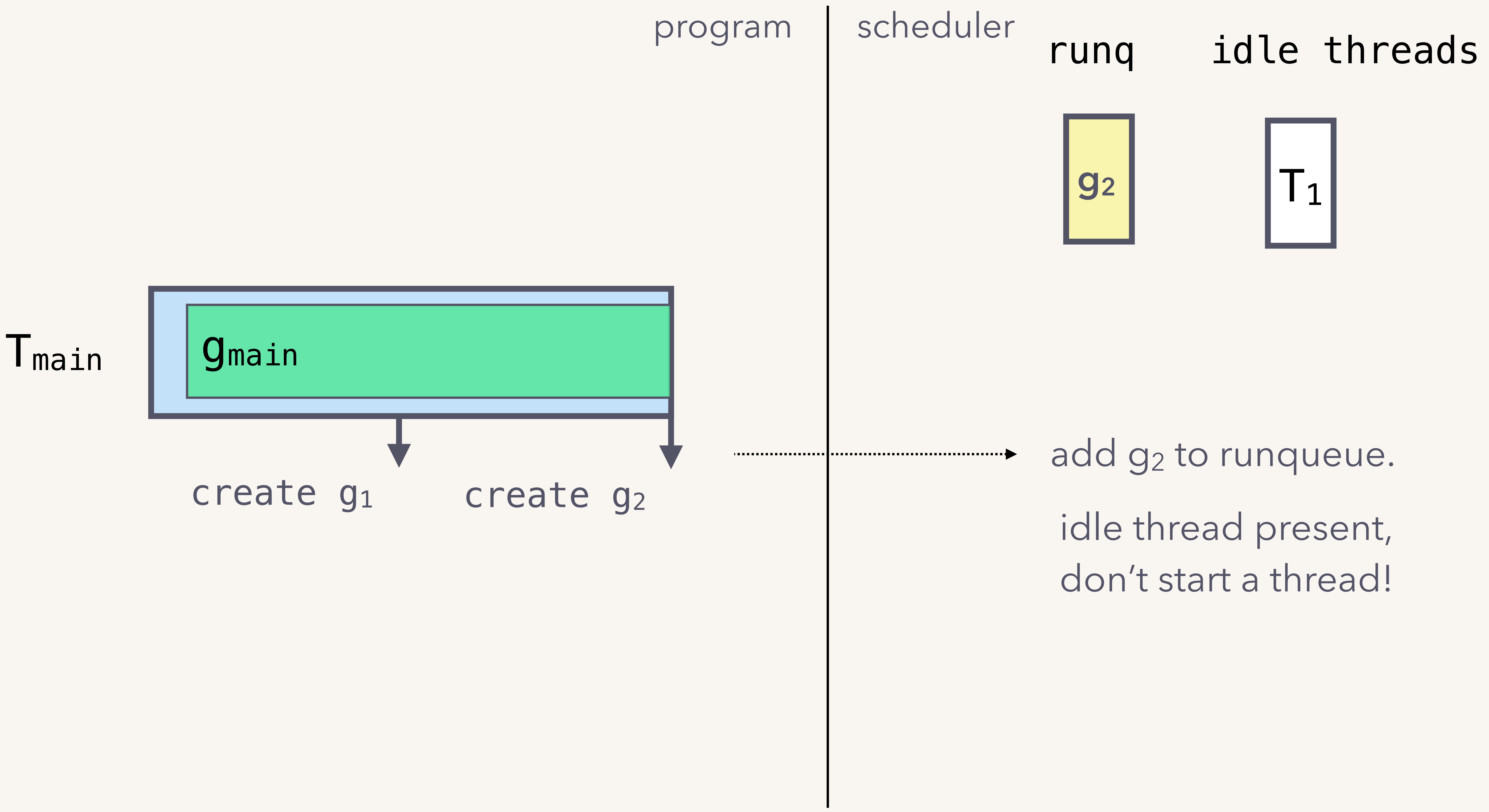
idle threads

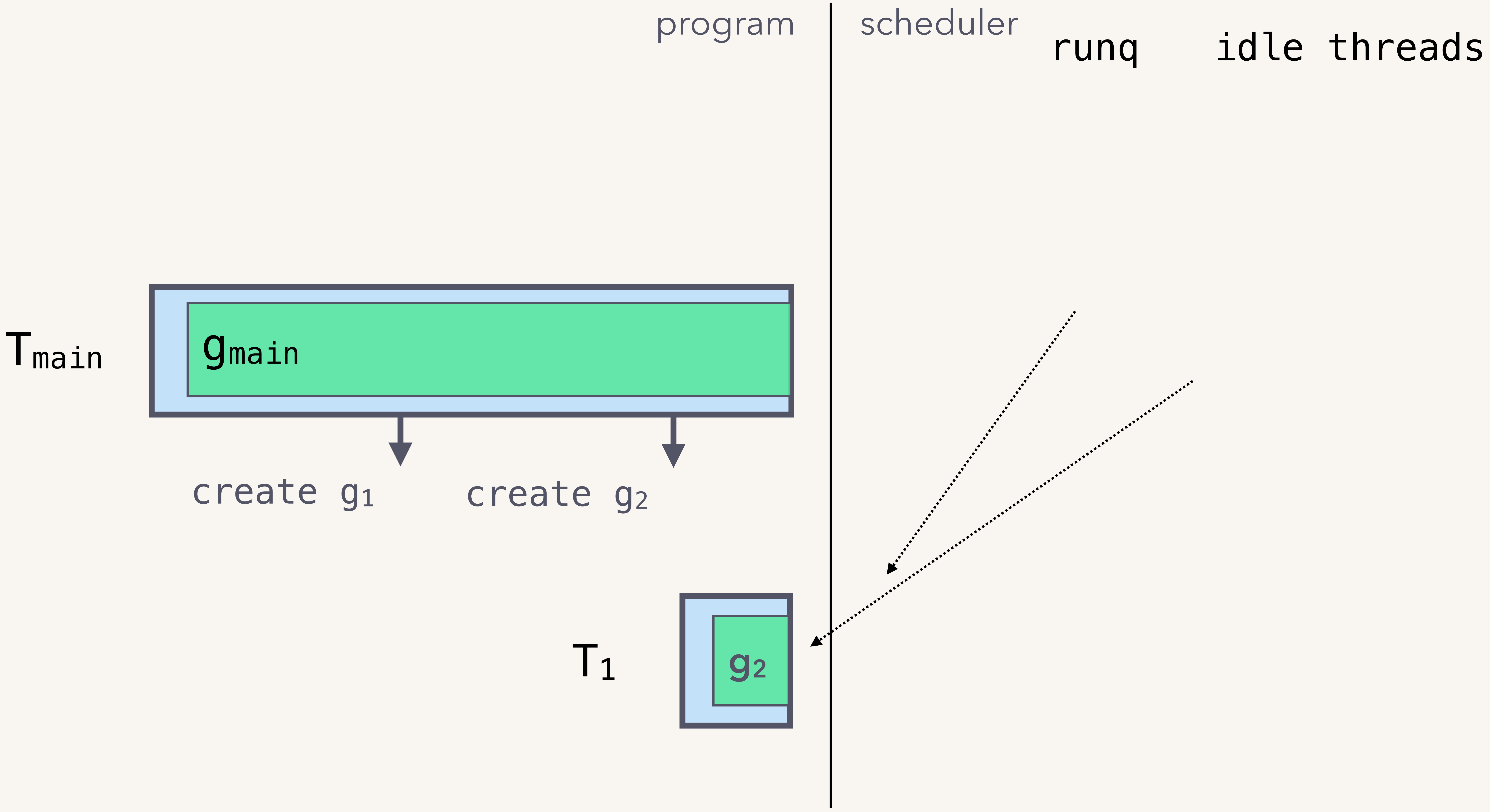




Say g_1 completes, park T_1 rather than destroying it.


program
scheduler
runq
idle threads
T1
Tmain
gmain
create g1
create g2


program
scheduler
runq
idle threads
g2
T1
Tmain
gmain
create g1
create g2
add g2 to runqueue.
idle thread present,
don't start a thread!



a match made in (scheduling) heaven.

sweet.

We have a scheme that nicely **reduces thread creations** and still provides concurrency, parallelism.

sweet.

We have a scheme that nicely **reduces thread creations** and still provides concurrency, parallelism.

…but

– multiple threads access the same runqueue, **so need a lock.**

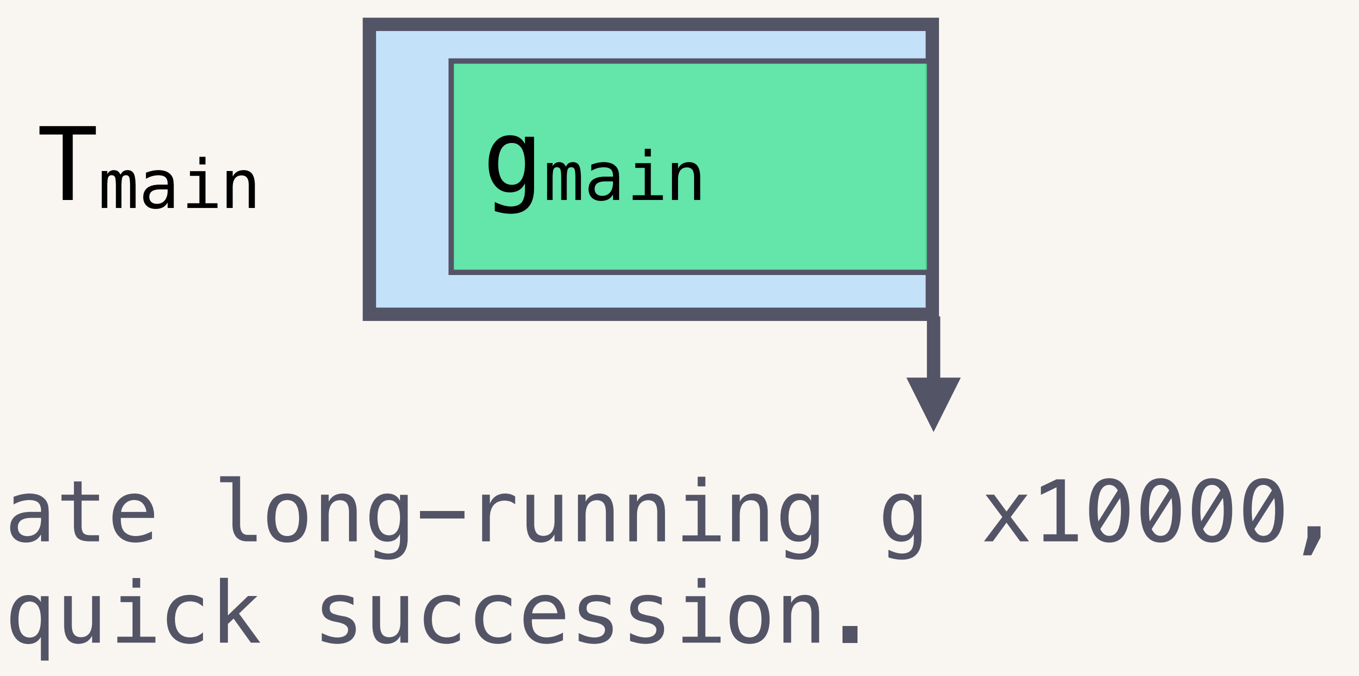
serializes scheduling.

sweet.

We have a scheme that nicely **reduces thread creations** and still provides concurrency, parallelism.

...but

– multiple threads access the same runqueue, **so need a lock.**



sweet.

We have a scheme that nicely **reduces thread creations** and still provides concurrency, parallelism.

…but

– multiple threads access the same runqueue, **so need a lock.**

– an **unbounded number of threads** can still be created.

sweet.

We have a scheme that nicely **reduces thread creations** and still provides concurrency, parallelism.

...but

– multiple threads access the same runqueue, **so need a lock.**

– an **unbounded number of threads** can still be created.

$\Downarrow$

hella contention possible.

sweet.

We have a scheme that nicely **reduces thread creations** and still provides concurrency, parallelism.

