Runtime Scheduling: Theory and Reality

Eben Freeman

Strange Loop 2016

Hi everyone!

Why talk about scheduling?

"Most modern servers can handle hundreds of small, active threads or processes simultaneously, but performance degrades seriously once memory is exhausted or **when high I/O load causes a large volume of context switches**."

https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/

"In this connection thread model, there are as many threads as there are clients currently connected, which has some disadvantages when server workload must scale to handle large numbers of connections. [...] Exhaustion of other resources can occur as well, and **scheduling overhead can become significant**.

https://dev.mysql.com/doc/refman/5.7/en/connection-threads.html

"Because OS threads are scheduled by the kernel, passing control from one thread to another requires a full context switch [...]. This operation is slow, due its poor locality and the number of memory accesses required.

[...]

Because it doesn't need a switch to kernel context, **rescheduling a goroutine is much cheaper than rescheduling a thread**."

Donovan & Kernighan, *The Go Programming Language*

So scheduling (multiplexing a lot of tasks onto few processors)

- can affect our programs' performance
- is kind of a black box.

Questions!

How expensive *is* a context switch?

How does the Linux kernel scheduler work, anyways?

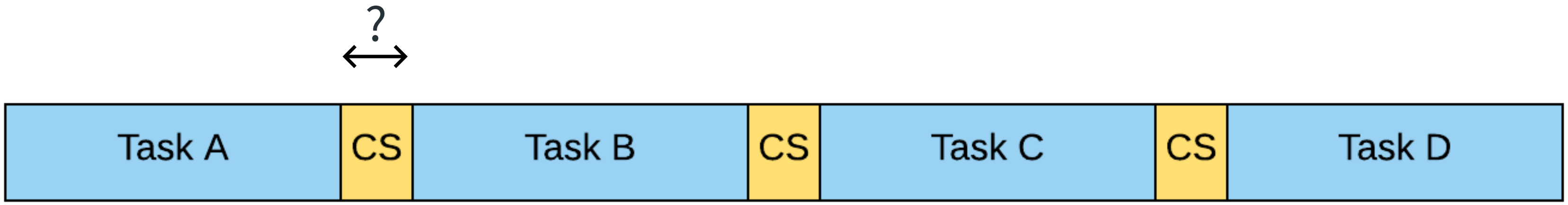
What about userspace schedulers? Are they radically different?

What design patterns do scheduler implementations follow?

What tradeoffs do they make?

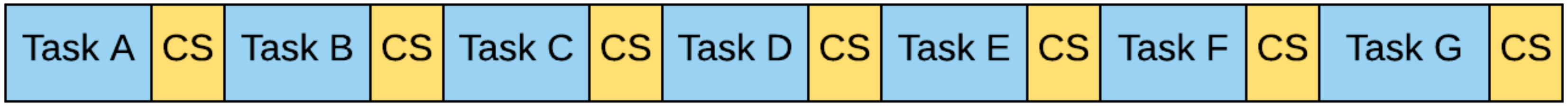
Scheduling in Kernel Space

Estimating kernel context-switch cost

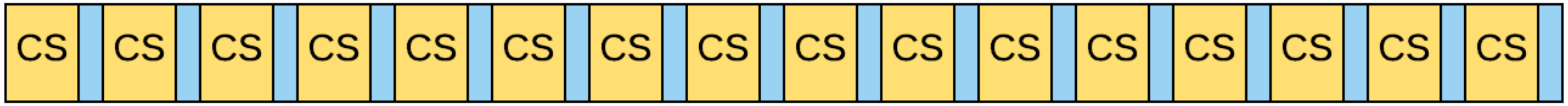


A heuristic for "how much concurrency can our system support?"

Maybe okay:

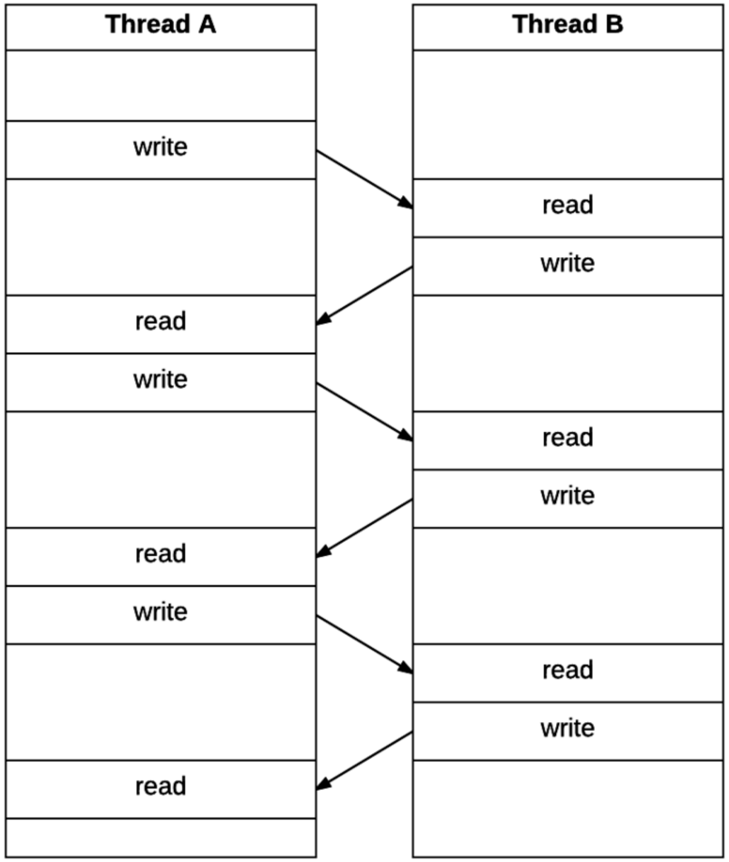


Probably not okay:



Estimating kernel context-switch cost

One classical approach: ping-pong over two pipes




```c
// linux/tools/perf/bench/sched-pipe.c

void *worker_thread(void *data) {
  struct thread_data *td = data;
  int m = 0;

  for (int i = 0; i < loops; i++) {
    if (!td->nr) {
      read(td->pipe_read, &m, sizeof(int));
      write(td->pipe_write, &m, sizeof(int));
    } else {
      write(td->pipe_write, &m, sizeof(int));
      read(td->pipe_read, &m, sizeof(int));
    }
  }

  return NULL;
}
```

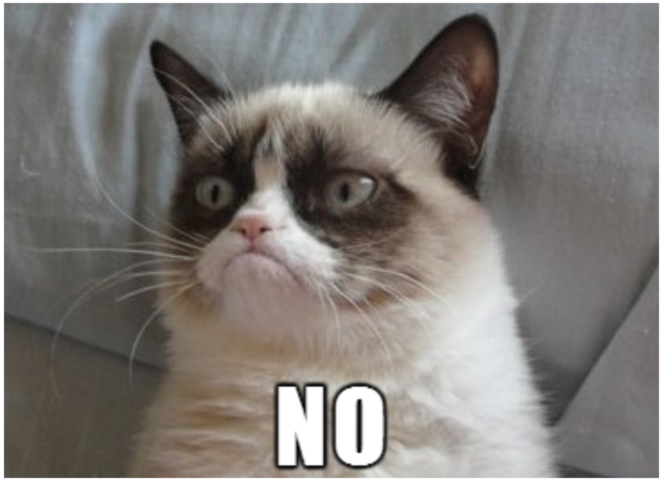
Conveniently, this is part of the *perf bench* suite in Linux:

```
➜  ~ perf bench sched pipe -T
# Running 'sched/pipe' benchmark:
# Executed 1000000 pipe operations between two threads

     Total time: 4.498 [sec]

       4.498076 usecs/op
         222317 ops/sec
```

⇒ upper bound: 2.25 µs per thread context switch

(2 context switches per read-write "op")

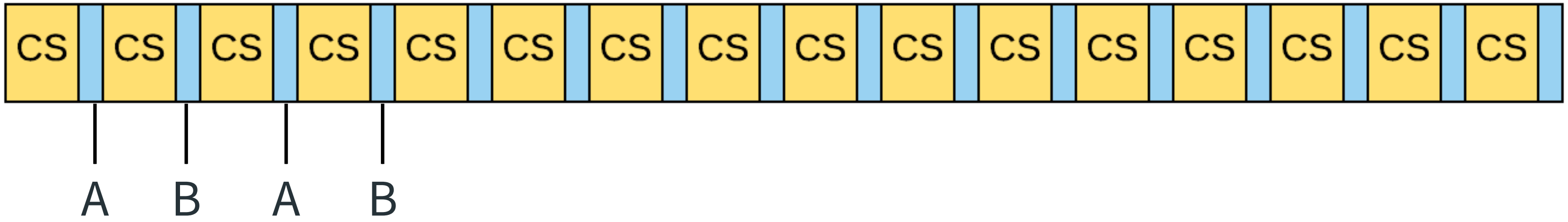
Is that our final answer?


NO

A performance haiku

Can't trust a benchmark
if you don't analyze it
while it is running.

This is our mental model of what's happening:



How well does it map to reality?

perf sched

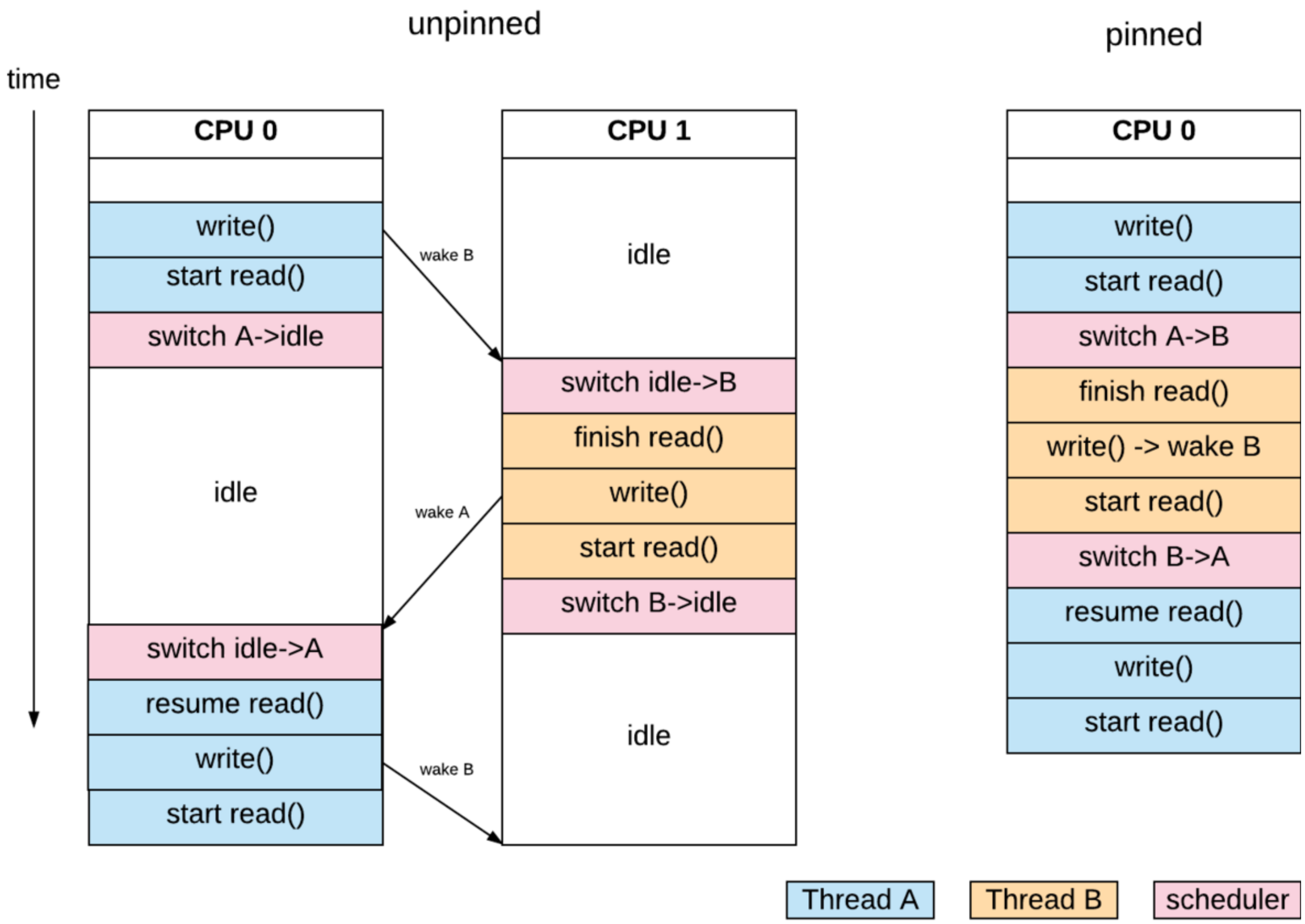
- One of many *perf* subcommands
- Records scheduler events
- Can show context switches, wakeup latency, etc.

```
➜  ~ perf sched record -- perf bench sched pipe -T
➜  ~ perf sched script
      .
      .
CPU   timestamp     event
[000] 98914.958984: sched:sched_stat_runtime: comm=sched-pipe pid=13128 runtime=3045
[000] 98914.958984: sched:sched_switch: sched-pipe:13128 [120] S ==> swapper/0:0 [12
[001] 98914.958985: sched:sched_wakeup: sched-pipe:13128 [120] success=1 CPU:000
[000] 98914.958986: sched:sched_switch: swapper/0:0 [120] R ==> sched-pipe:13128 [12
[001] 98914.958986: sched:sched_stat_runtime: comm=sched-pipe pid=13127 runtime=3010
[001] 98914.958986: sched:sched_switch: sched-pipe:13127 [120] S ==> swapper/1:0 [12
[000] 98914.958987: sched:sched_wakeup: sched-pipe:13127 [120] success=1 CPU:001
[001] 98914.958988: sched:sched_switch: swapper/3:0 [120] R ==> sched-pipe:13127 [12
[000] 98914.958988: sched:sched_stat_runtime: comm=sched-pipe pid=13128 runtime=3020
[000] 98914.958988: sched:sched_switch: sched-pipe:13128 [120] S ==> swapper/0:0 [ns
[001] 98914.958989: sched:sched_wakeup: sched-pipe:13128 [120] success=1 CPU:
[000] 98914.958990: sched:sched_switch: swapper/0:0 [120] R ==> sched-pipe:13128 [ns
[001] 98914.958990: sched:sched_stat_runtime: comm=sched-pipe pid=13127 runtime=2964
      .
      .
```

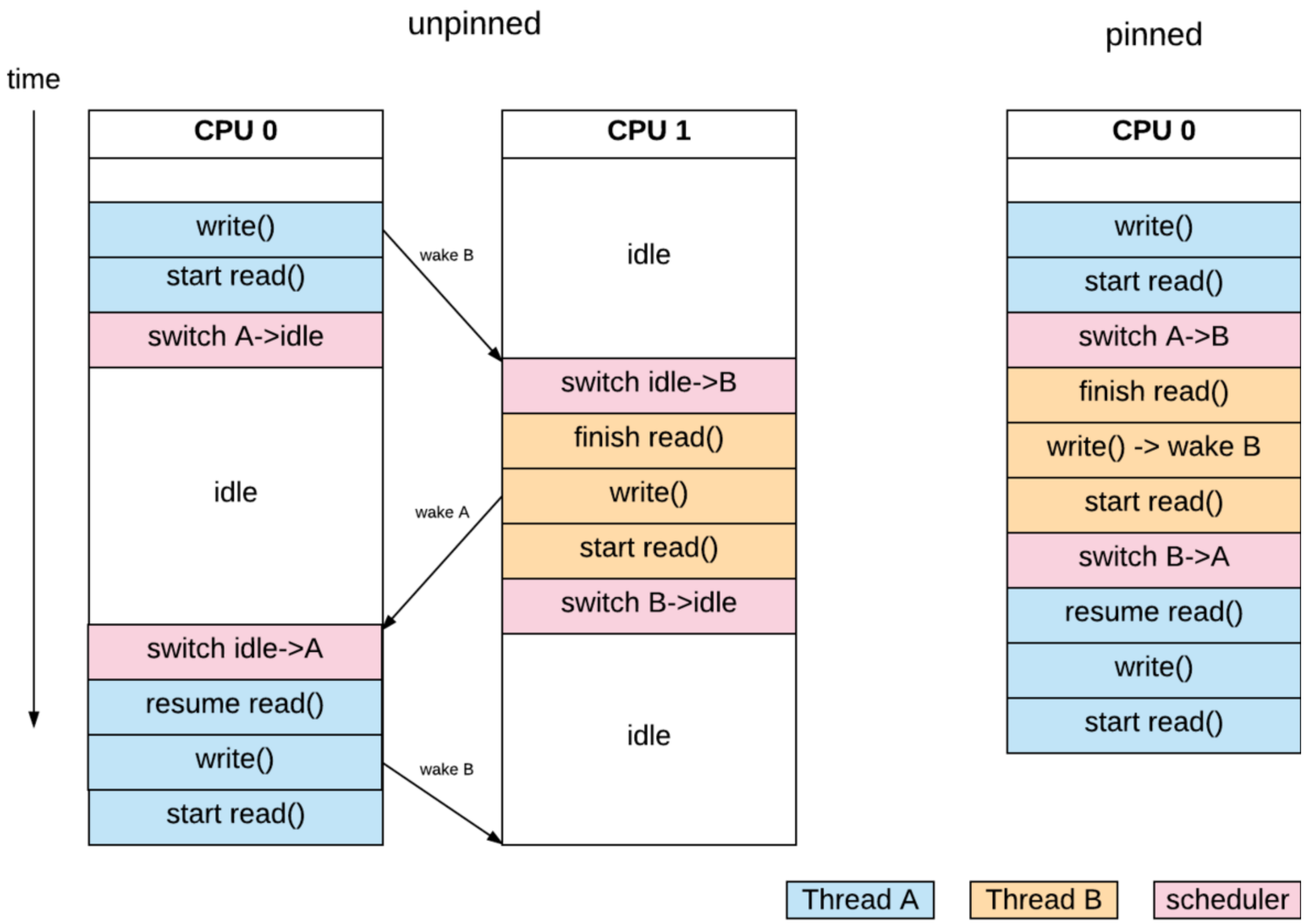
Our pipe tasks are alternating with the "swapper" (idle) process on separate CPUs, not with each other

```
➜  ~ perf sched record -- perf bench sched pipe -T
➜  ~ perf sched script
        .
        .
CPU    timestamp       event
[000] 98914.958984: sched:sched_stat_runtime: comm=sched-pipe pid=13128 runtime=304
[000] 98914.958984: sched:sched_switch: sched-pipe:13128 [120] S ==> swapper/0:0 [1
[001] 98914.958985: sched:sched_wakeup: sched-pipe:13128 [120] success=1 CPU:000
[000] 98914.958986: sched:sched_switch: swapper/0:0 [120] R ==> sched-pipe:13128 [1
[001] 98914.958986: sched:sched_stat_runtime: comm=sched-pipe pid=13127 runtime=301
[001] 98914.958986: sched:sched_switch: sched-pipe:13127 [120] S ==> swapper/1:0 [1
[000] 98914.958987: sched:sched_wakeup: sched-pipe:13127 [120] success=1 CPU:001
[001] 98914.958988: sched:sched_switch: swapper/3:0 [120] R ==> sched-pipe:13127 [1
[000] 98914.958988: sched:sched_stat_runtime: comm=sched-pipe pid=13128 runtime=302
[000] 98914.958988: sched:sched_switch: sched-pipe:13128 [120] S ==> swapper/0:0 [n
[001] 98914.958989: sched:sched_wakeup: sched-pipe:13128 [120] success=1 CPU:
[000] 98914.958990: sched:sched_switch: swapper/0:0 [120] R ==> sched-pipe:13128 [n
[001] 98914.958990: sched:sched_stat_runtime: comm=sched-pipe pid=13127 runtime=296
        .
        .
```

Let's draw a picture.



When threads are scheduled on separate cores, cross-core wakeup adds overhead.



Let's run that benchmark *slightly* differently...

```
➜  ~ perf bench sched pipe -T
# Running 'sched/pipe' benchmark:
# Executed 1000000 pipe operations between two threads

     Total time: 4.498 [sec]

       4.498076 usecs/op
         222317 ops/sec


➜  ~ taskset -c 0 perf bench sched pipe -T
# Running 'sched/pipe' benchmark:
# Executed 1000000 pipe operations between two threads

     Total time: 1.935 [sec]

       1.935758 usecs/op
         516593 ops/sec
```

pin tasks to core 0

~2x difference

```
➜  ~ perf bench sched pipe -T
# Running 'sched/pipe' benchmark:
# Executed 1000000 pipe operations between two threads

     Total time: 4.498 [sec]

       4.498076 usecs/op
         222317 ops/sec


➜  ~ taskset -c 0 perf bench sched pipe -T
# Running 'sched/pipe' benchmark:
# Executed 1000000 pipe operations between two threads

     Total time: 1.935 [sec]

       1.935758 usecs/op
         516593 ops/sec
```

What did we learn?