…but

– multiple threads access the same runqueue, **so need a lock.**

– an **unbounded number of threads** can still be created.

hella not scalable.

reusing threads is still a good idea.

thread creation is expensive; reusing threads amortizes that cost.

If the problem is an unbounded number threads can access the runqueue…

idea II: limit threads accessing runqueue

idea II: limit threads accessing runqueue

Limit the number of threads accessing the runqueue.
As before, keep threads around for reuse;
get goroutines to run from the runqueue.

idea II: limit threads accessing runqueue

Limit the number of threads accessing the runqueue.
As before, keep threads around for reuse;
get goroutines to run from the runqueue.

threads that are running goroutines;
threads in syscalls etc. won't count
towards this limit.

idea II: limit threads accessing runqueue

Limit the number of threads accessing the runqueue.

…to what?

too many –> too much contention.

too few -> won't use all the CPU cores, i.e.
will give up on parallelism.

idea II: limit threads accessing runqueue

Limit the number of threads accessing the runqueue.

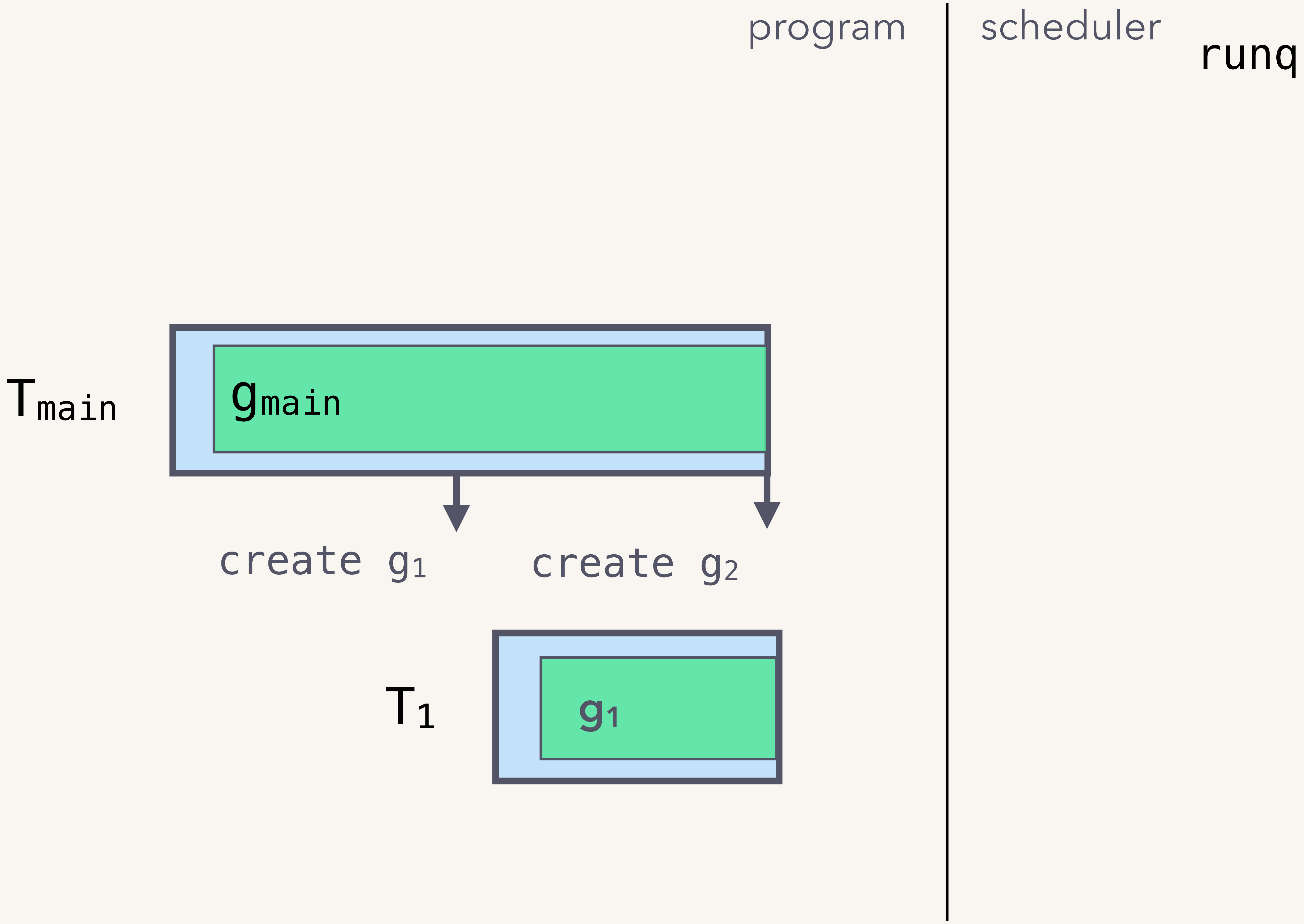
…to what?

too many –> too much contention.

too few -> won't use all the CPU cores, i.e. will give up on parallelism.

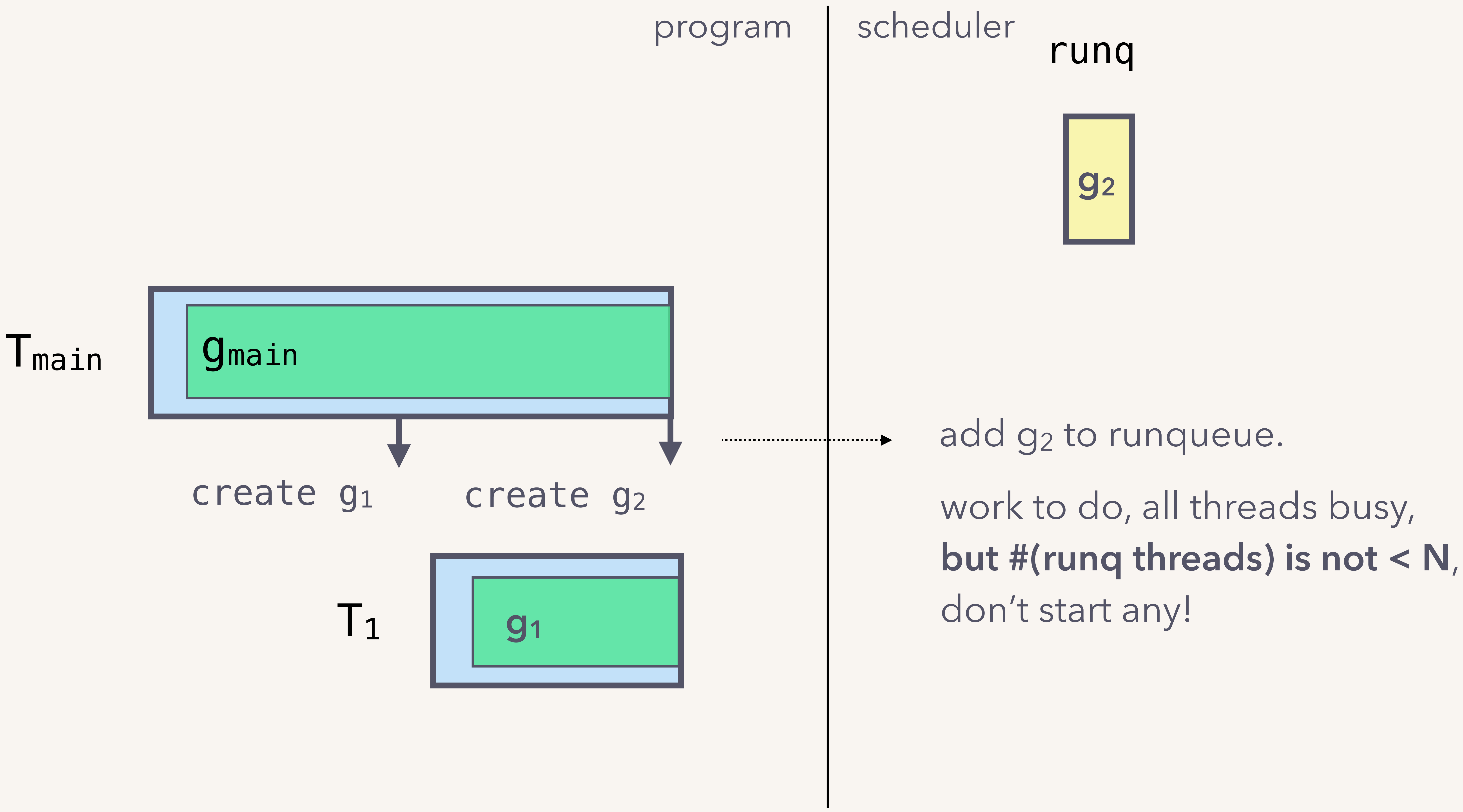
To the **number of CPU cores**, to get **all the parallelism** we can!

Limit # threads accessing runqueue to number of CPU cores (N) = 2.



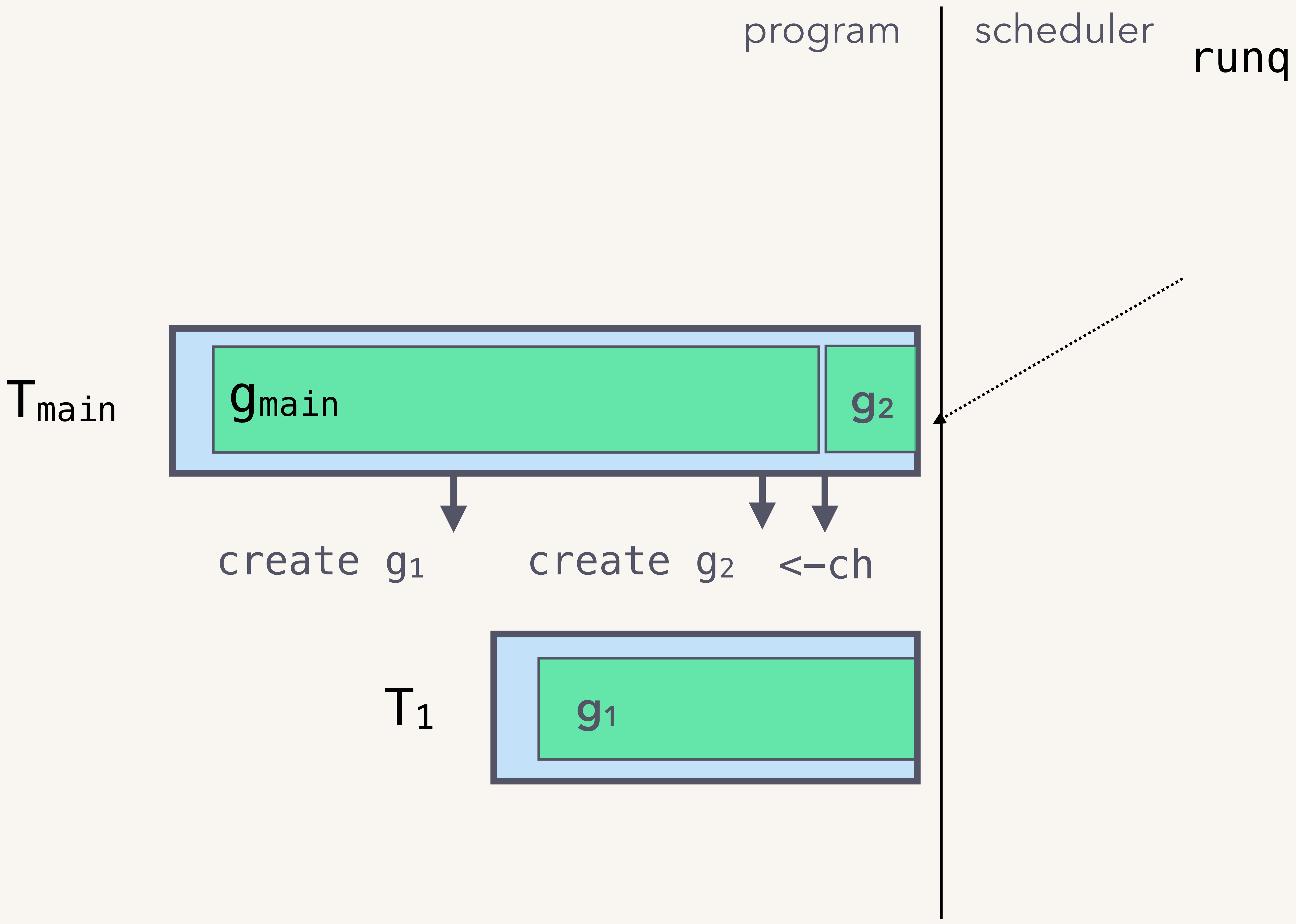
Say T_{main} creates g_2, but g_{main} and g_1 are still running.