- The direct cost of a thread context switch is around 1 microsecond (on this machine, with caveats, etc.)

Meta-lessons

- Benchmarking is tricky.
- Can't just run random experiments -- need introspection into scheduler
- Helpful to have some idea how the scheduler works!

The Linux kernel scheduler

Required features:

- Preemption (misbehaving tasks cannot block system)
- Prioritization (important tasks first)

Okay, we've got this!

We'll keep a list of running tasks

```
struct task_struct* init_task;
struct task_struct * task[512] = {&init_task, };

void schedule(void)
{
  int c;
  struct task_struct *p, *next;
  c = -1000;
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      break;
    if (p->state == TASK_RUNNING && p->counter > c
      c = p->counter, next = p;
  }
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority
  }
  if (current == next)
    return;
  switch_to(next);
}
```

We'll keep a list of running tasks

And when we need to schedule

```
struct task_struct* init_task;
struct task_struct * task[512] = {&init_task, };

void schedule(void)
{
  int c;
  struct task_struct *p, *next;
  c = -1000;
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      break;
    if (p->state == TASK_RUNNING && p->counter > c
      c = p->counter, next = p;
  }
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority
  }
  if (current == next)
    return;
  switch_to(next);
}
```

We'll keep a list of running tasks

And when we need to schedule

Iterate through our tasks

```
struct task_struct* init_task;
struct task_struct * task[512] = {&init_task, };

void schedule(void)
{
  int c;
  struct task_struct *p, *next;
  c = -1000;
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      break;
    if (p->state == TASK_RUNNING && p->counter > c
      c = p->counter, next = p;
  }
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority
  }
  if (current == next)
    return;
  switch_to(next);
}
```

We'll keep a list of running tasks

And when we need to schedule

Iterate through our tasks

Keep a countdown for each task

Pick the task with the lowest countdown to run next

```
struct task_struct* init_task;
struct task_struct * task[512] = {&init_task, };

void schedule(void)
{
  int c;
  struct task_struct *p, *next;
  c = -1000;
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      break;
    if (p->state == TASK_RUNNING && p->counter > c
      c = p->counter, next = p;
  }
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority
  }
  if (current == next)
    return;
  switch_to(next);
}
```

We'll keep a list of running tasks

And when we need to schedule

Iterate through our tasks

Keep a countdown for each task

Pick the task with the lowest countdown to run next

Decrement countdown for each task (hi-pri tasks count down faster)

```
struct task_struct* init_task;
struct task_struct * task[512] = {&init_task, };

void schedule(void)
{
  int c;
  struct task_struct *p, *next;
  c = -1000;
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      break;
    if (p->state == TASK_RUNNING && p->counter > c)
      c = p->counter, next = p;
  }
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority;
  }
  if (current == next)
    return;
  switch_to(next);
}
```

We'll keep a list of running tasks

And when we need to schedule

Iterate through our tasks

Keep a countdown for each task

Pick the task with the lowest countdown to run next

Decrement countdown for each task (hi-pri tasks count down faster)

Then switch to the next task

```
struct task_struct* init_task;
struct task_struct * task[512] = {&init_task, };

void schedule(void)
{
  int c;
  struct task_struct *p, *next;
  c = -1000;
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      break;
    if (p->state == TASK_RUNNING && p->counter > c
      c = p->counter, next = p;
  }
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority
  }
  if (current == next)
    return;
  switch_to(next);
}
```

This is how the Linux scheduler worked in 1995.

```c
struct task_struct* init_task;
struct task_struct * task[512] = {&init_task, };

void schedule(void)
{
  int c;
  struct task_struct *p, *next;
  c = -1000;
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      break;
    if (p->state == TASK_RUNNING && p->counter > c
      c = p->counter, next = p;
  }
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority
  }
  if (current == next)
    return;
  switch_to(next);
}
```

(okay, it was ~75 lines)

```c
struct task_struct* init_task;
struct task_struct * task[512] = {&init_task, };

void schedule(void)
{
  int c;
  struct task_struct *p, *next;
  c = -1000;
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      break;
    if (p->state == TASK_RUNNING && p->counter > c)
      c = p->counter, next = p;
  }
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority;
  }
  if (current == next)
    return;
  switch_to(next);
}
```

```c
asmlinkage void schedule(void)
{
  int c;
  struct task_struct * p;
  struct task_struct * next;
  unsigned long ticks;

/* check alarm, wake up any interruptible tasks that have got a signal */

  if (intr_count) {
    printk("Aiee: scheduling in interrupt\n");
    intr_count = 0;
  }
  cli();
  ticks = itimer_ticks;
  itimer_ticks = 0;
  itimer_next = ~0;
  sti();
  need_resched = 0;
  p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      goto confuse_gcc1;
    if (ticks && p->it_real_value) {
      if (p->it_real_value <= ticks) {
        send_sig(SIGALRM, p, 1);
        if (!p->it_real_incr) {
          p->it_real_value = 0;
          goto end_itimer;
        }
        do {
          p->it_real_value += p->it_real_incr;
        } while (p->it_real_value <= ticks);
      }
      p->it_real_value -= ticks;
      if (p->it_real_value < itimer_next)
        itimer_next = p->it_real_value;
    }
end_itimer:
    if (p->state != TASK_INTERRUPTIBLE)
      continue;
    if (p->signal & ~p->blocked) {
      p->state = TASK_RUNNING;
      continue;
    }
    if (p->timeout && p->timeout <= jiffies) {
      p->timeout = 0;
      p->state = TASK_RUNNING;
    }
  }
confuse_gcc1:

/* this is the scheduler proper: */
#if 0
  /* give processes that go to sleep a bit higher priority.. */
  /* This depends on the values for TASK_XXX */
  /* This gives smoother scheduling for some things, but */
  /* can be very unfair under some circumstances, so.. */
   if (TASK_UNINTERRUPTIBLE >= (unsigned) current->state &&
      current->counter < current->priority*2) {
    ++current->counter;
  }
#endif
  c = -1000;
  next = p = &init_task;
  for (;;) {
    if ((p = p->next_task) == &init_task)
      goto confuse_gcc2;
    if (p->state == TASK_RUNNING && p->counter > c)
      c = p->counter, next = p;
  }
confuse_gcc2:
  if (!c) {
    for_each_task(p)
      p->counter = (p->counter >> 1) + p->priority;
  }
  if (current == next)
    return;
  kstat.context_swtch++;
  switch_to(next);
}
```

Today, there are a lot more requirements:

- Preemption
- Prioritization
- Fairness
- Multicore scalability
- Power efficiency
- Resource constraints (cgroups)
- etc.

The completely fair scheduler

- In general, scheduling happens on a per-core basis (more about inter-CPU load balancing later).

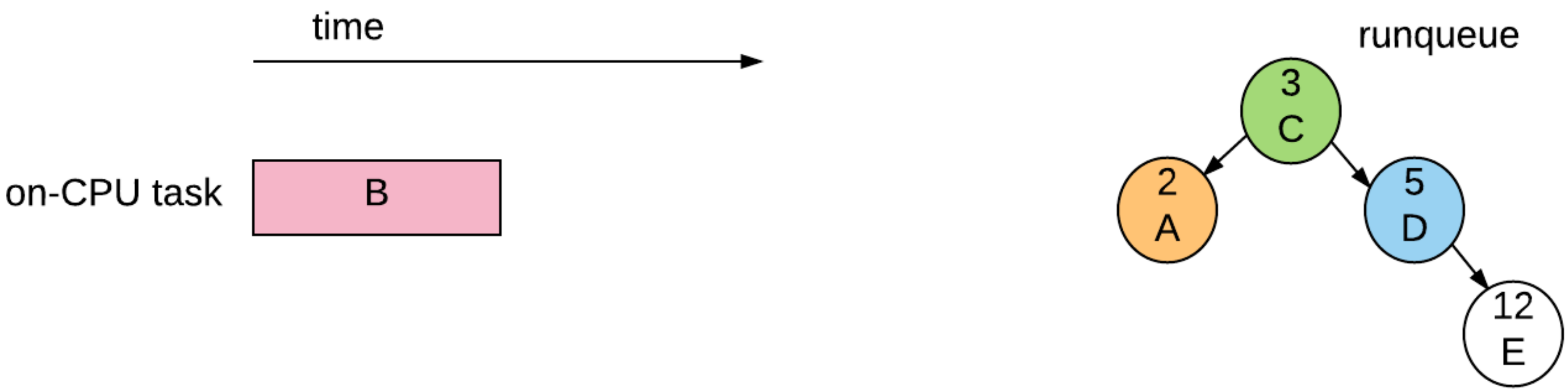
- For each core, there's a *runqueue* of runnable tasks.

- This is actually a red-black tree, ordered by task *vruntime* (basically real runtime divided by task weight).

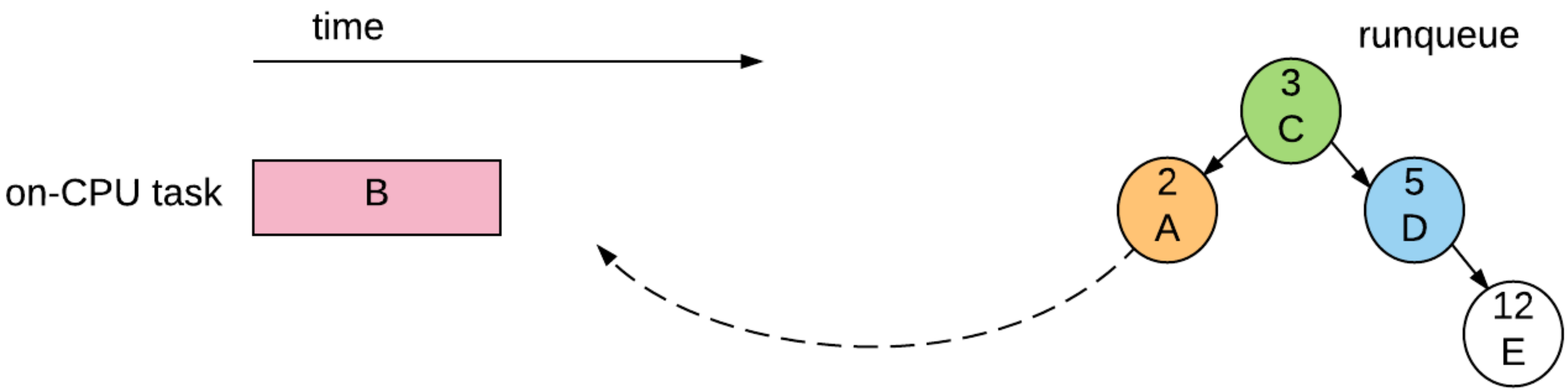
- As tasks run, they accumulate vruntime.

(Note: We're talking about the 'fair' scheduler here. There are other, non-default scheduling policies too.)

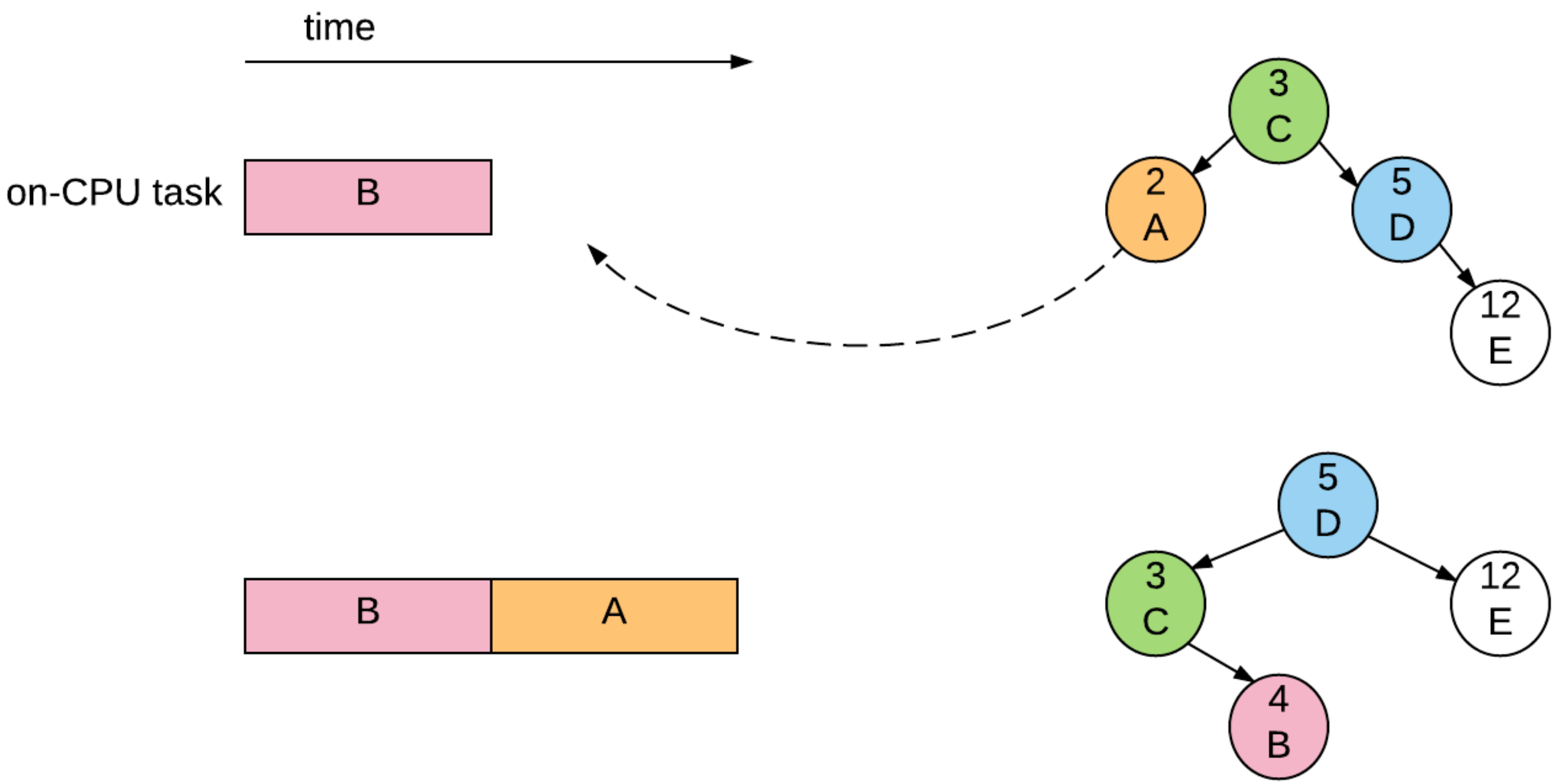
At task switch time, the scheduler pulls the leftmost task off the runqueue and runs it next.



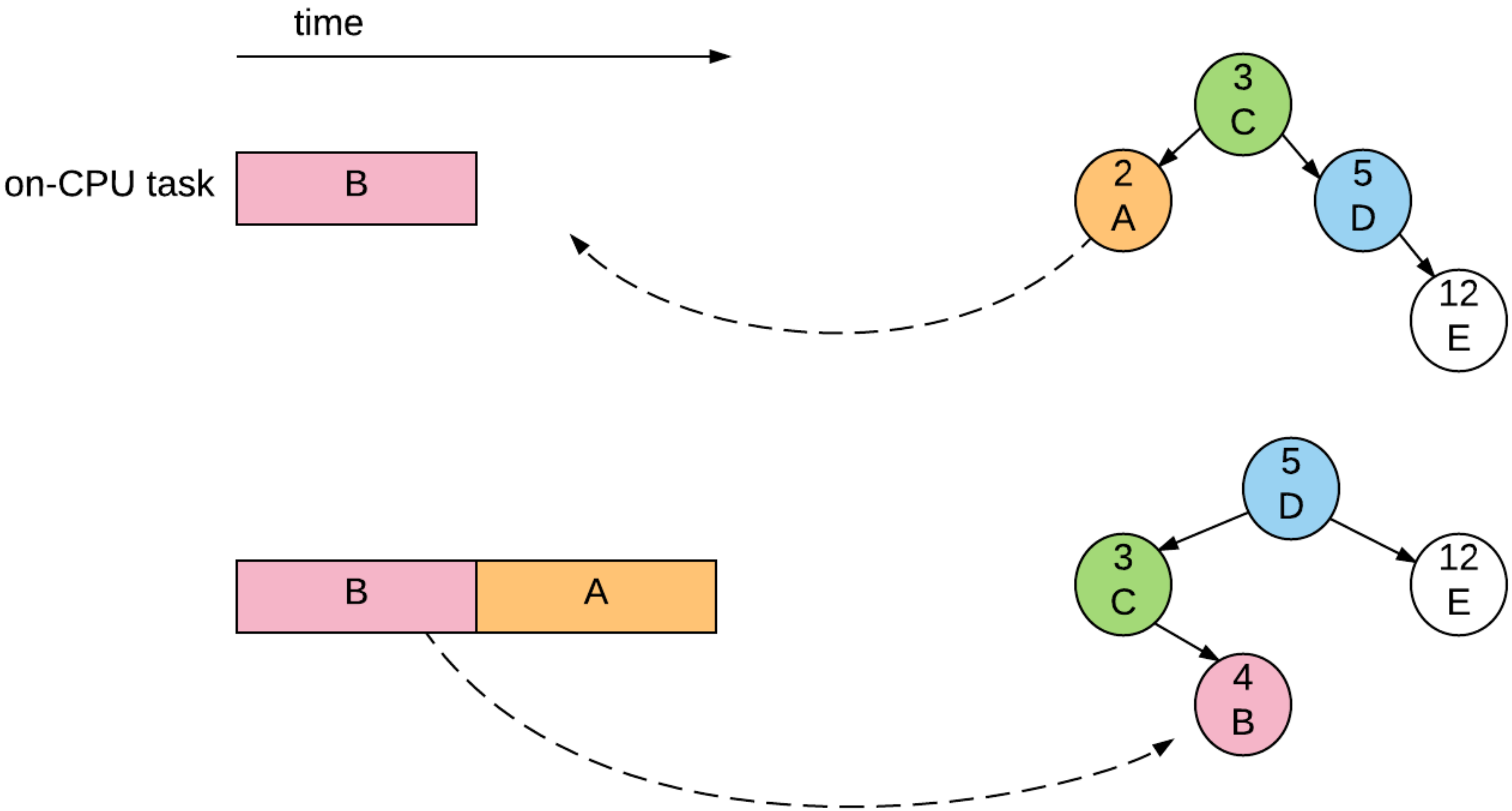
At task switch time, the scheduler pulls the leftmost task off the runqueue and runs it next.



At task switch time, the scheduler pulls the leftmost task off the runqueue and runs it next.



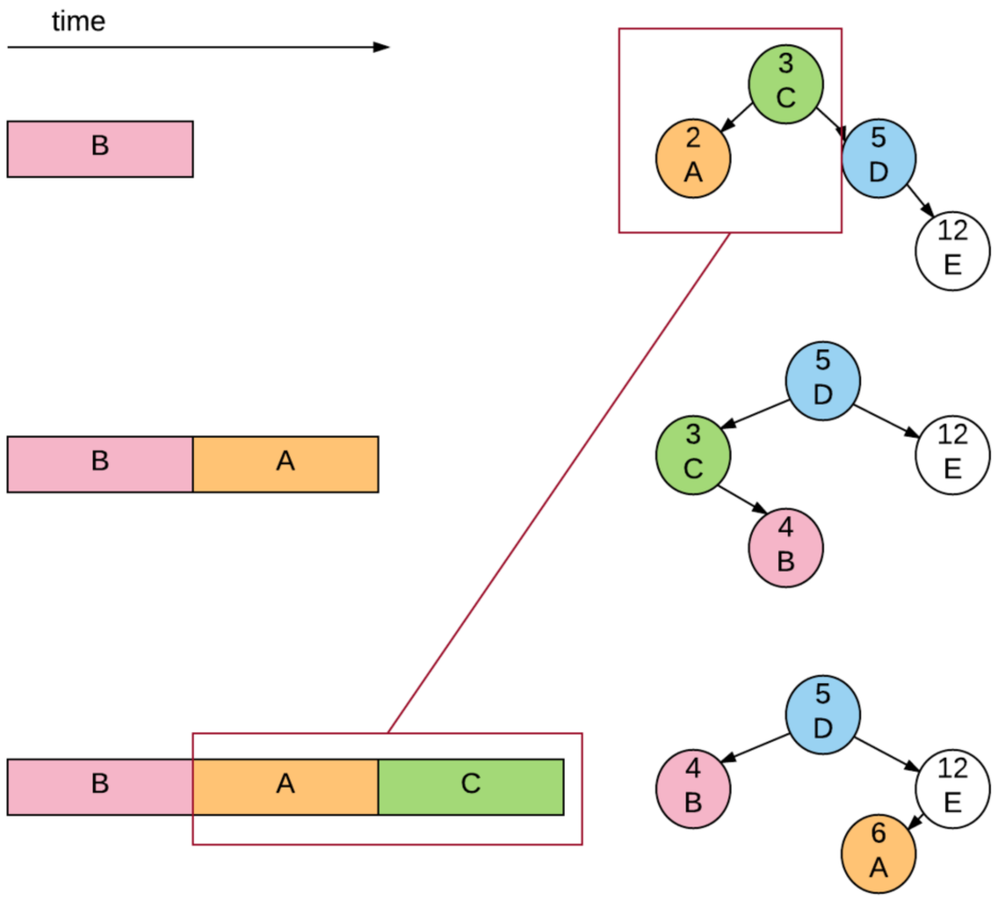
Preempted (and new or woken) tasks go on the runqueue.