Limit # threads accessing runqueue to number of CPU cores (N) = 2.



Say T_{main} creates g_2, but g_{main} and g_1 are still running.

Limit # threads accessing runqueue to number of CPU cores (N) = 2.



g_2 will be run at a future point.

seems reasonable.

We get around unbounded thread contention, without giving up parallelism.

seems reasonable.

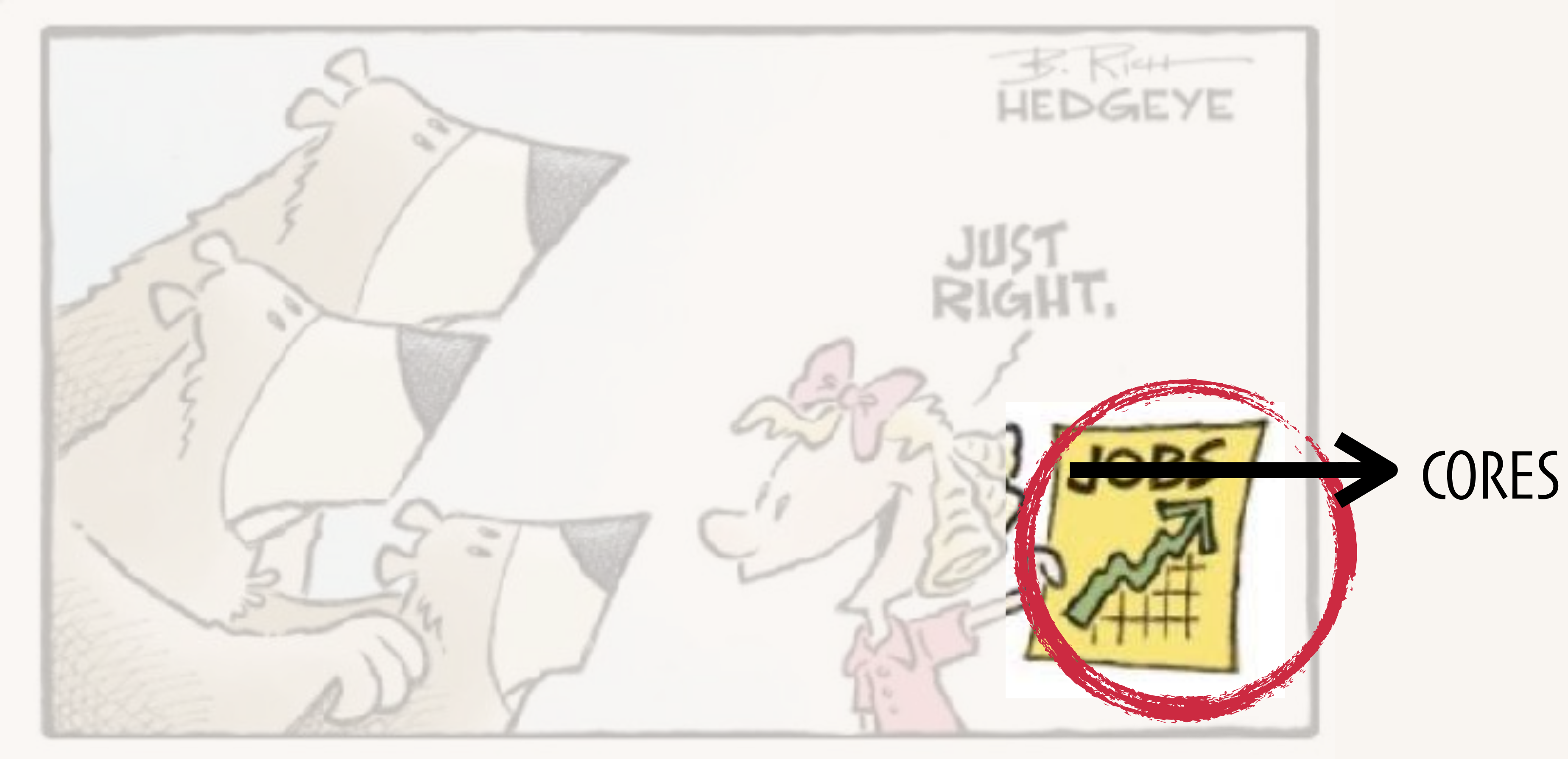
We get around unbounded thread contention, without giving up parallelism.

…ship it?

seems reasonable.

We get around unbounded thread contention, without giving up parallelism.

…ship it?



seems reasonable.

We get around unbounded thread contention, without giving up parallelism.

…ship it?

– This scheme does **not** scale with the number of CPU cores!

As N ↑ ⟶ number of runqueue-accessing threads ↑.

‖

ruh-roh, we're in hella contention land again.

the experiment

the modified Go scheduler:

- uses a global runqueue, and
  #(goroutine-running threads) = #(CPU cores).
- everything else about the runtime is unmodified.

the experiment

the modified Go scheduler:

- uses a global runqueue, and
  #(goroutine-running threads) = #(CPU cores).
- everything else about the runtime is unmodified.

the benchmark:

- `CreateGoroutineParallel`, in the go repo.
- creates #(CPU cores) goroutines in parallel,
  until a total of b.N goroutines have been created.

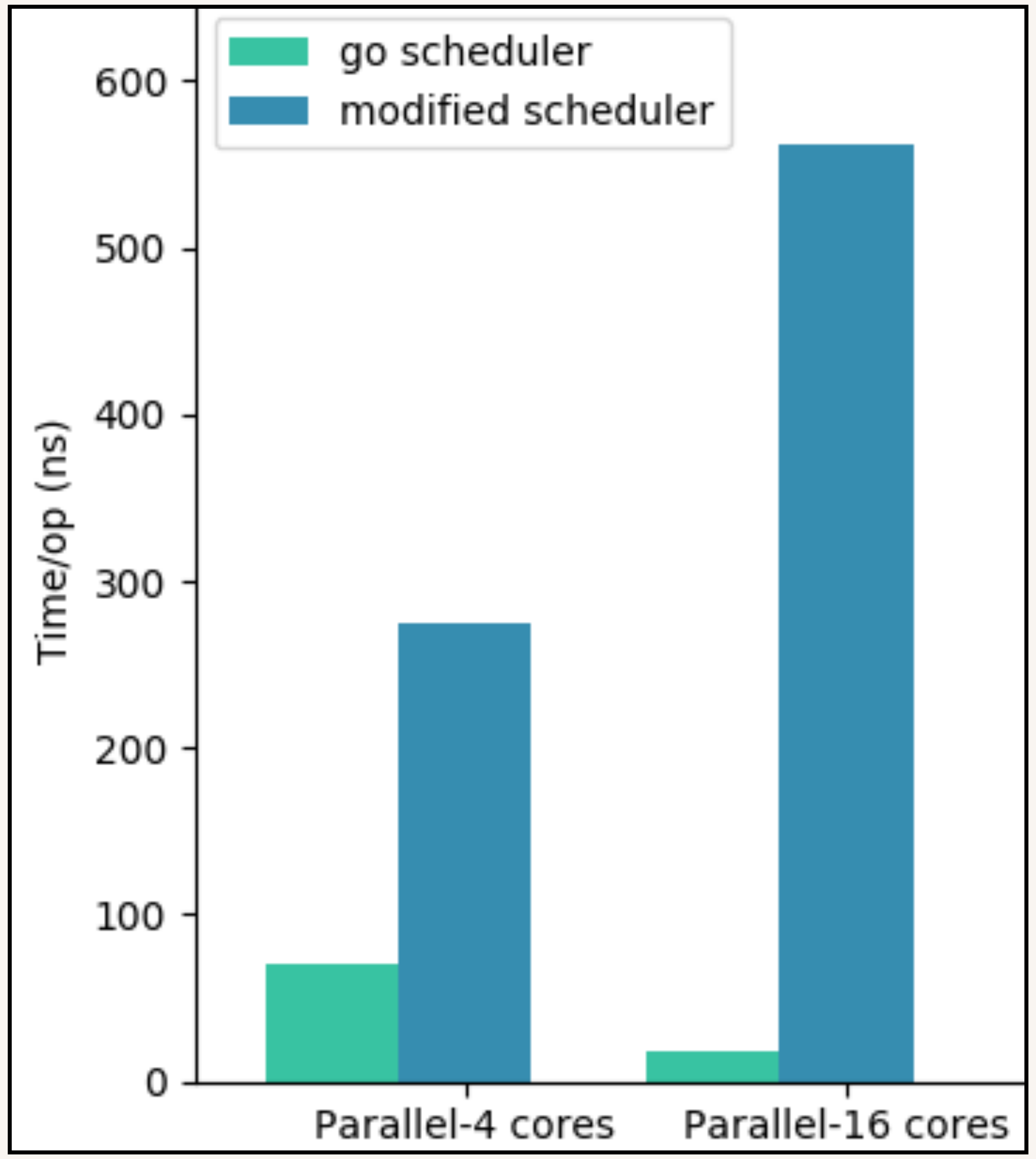
the machines:

A 4-core and 16-core x86-64.

the experiment

On the 4-core:
the modified scheduler takes about 4x longer than the Go scheduler.

On the 16-core:
the modified scheduler takes about <u>31x longer</u> than the Go scheduler!



scheduler benchmarks

(`CreateGoroutineParallel`)

seems reasonable.

We get around unbounded thread contention, without giving up parallelism.

…ship it?

nope.

– This scheme does **not** scale with the number of CPU cores!

As N ↑ ⟶ number of runqueue-accessing threads ↑.

‖

ruh-roh, we're in hella contention land again.

#(goroutine-running threads) = #(CPU cores) is still clever.

we maximally leverage parallelism by this.

really, the problem is the single shared runqueue.

idea III: distributed runqueues

Use **N runqueues** on an N-core machine.