So the runqueue is a timeline of future task execution.

Tasks are guaranteed a "fair" allocation of runtime.

Scheduling is O(log n) in the number of tasks.



What prompts a task switch?

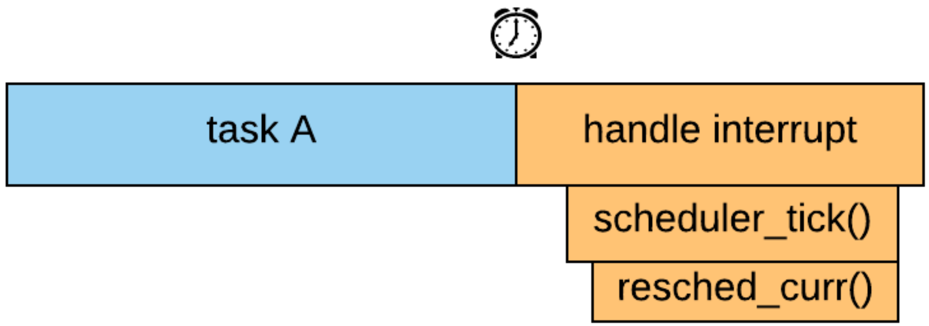
1. The running task blocks, and explicitly calls into the scheduler:

```
// fs/pipe.c
void pipe_wait(struct pipe_inode_info *pipe)
{
        // ...
        prepare_to_wait(&pipe->wait, &wait, TASK_INTERRUPTIBLE);
        pipe_unlock(pipe);
        schedule();
        finish_wait(&pipe->wait, &wait);
        pipe_lock(pipe);
}
```

2. The running task is forcibly preempted.

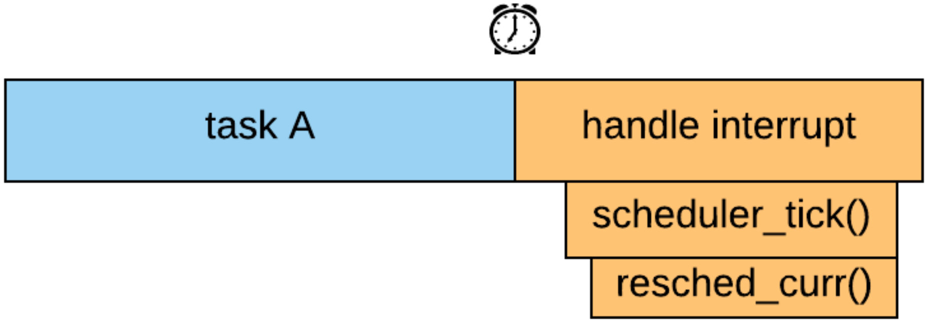
Preemption

A hardware timer drives preemption of CPU-hogging tasks.



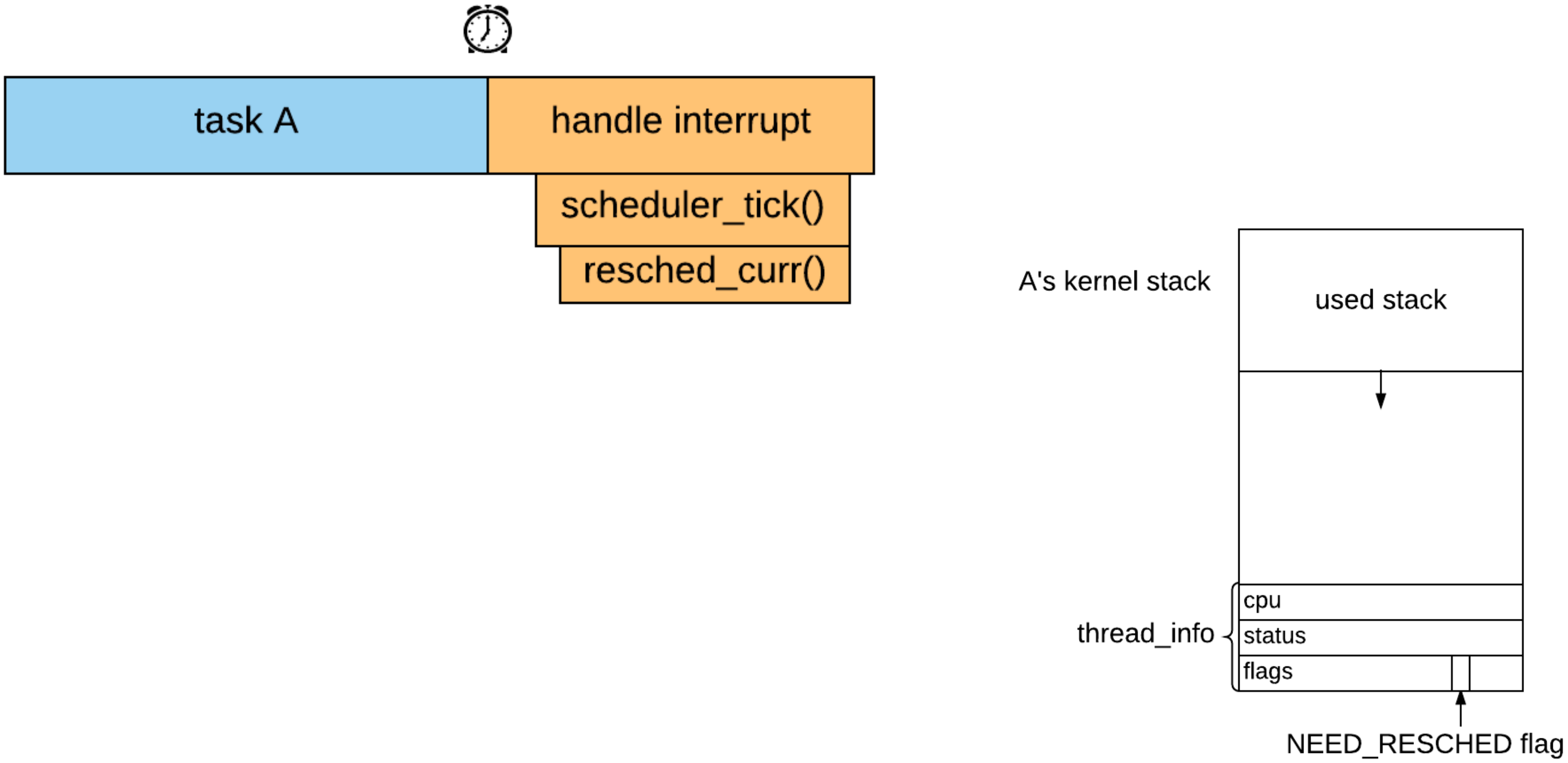
A hardware timer drives preemption of CPU-hogging tasks.

Preempting directly from the interrupt handler could cause funny stuff in a nested control path.



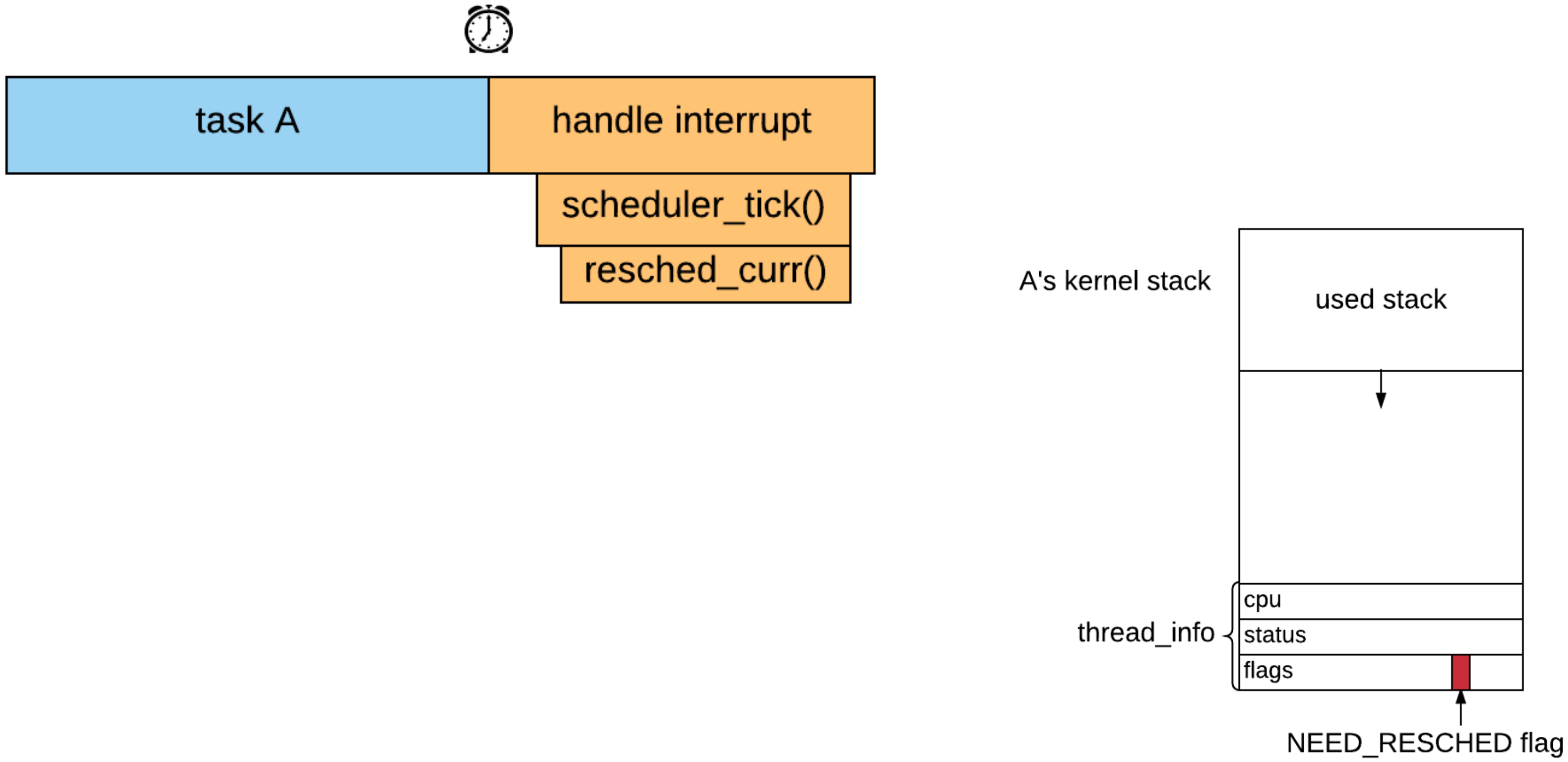
A hardware timer drives preemption of CPU-hogging tasks.