idea III: distributed runqueues

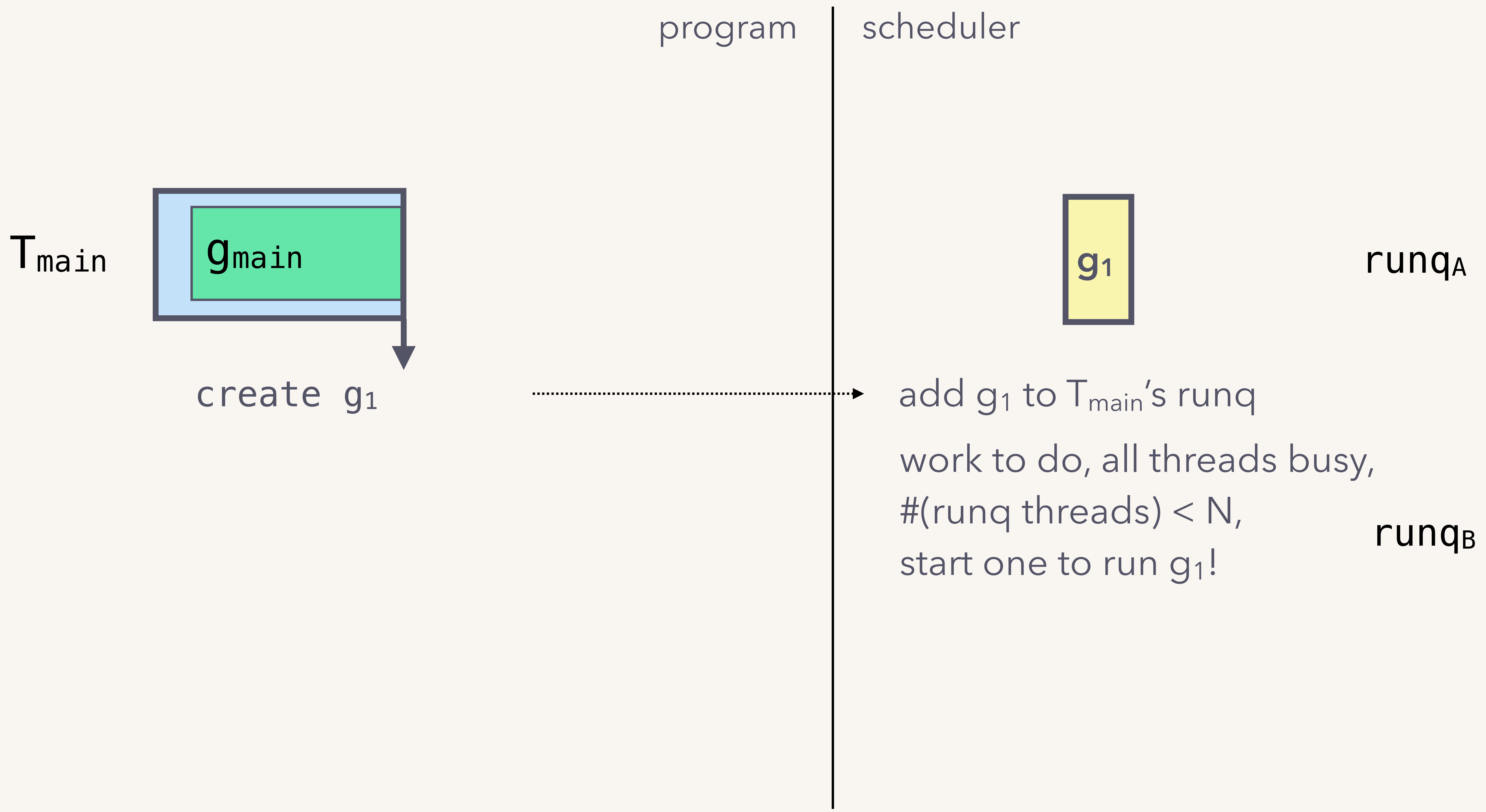
Use **N runqueues** on an N-core machine.

A thread claims a runqueue to run goroutines.

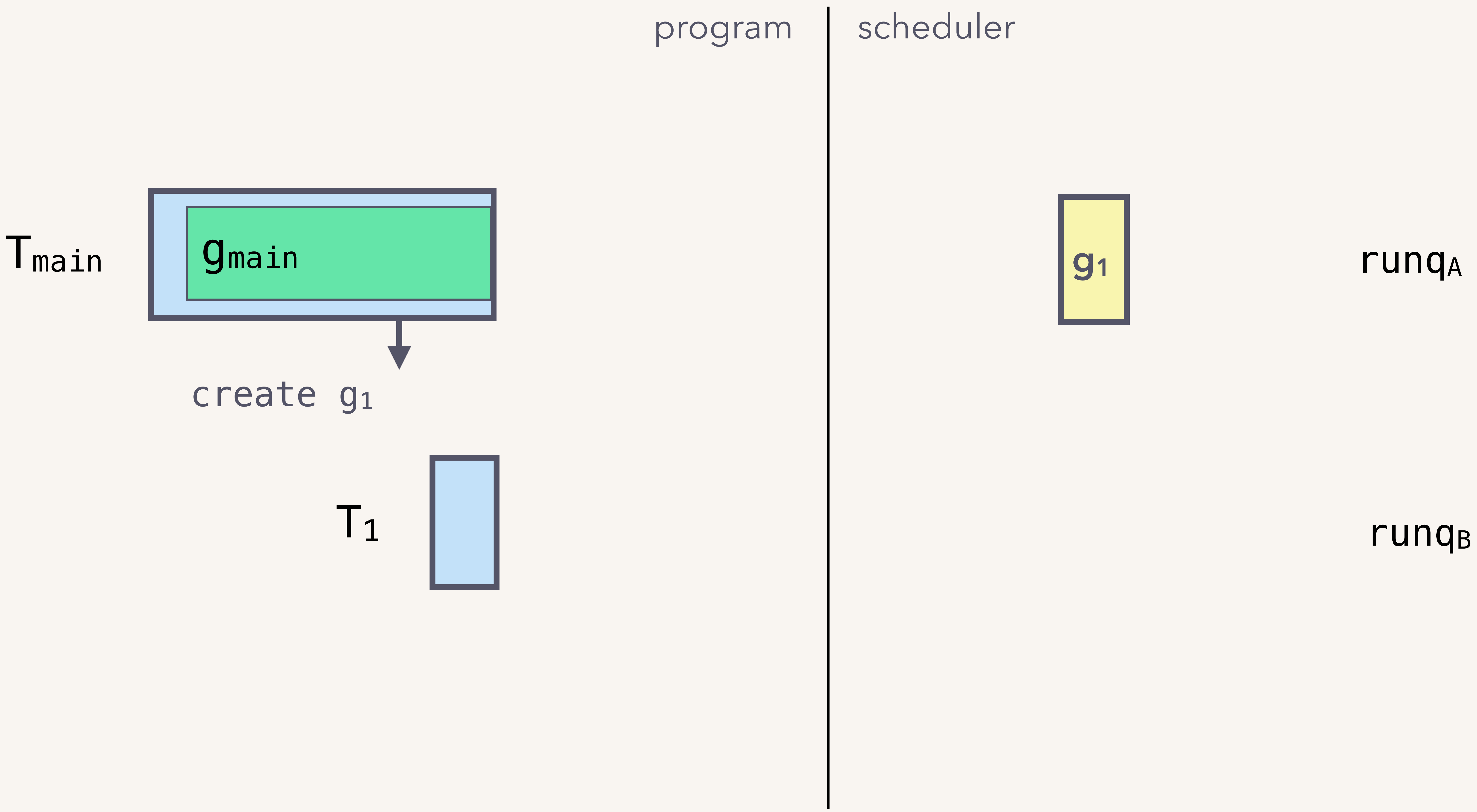
it inserts and removes goroutines
from the runqueue it is **associated with**.

As before, reuse threads.

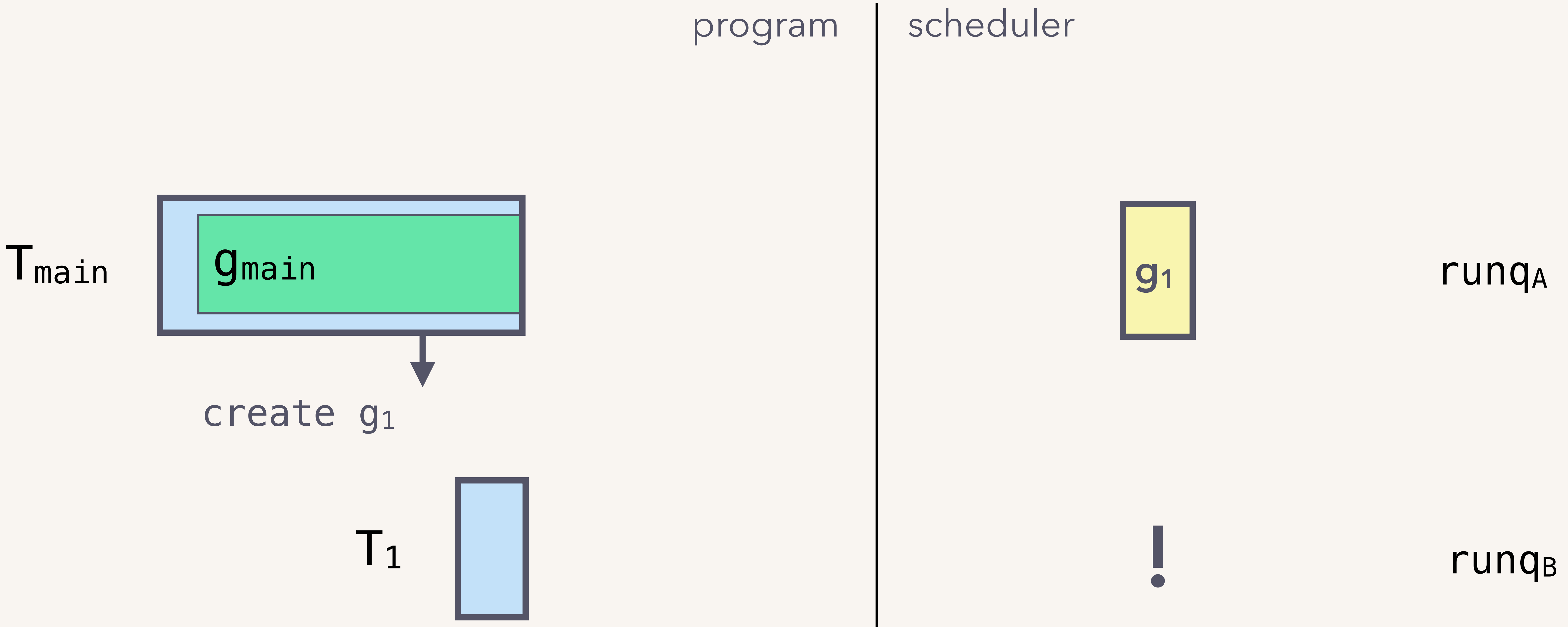
Number of CPU cores (N) = number of runqueues = 2.



Number of CPU cores (N) = number of runqueues = 2.



Number of CPU cores (N) = number of runqueues = 2.



uh oh.

The local runq is empty.

so, steal!

If the local runqueue is empty, **steal work** from another runqueue.

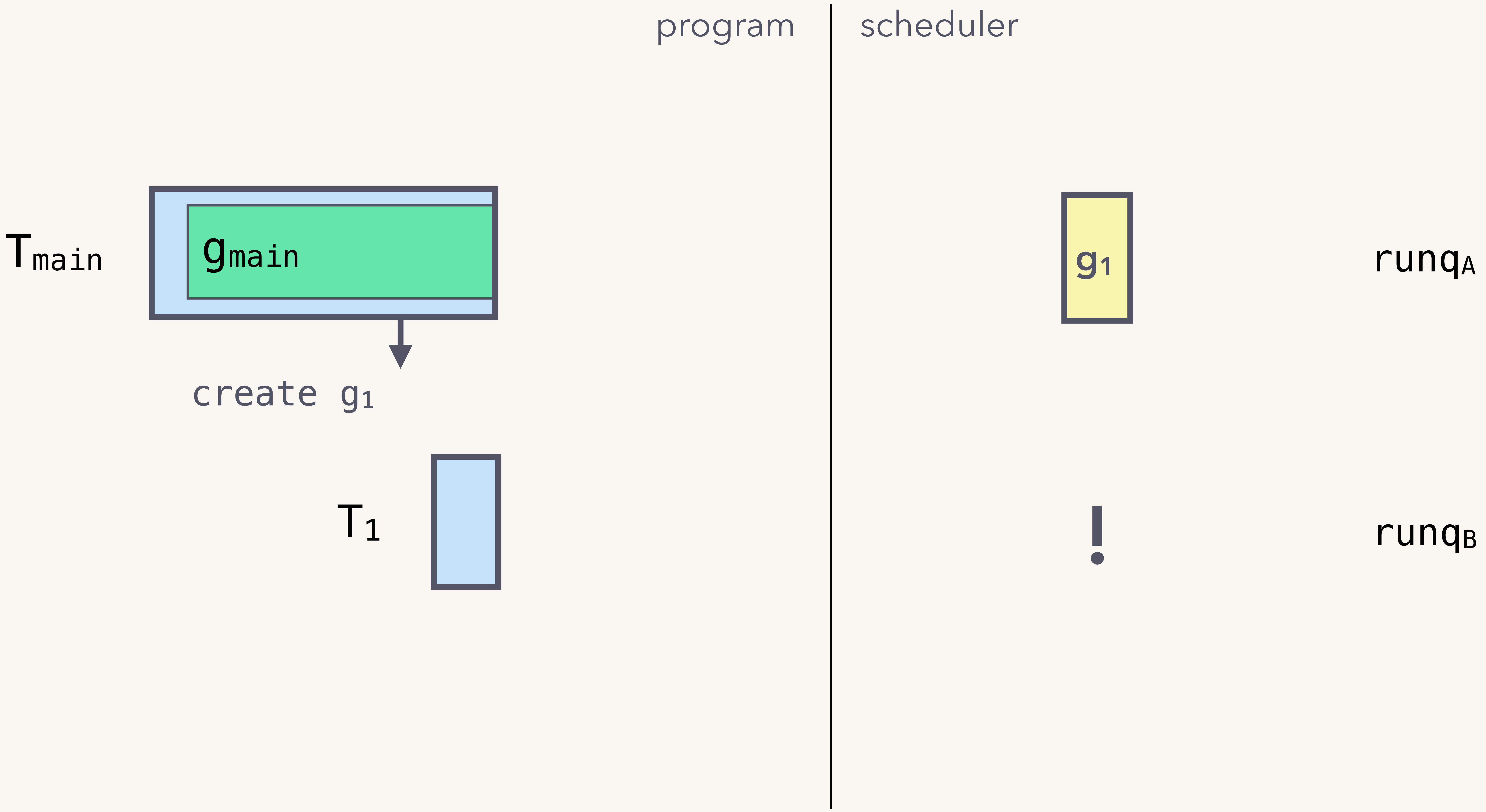
“work stealing”

pick another runqueue at random, steal half its work.