If the task is due for preemption, the interrupt handler sets a flag in the task's *thread_info* struct, signalling that rescheduling should happen.



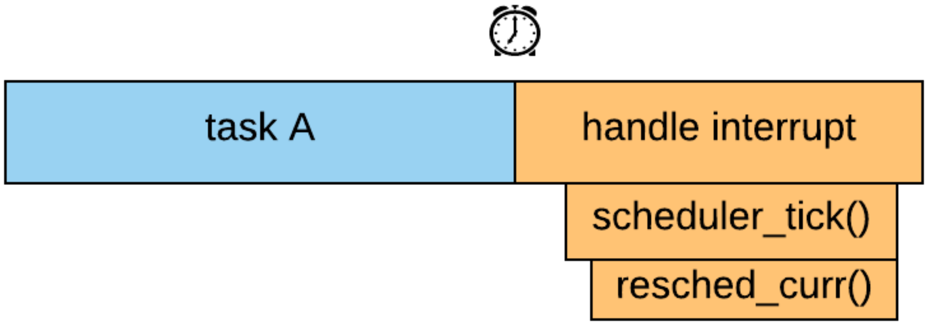
A hardware timer drives preemption of CPU-hogging tasks.

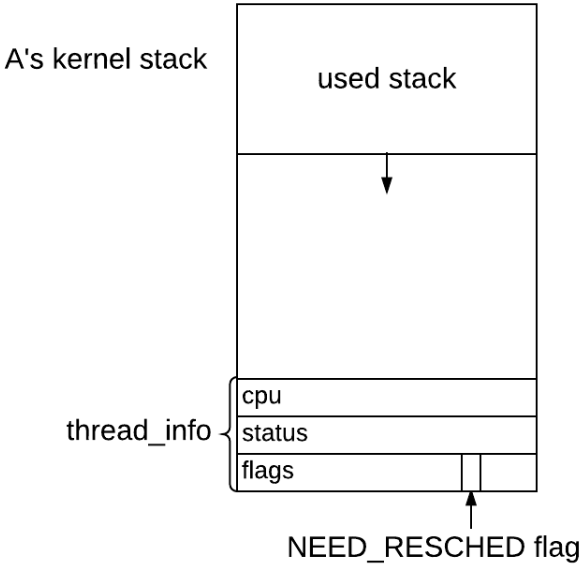
If the task is due for preemption, the interrupt handler sets a flag in the task's *thread_info* struct, signalling that rescheduling should happen.



A hardware timer drives preemption of CPU-hogging tasks.

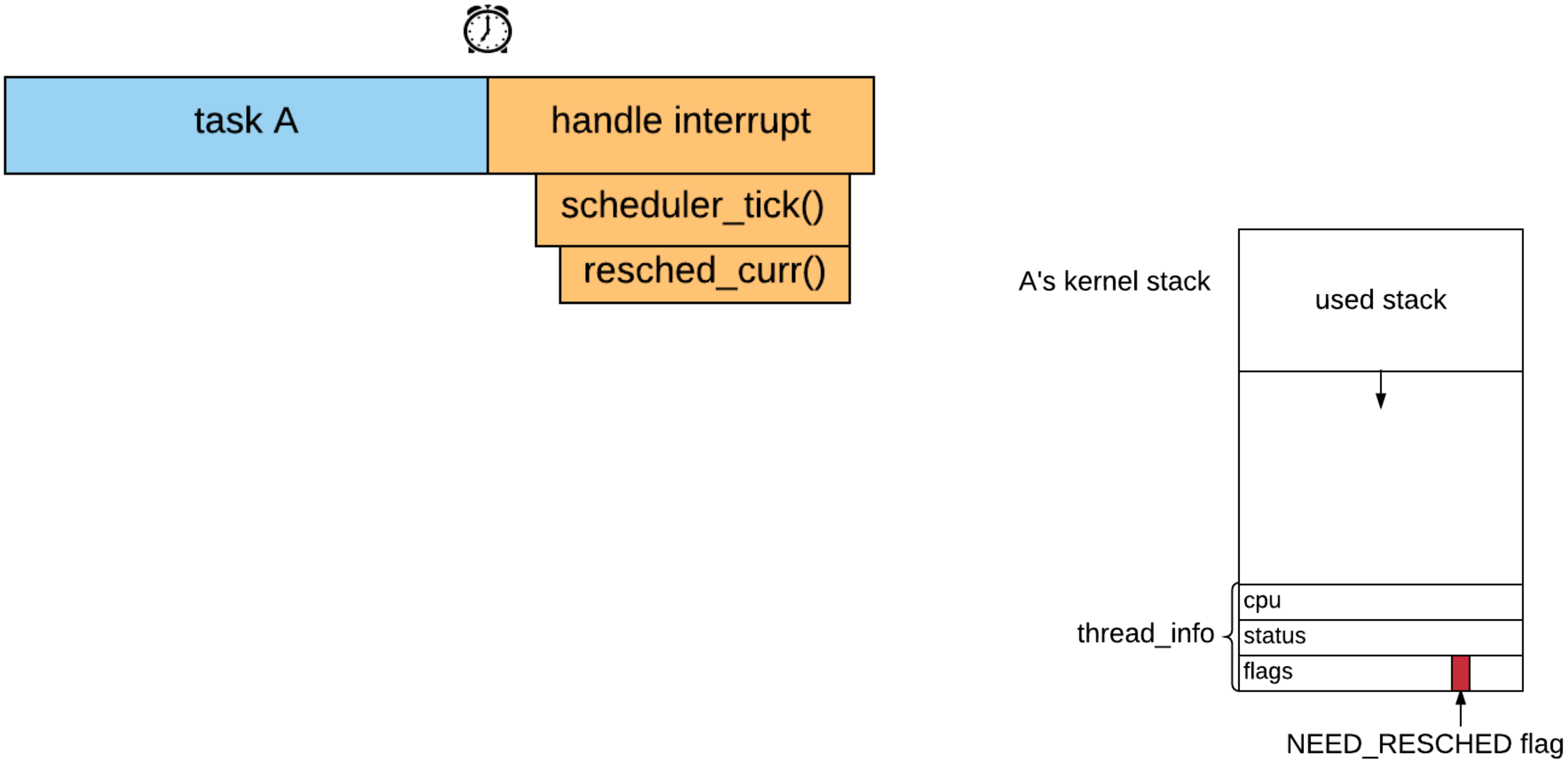
If the task is due for preemption, the interrupt handler sets a flag in the task's *thread_info* struct, signalling that rescheduling should happen.





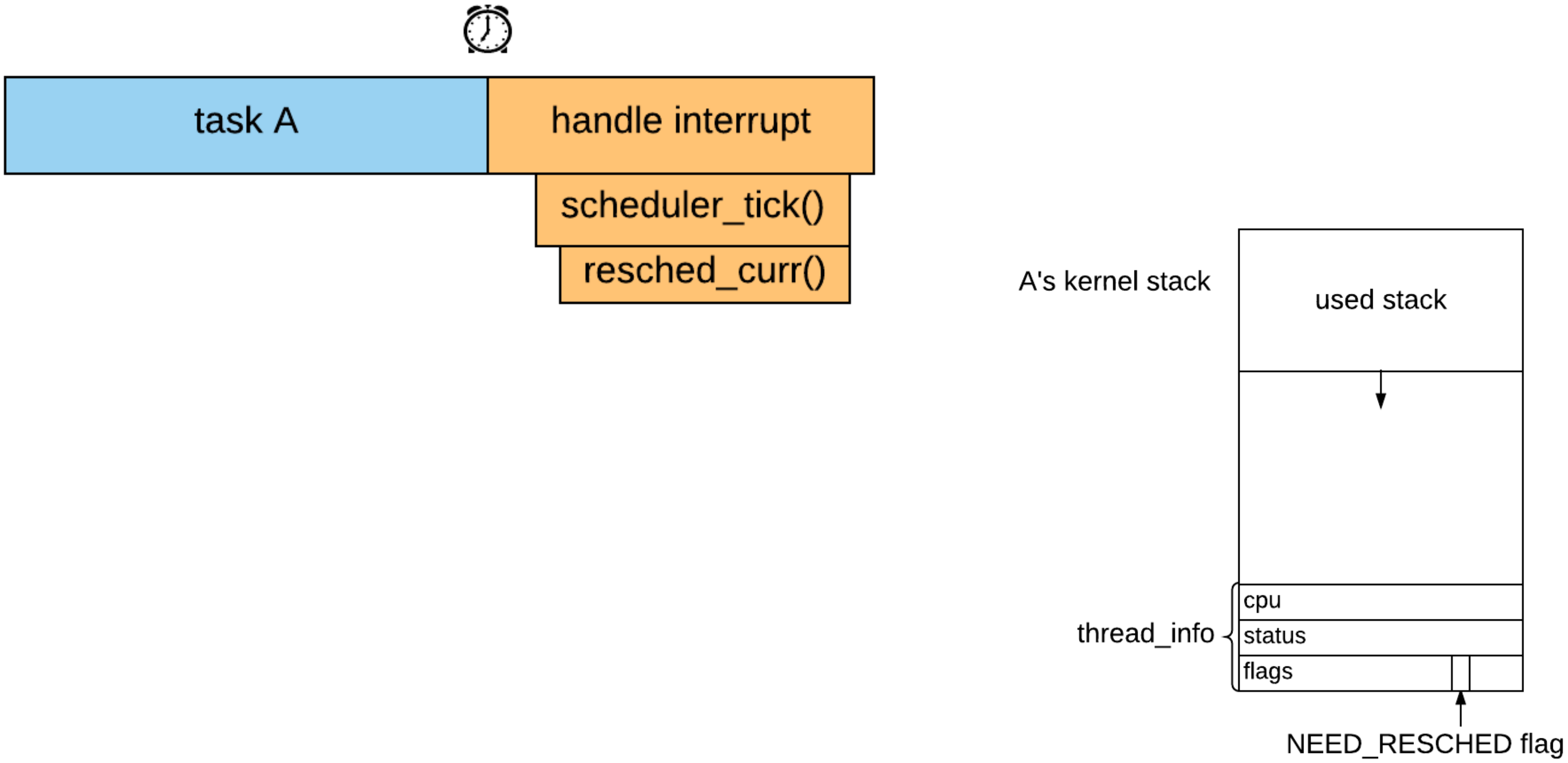
A hardware timer drives preemption of CPU-hogging tasks.