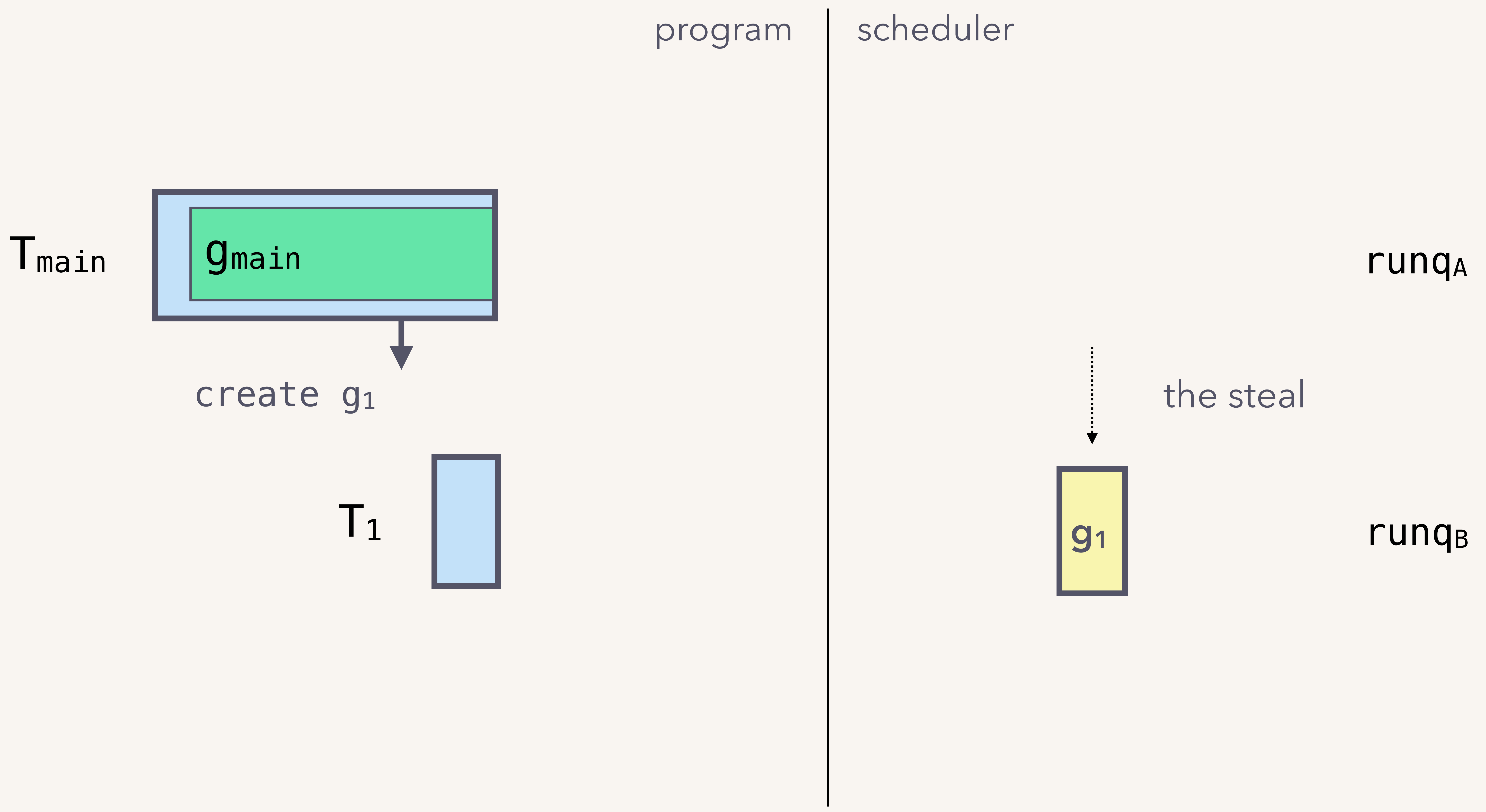
so, steal!

If the local runqueue is empty, **steal work** from another runqueue.
It organically **balances work** across threads.

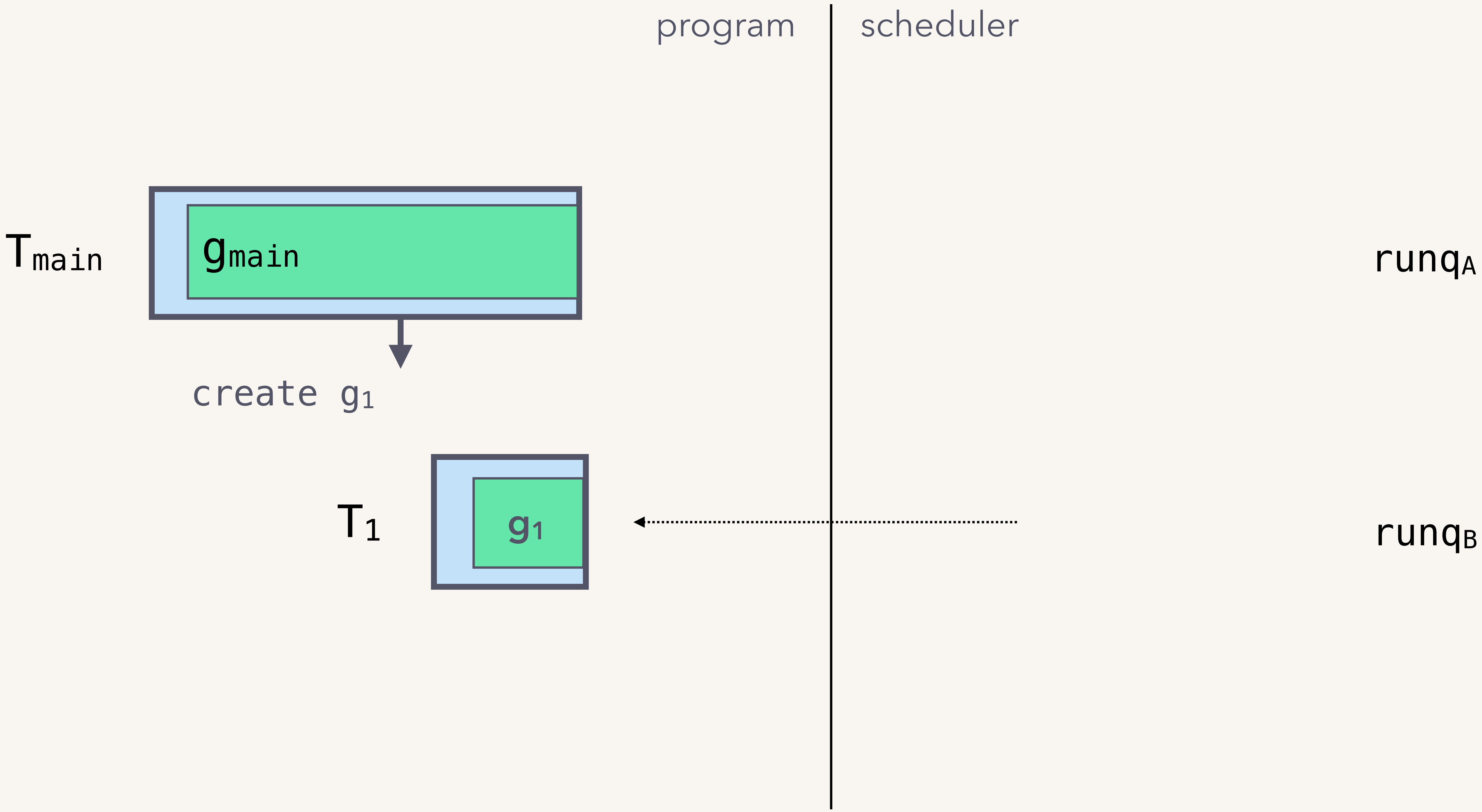
Number of CPU cores (N) = number of runqueues = 2.



Number of CPU cores (N) = number of runqueues = 2.



Number of CPU cores (N) = number of runqueues = 2.



"the end justifies the means"?

this looks promising!

This scheme **scales nicely** with the number of CPU cores, and threads don't contend for work.

The work across threads is <u>balanced</u> with work-stealing.

let's continue.

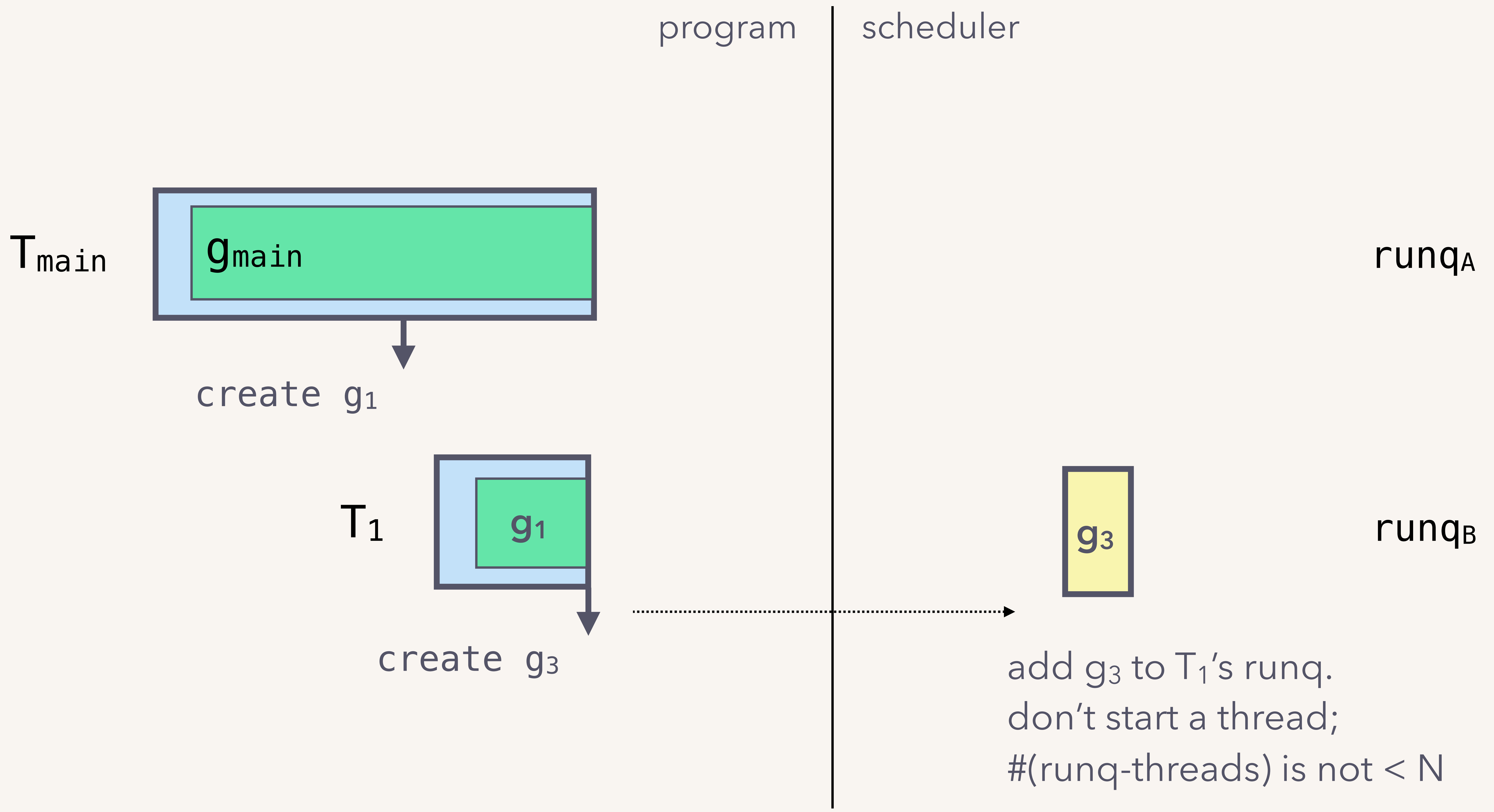
g_1

```
func process(image) {
  // Create goroutine.
  go reportMetrics()

  complicatedAlgorithm(image)

  // Write to file.
  f, err := os.OpenFile()
  ...
}
```

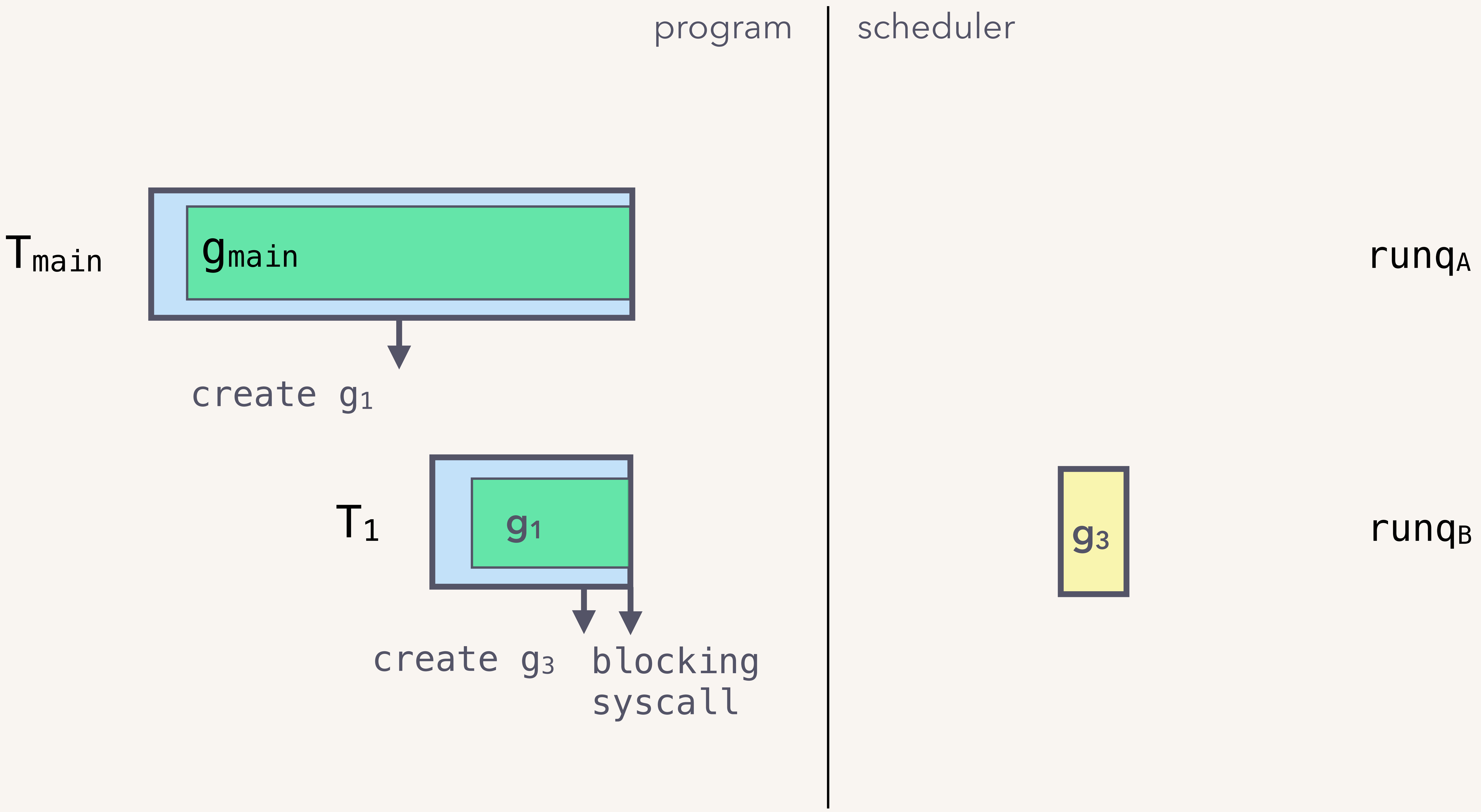
creates g_3

goroutine & thread block

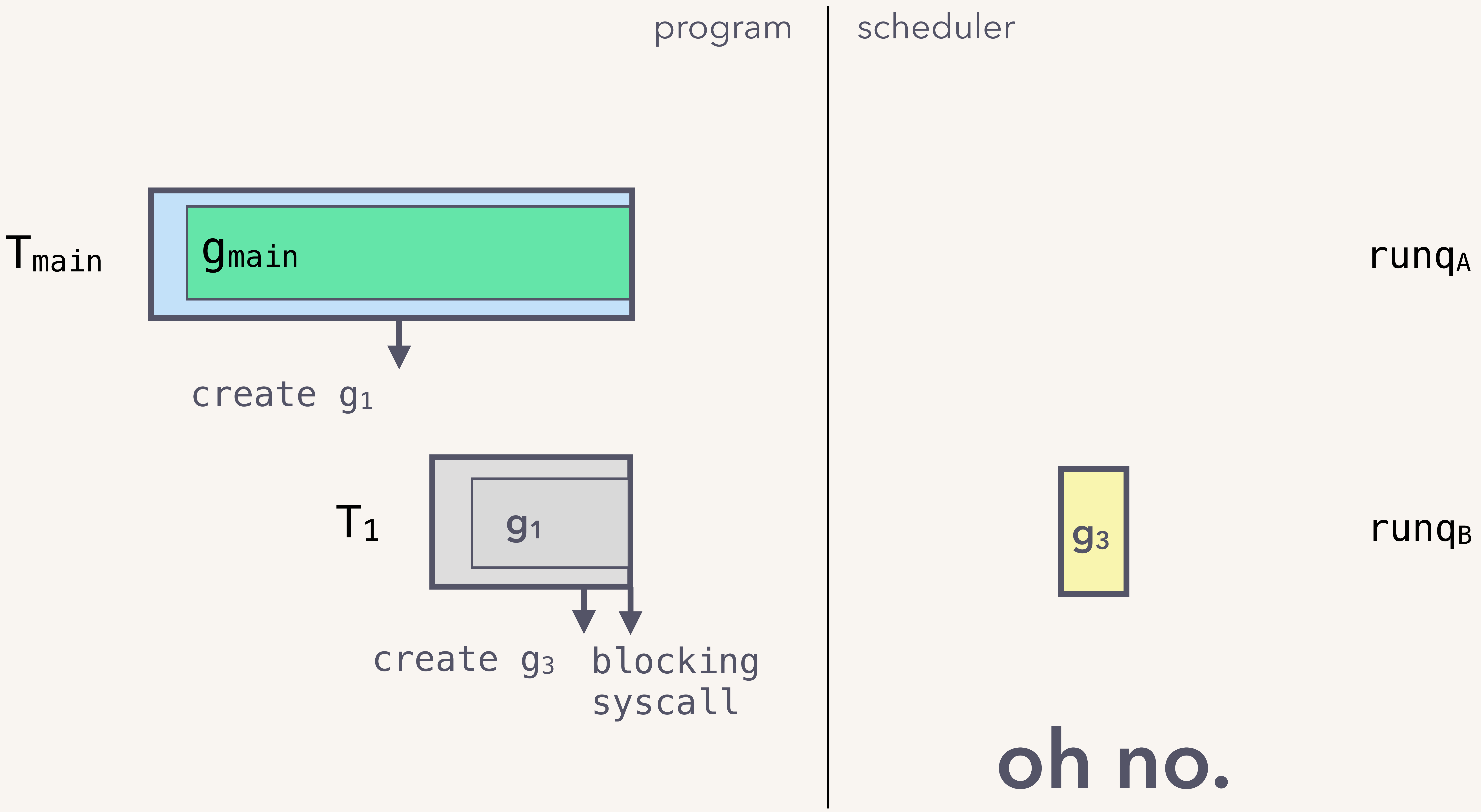
Number of CPU cores (N) = number of runqueues = 2.



Number of CPU cores (N) = number of runqueues = 2.



Number of CPU cores (N) = number of runqueues = 2.



oh no.

The runqueue has work, and the thread's blocked.

"handoff"

Use a mechanism to **transfer a blocked thread's runqueue** to another thread.

a background monitor thread that detects threads blocked for a while, takes and gives the runqueues away.

Why can't the thread itself handoff the runqueue, before it enters the system call?

If it did, it could give up its runqueue unnecessarily!

“handoff”

Use a mechanism to **transfer a blocked thread’s runqueue** to another thread.

Unpark a parked thread or start a thread if needed.

this is okay to do!

The thread limit (= number of CPU cores) applies to goroutine-running threads only.

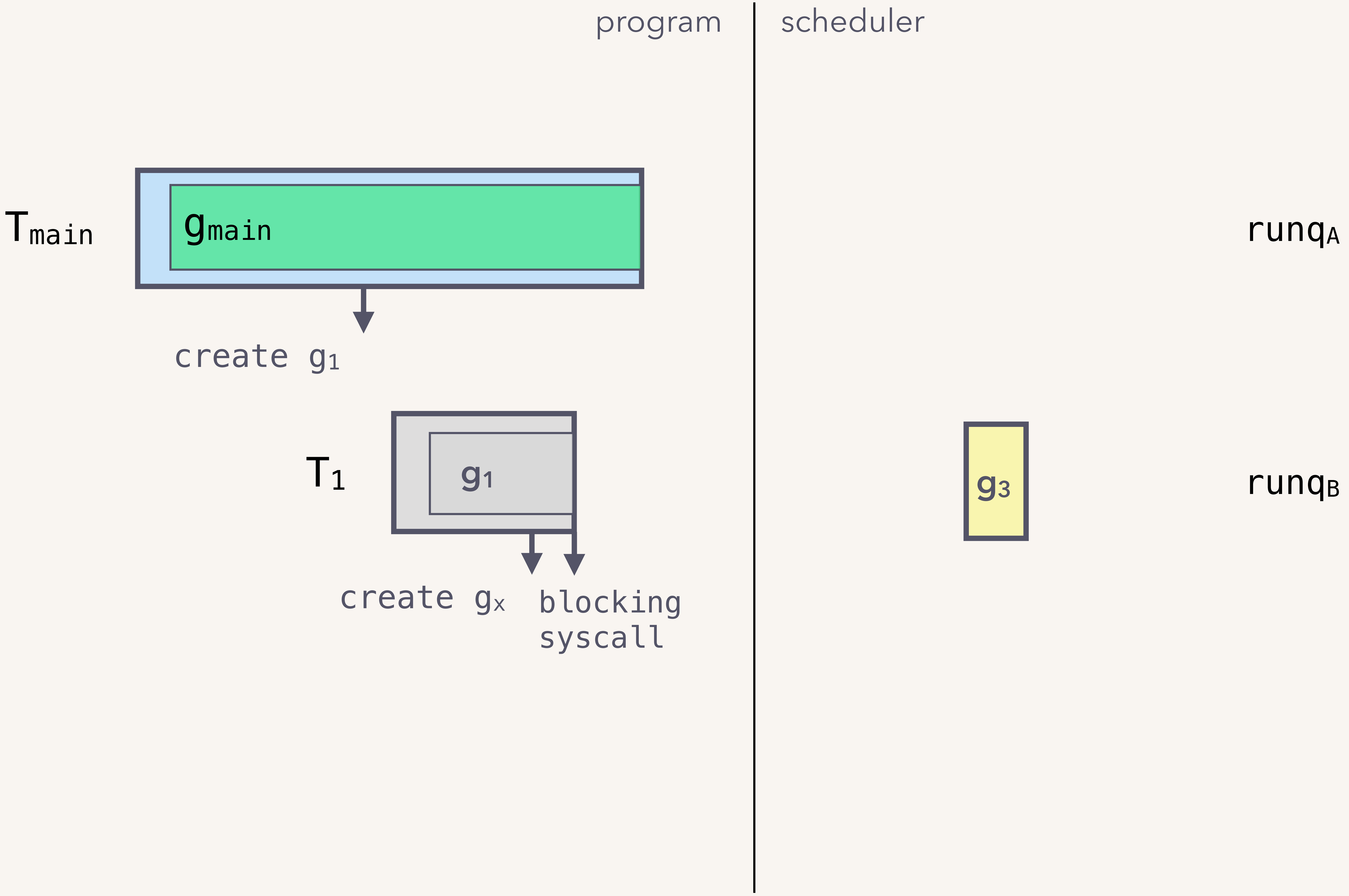
The original thread is blocked; so, another thread can take its place running goroutines.

“handoff”

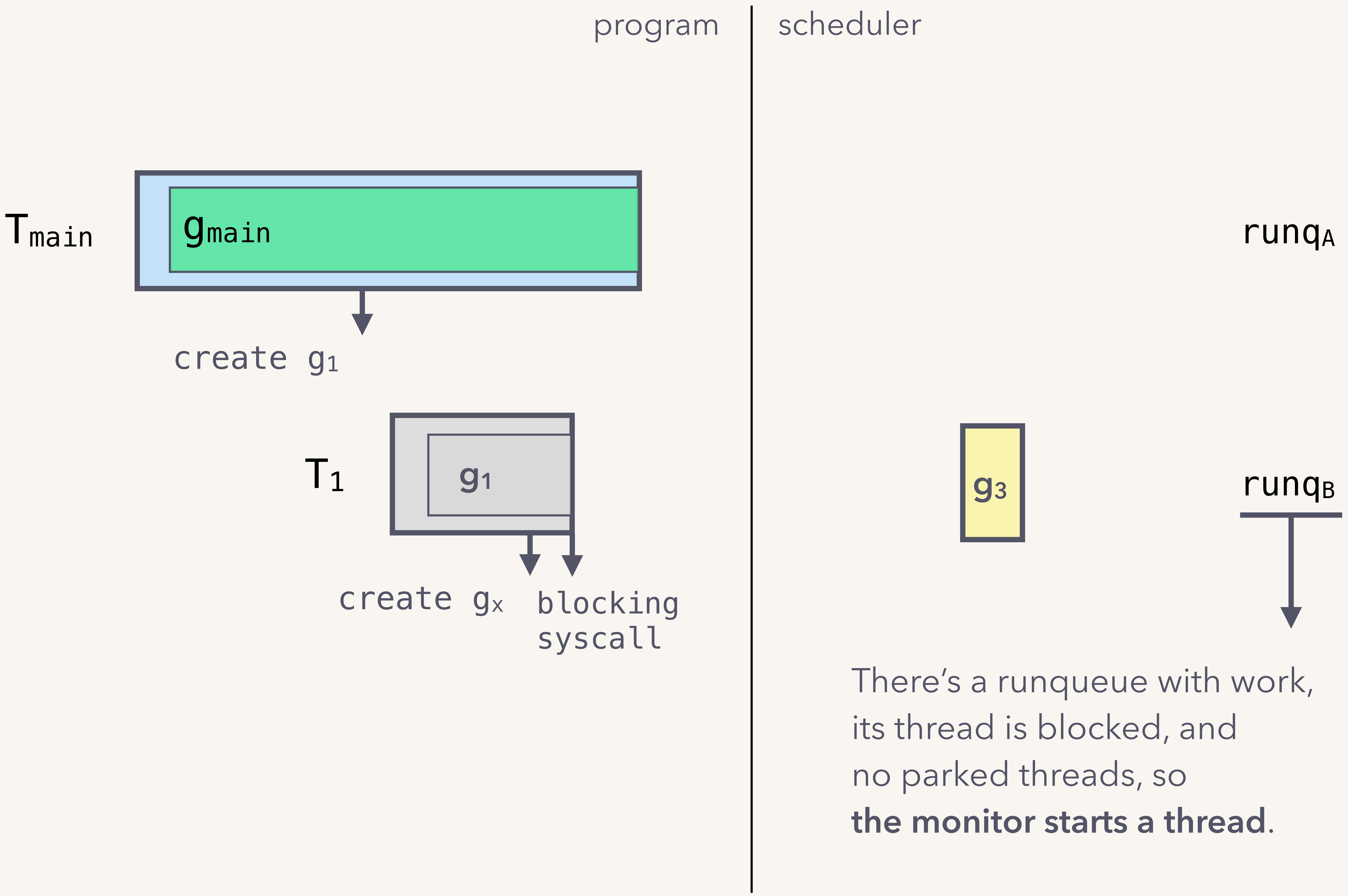
Use a mechanism to **transfer a blocked thread’s runqueue** to another thread.

Prevents **goroutine starvation**.

Number of CPU cores (N) = number of runqueues = 2.

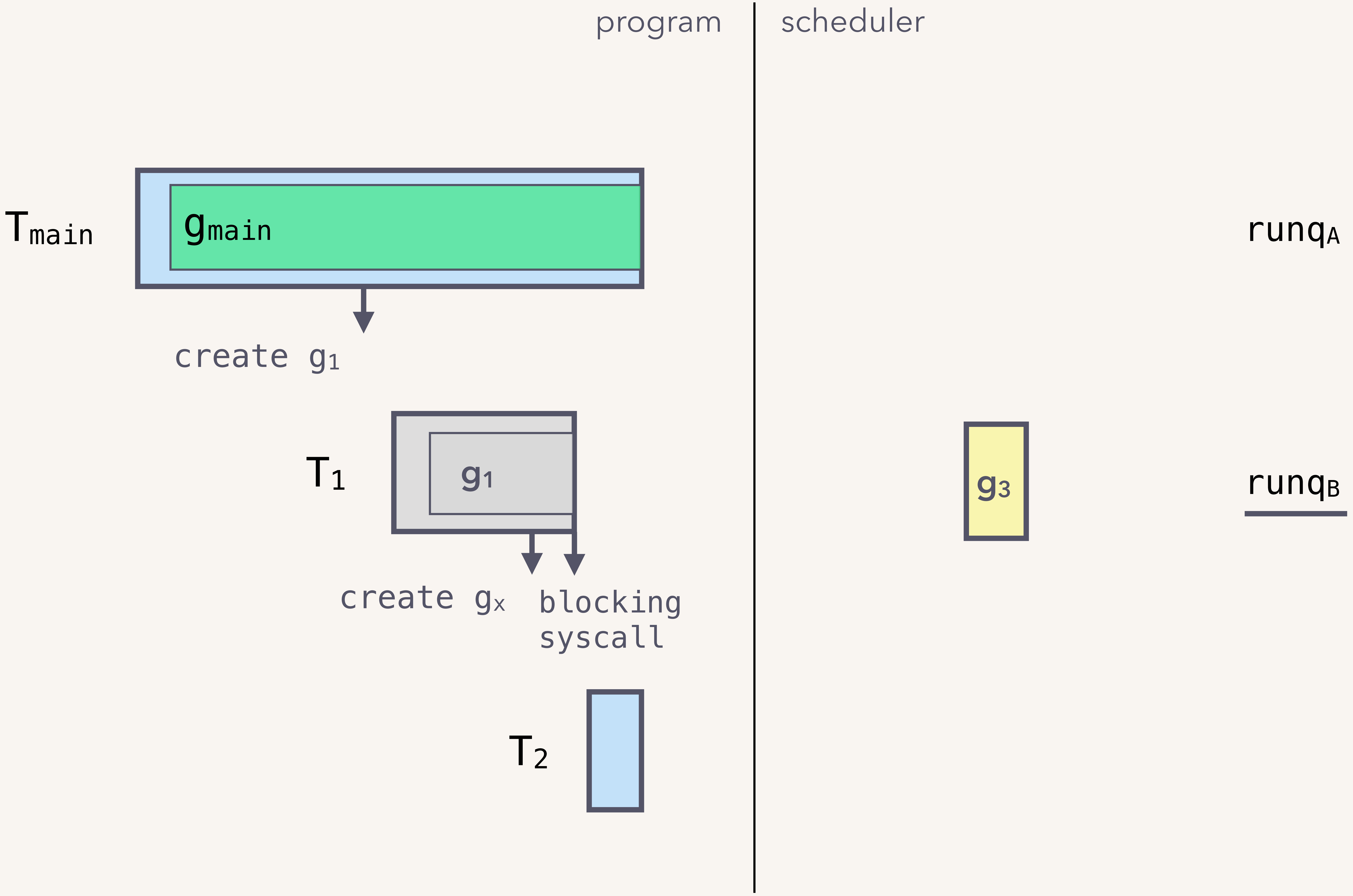


Number of CPU cores (N) = number of runqueues = 2.

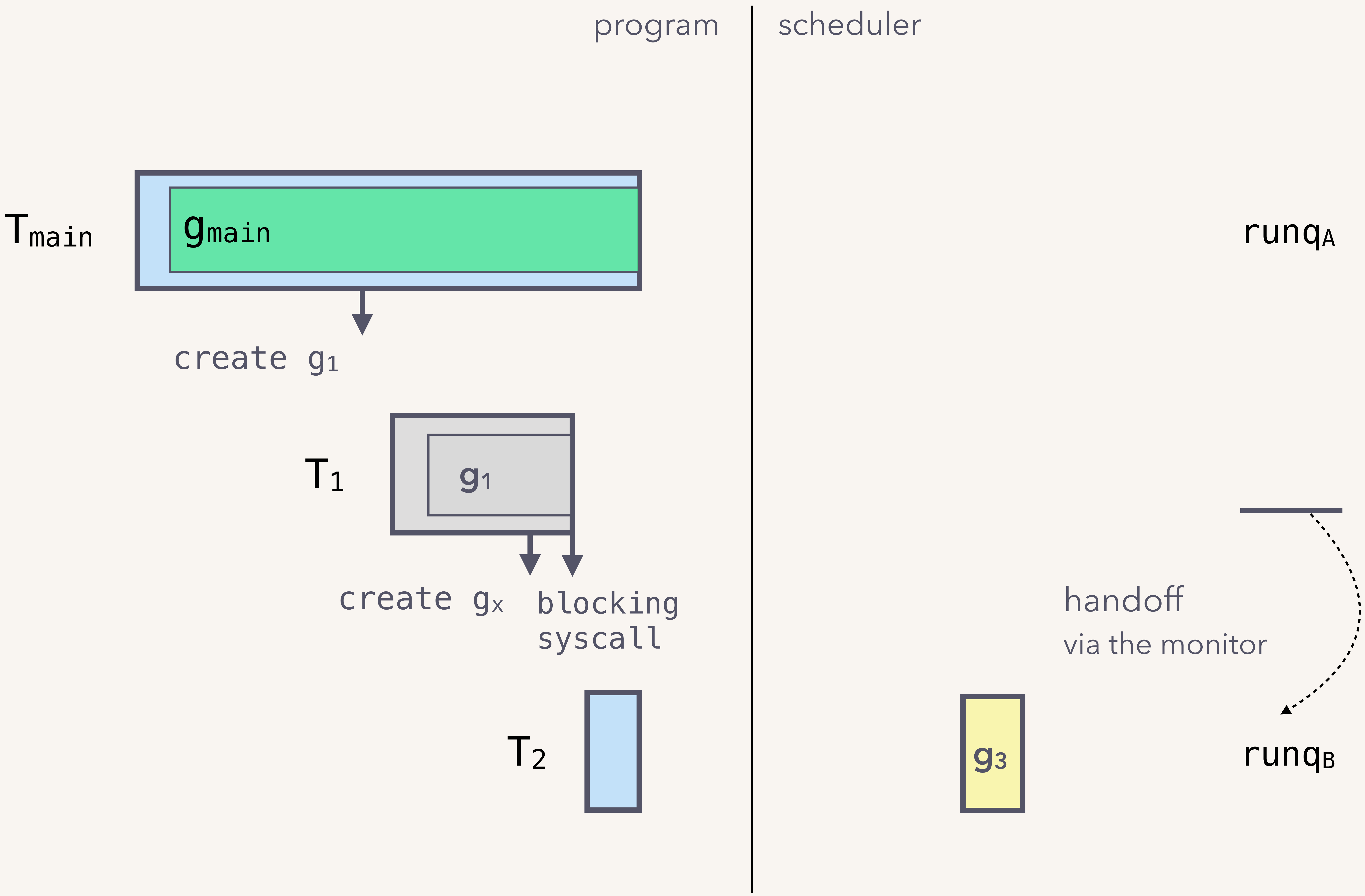


There's a runqueue with work,
its thread is blocked, and
no parked threads, so
the monitor starts a thread.

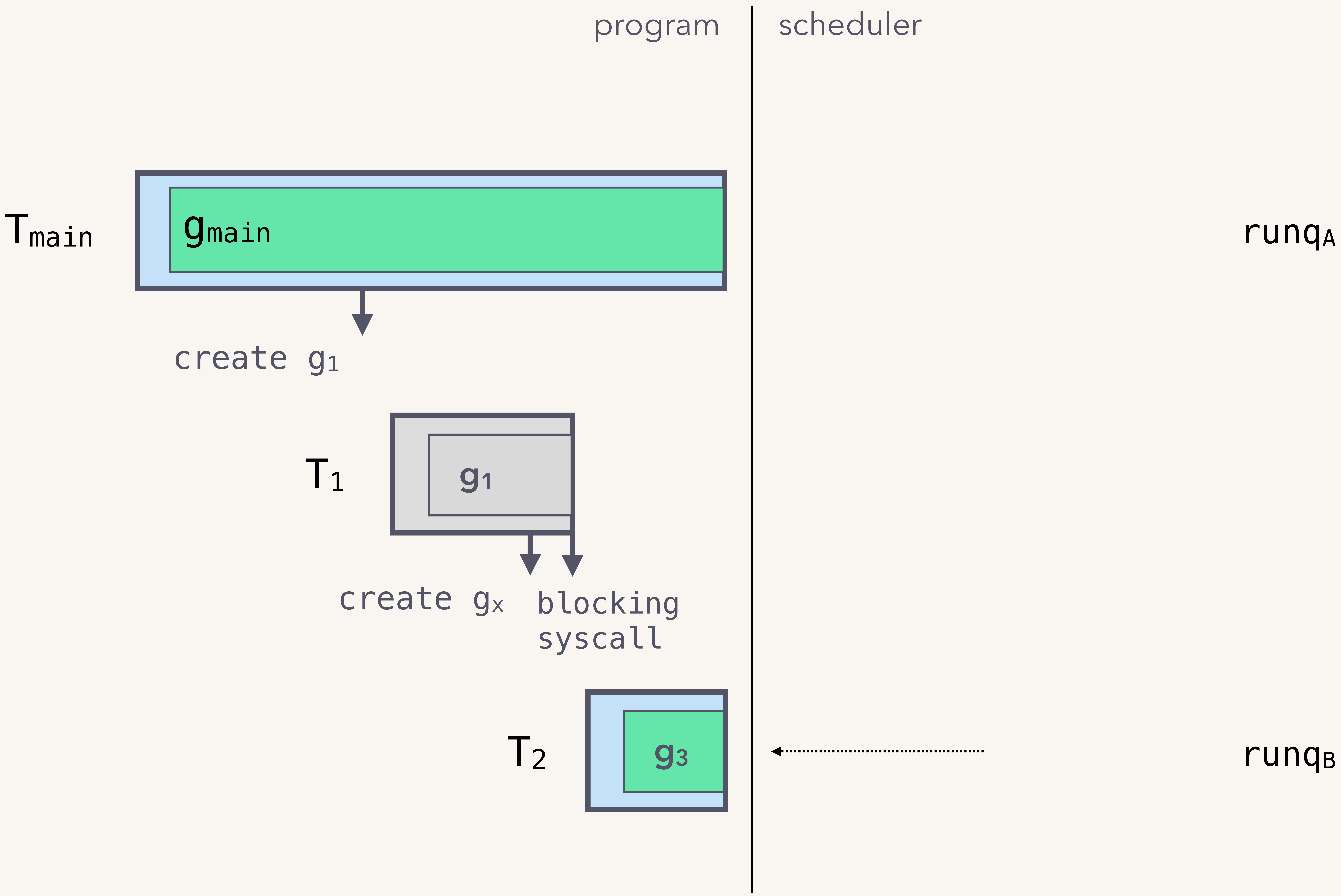
Number of CPU cores (N) = number of runqueues = 2.



Number of CPU cores (N) = number of runqueues = 2.



Number of CPU cores (N) = number of runqueues = 2.



this looks promising!

This scheme **scales nicely** with the number of CPU cores, and threads don't contend for work.

The work across threads is balanced with work-stealing; handoff prevents starvation from blocked threads.

we have (finally) arrived.

the Go scheduler

the big ideas.

reuse threads.

the Go scheduler

the big ideas.

limit #(goroutine-running) threads to
number of CPU cores.
||
GOMAXPROCS

reuse threads.

the Go scheduler

distributed runqueues with stealing and handoff.

the big ideas.

limit #(goroutine-running) threads to
number of CPU cores.

||

GOMAXPROCS

reuse threads.

…and one sneaky idea.

The scheduling points are **cooperative** i.e. <u>the program</u> calls into the scheduler.

```
// A CPU-bound computation that runs
// for a long, long time.

func complicatedAlgorithm(image) {
  // Do not create goroutines, or do
  // anything the blocks at all.
  ...
}
```

ruh-roh.

a CPU-hog can starve runqueues

To avoid this, the Go scheduler implements **preemption***.

It runs a background thread called the "sysmon",
to detect long-running goroutines (> 10ms; with caveats),
and unschedule them when possible.

* technically, cooperative preemption.

To avoid this, the Go scheduler implements **preemption***.

…where would preempted goroutines be put?

They essentially starved other goroutines from running, so don't want to put them back on the per-core runqueues; it would not be fair.

* technically, cooperative preemption.

…on a **global runqueue**.

that’s right.

The Go scheduler has a global runqueue <u>in addition</u> to the distributed runqueues.

…on a **global runqueue**.

It uses this as a **lower priority** runqueue.

Threads access it less frequently than their local runqueues;
so, contention is not a real issue.

a neat detail (or two)…

thread spinning

- Threads without work "spin" looking for work before parking;
they check the global runqueue, poll the network, attempt to run gc tasks, and work-steal.
- This burns CPU cycles, but maximally **leverages available parallelism**.

Ps and runqueues

- The per-core runqueues are stored in a heap-allocated "`p`" struct.
- It stores other resources a thread needs to run goroutines too, like a memory cache.
- A thread claims a `p` to run goroutines, and the entire `p` is handed-off when it's blocked.

Fun fact: this handoff is taken care of by the sysmon too.

assess it.

the difficult questions.

#schedgoals

for scheduling **goroutines** onto **kernel threads**.

- ☑ use a **small number of kernel threads.**

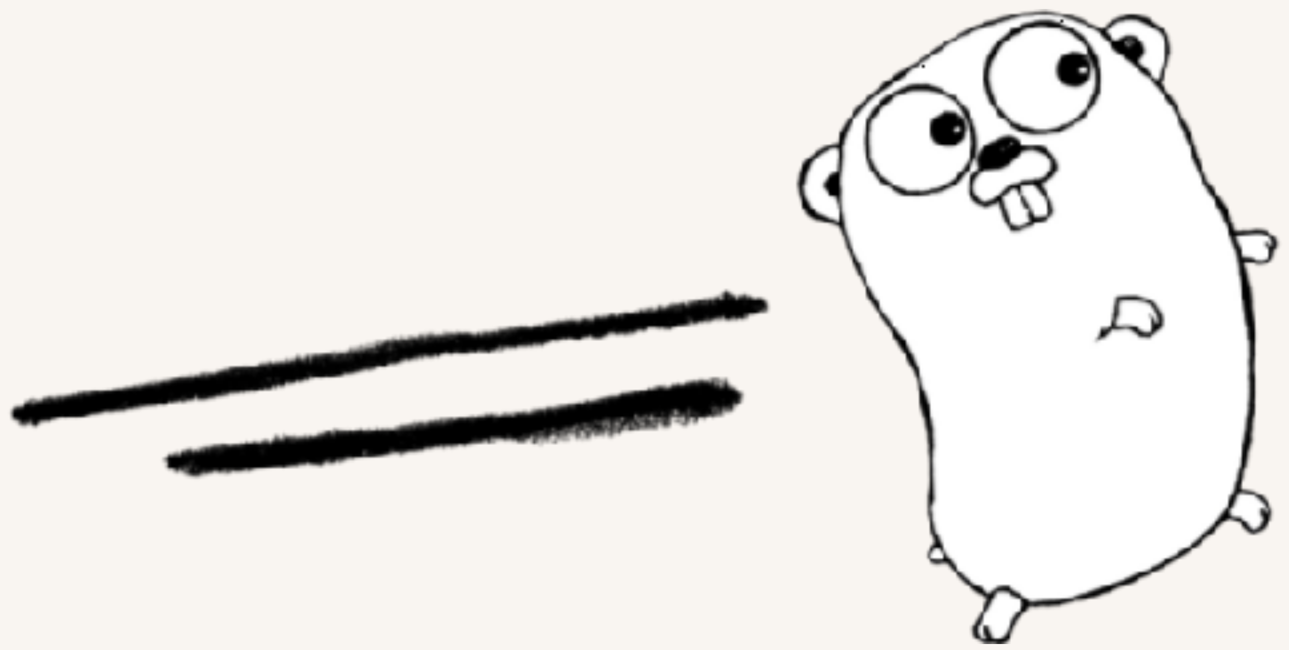
  ideas: reuse threads & limit the number of goroutine-running threads.

- ☑ support high **concurrency.**

  ideas: threads use independent runqueues & keep them balanced.

- ☑ leverage **parallelism i.e. scale to N cores.**

  ideas: use a runqueue per core & employ thread spinning.



limitations

FIFO runqueues → no notion of goroutine priorities.

Implement runqueues as priority queues, like the Linux scheduler.

No strong preemption → no strong fairness or latency guarantees.

recent proposal to fix this: Non-cooperative goroutine preemption.

Is not aware of the system topology → no real locality.

dated proposal to fix this: NUMA-aware scheduler

Use LIFO, rather than FIFO, runqueues; better for cache utilization.

The Go scheduler motto, in a picture.

References

Scalable scheduler design doc

https://github.com/golang/go/blob/master/src/runtime/runtime2.go

https://github.com/golang/go/blob/master/src/runtime/proc.go

Go scheduler blog post

Scheduling Multithreaded Computations by Work Stealing

@kavya719

speakerdeck.com/kavya719/the-scheduler-saga

Special thanks to Eben Freeman and Austin Duffield for reading drafts of this, & also Chris Frost, Bernardo Farah, Anubhav Jain and Jeffrey Chen.