If the task is due for preemption, the interrupt handler sets a flag in the task's *thread_info* struct, signalling that rescheduling should happen.



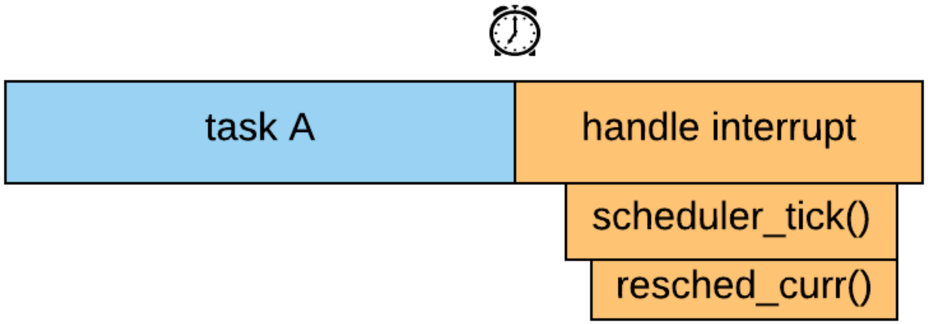
A hardware timer drives preemption of CPU-hogging tasks.

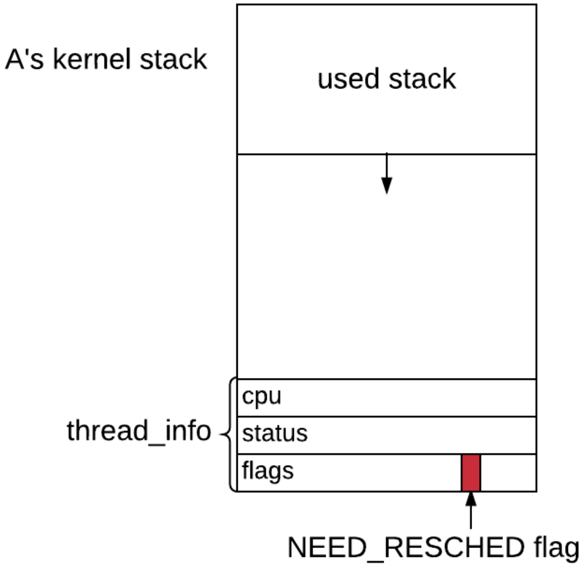
If the task is due for preemption, the interrupt handler sets a flag in the task's *thread_info* struct, signalling that rescheduling should happen.



A hardware timer drives preemption of CPU-hogging tasks.

If the task is due for preemption, the interrupt handler sets a flag in the task's *thread_info* struct, signalling that rescheduling should happen.

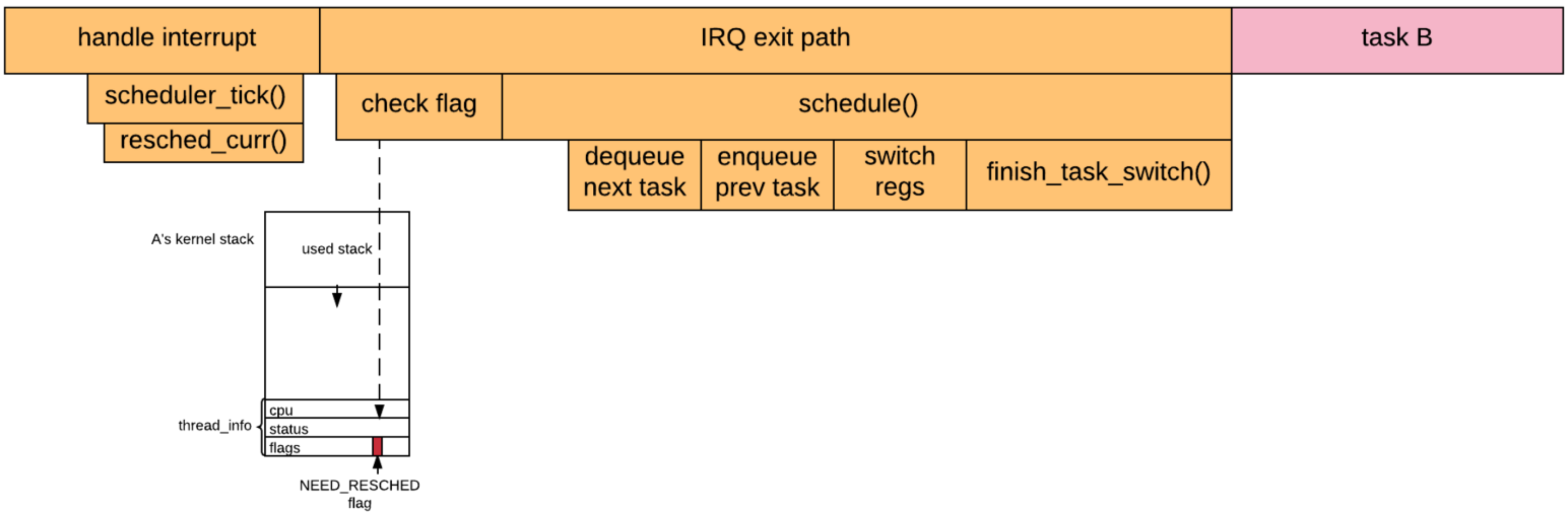




A hardware timer drives preemption of CPU-hogging tasks.

If the current task is due for preemption, the timer handler sets a flag in the task's *thread_info* struct.

Before returning to normal execution, we check that NEED_RESCHED flag, and call *schedule()* if we need to.

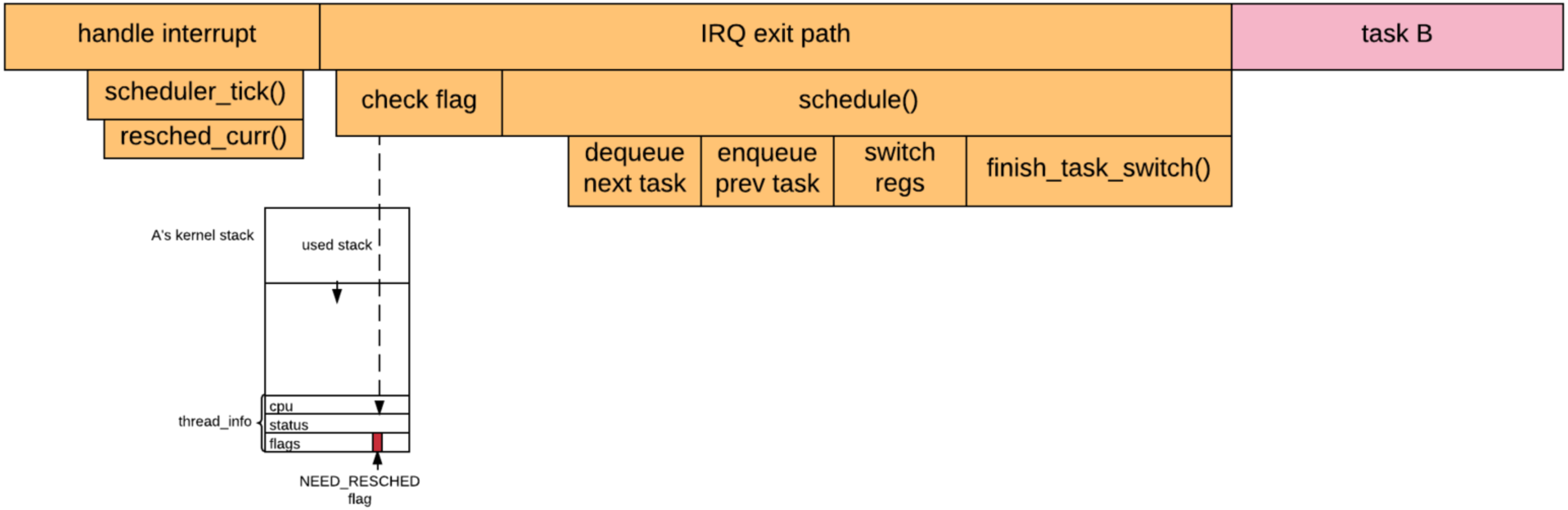


A hardware timer drives preemption of CPU-hogging tasks.

If the current task is due for preemption, the timer handler sets a flag in the task's *thread_info* struct.

Before returning to normal execution, we check that NEED_RESCHED flag, and call *schedule()* if we need to.

The schedule function dequeues the next task, enqueues the preempted one, swaps their processor state, and does some cleanup before actually running the next task.

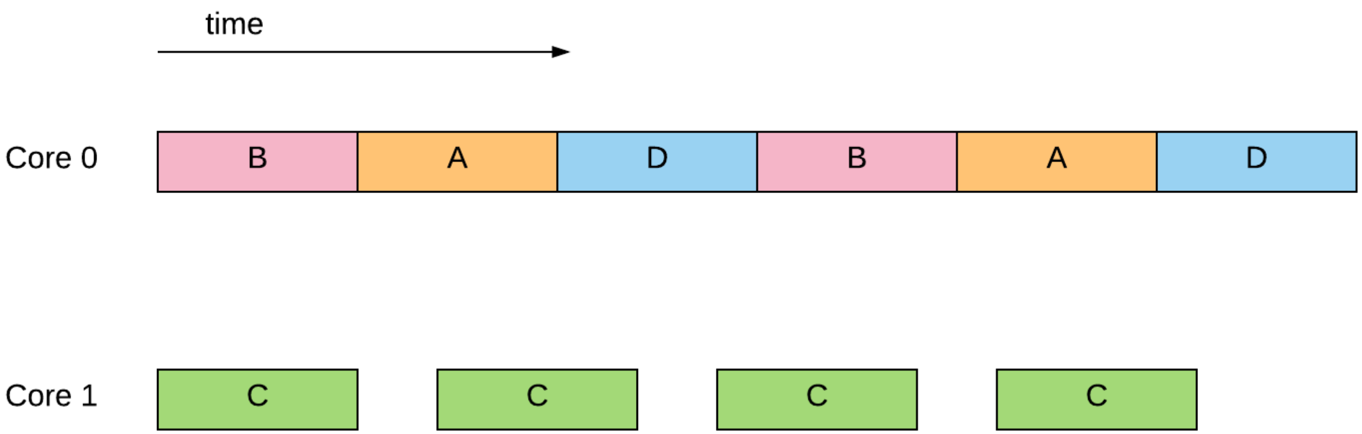


So far we have:

- Preemption
- Prioritization
- Fairness

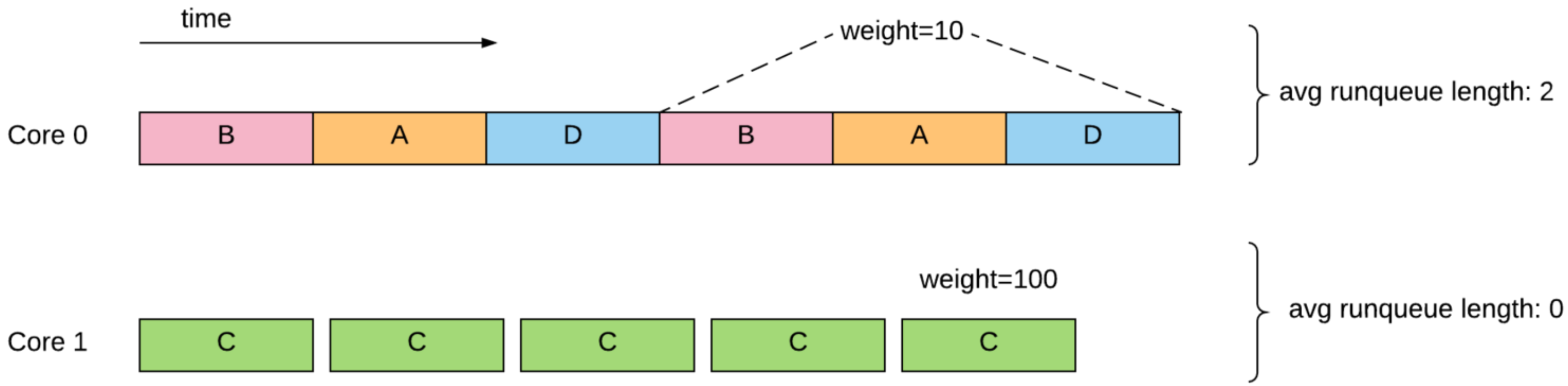
Multicore

Per-process runqueues limit contention and cache thrashing but can lead to unbalanced task distribution.



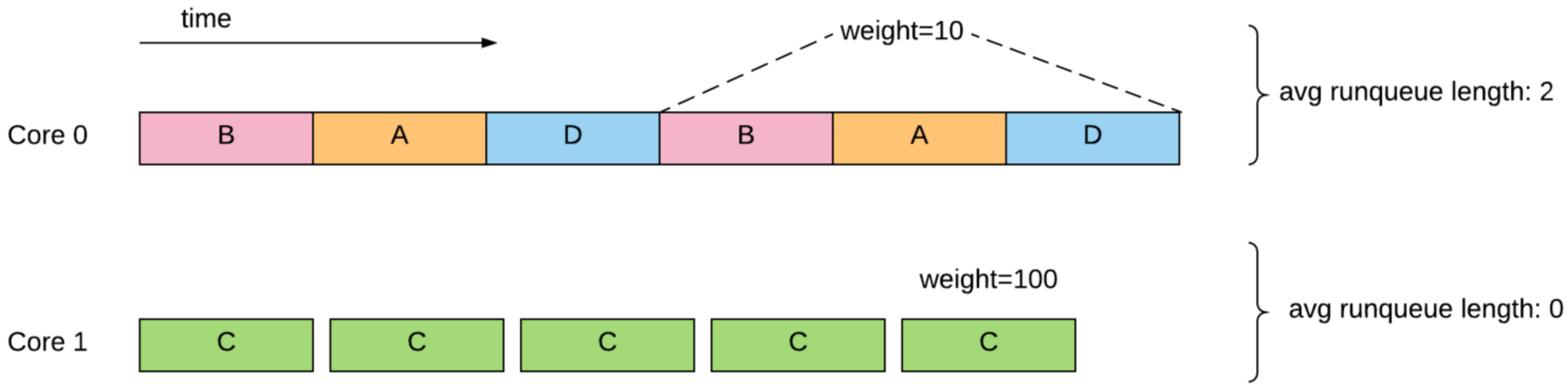
So each core periodically runs a load-balancing procedure.

But fair balancing is tricky.
Say task *C* has higher weight (priority) than tasks *A, B, D*:



Balancing runqueues based on length alone could deprive C of runtime.

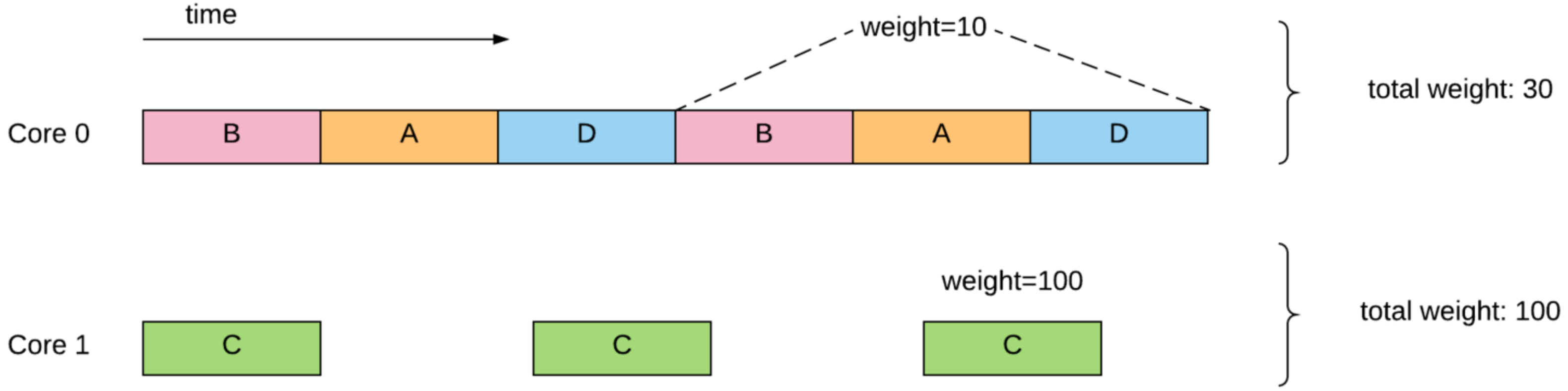
But fair balancing is tricky.
Say task *C* has higher weight (priority) than tasks *A, B, D*:



Balancing runqueues based on length alone could deprive C of runtime.

We could try balancing based on total task weight.

But fair balancing is tricky.
Say task *C* has higher weight (priority) than tasks *A, B, D*:



Balancing runqueues based on length alone could deprive C of runtime.

We could try balancing based on total task weight.

But if task C frequently sleeps, this is inefficient.

So balancing uses a "load" metric based on task weight and task CPU utilization.

At this point, you could be thinking. . .

This is kind of complicated! How can I figure out all these details?

1. Listen to some bozo's talk
2. Stare really hard at the source code
3. Use *ftrace*, 'the function tracer'

 - Dynamically traces all* function entry/return points in the kernel!

*almost (not architecture-specific functions defined in assembly) ftrace is kind of wonky to use:

```
$ mount -t debugfs none /sys/kernel/debug/
$ echo function_graph > /sys/kernel/debug/current_tracer
$ cat /sys/kernel/debug/trace

# tracer:
CPU   TASK/PID          DURATION                  FUNCTION CALLS
|     |    |             |   |                     |   |   |   |
2)     <idle>-0    |                |          local_apic_timer_interrupt
2)     <idle>-0    |                |            hrtimer_interrupt() {
2)     <idle>-0    |    0.042 us    |              _raw_spin_lock();
2)     <idle>-0    |    0.101 us    |              ktime_get_update_offse
2)     <idle>-0    |                |              __run_hrtimer() {
```

But very powerful!

"What's the code path through the scheduler look like?"

```
 DURATION                  FUNCTION CALLS
  |   |                    |   |   |   |
                 |         schedule() {
                 |           __schedule() {
  0.043 us       |             rcu_note_context_switch();
  0.044 us       |             _raw_spin_lock_irq();
                 |             deactivate_task() {
                 |               dequeue_task() {
  0.045 us       |                 update_rq_clock.part.84();
                 |                 dequeue_task_fair() {
                 |                   update_curr() {
  0.027 us       |                     update_min_vruntime();
  0.133 us       |                     cpuacct_charge();
  0.912 us       |                   }
  0.037 us       |                   update_cfs_rq_blocked_load();
  0.040 us       |                   clear_buddies();
  0.044 us       |                   account_entity_dequeue();
  0.043 us       |                   update_min_vruntime();
  0.038 us       |                   update_cfs_shares();
  0.039 us       |                   hrtick_update();
  4.197 us       |                 }
  4.906 us       |               }
  5.284 us       |             }
                 |             pick_next_task_fair() {
                 |               pick_next_entity() {
  0.026 us       |                 clear_buddies();
  0.564 us       |               }
  0.041 us       |               put_prev_entity();
  0.120 us       |               set_next_entity();
  1.861 us       |             }
  0.075 us       |             finish_task_switch();
```

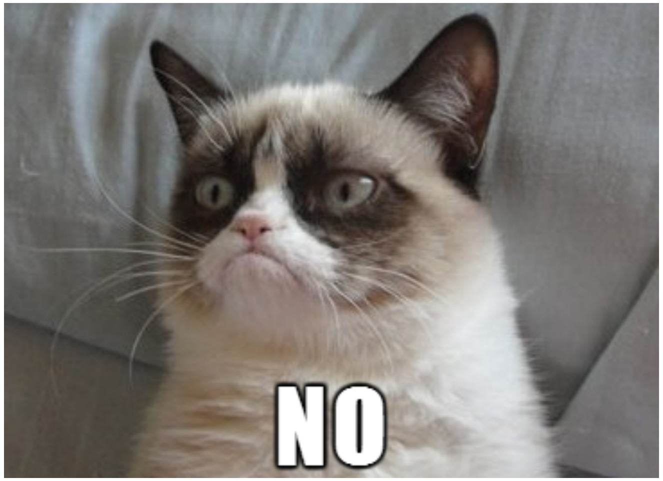
"What happens when you call *read()* on a pipe?"

```
                |   SyS_read() {
                |     __fdget_pos() {
0.059 us        |       __fget_light();
0.529 us        |     }
                |     vfs_read() {
                |       rw_verify_area() {
                |         security_file_permission() {
0.039 us        |           cap_file_permission();
0.058 us        |           __fsnotify_parent();
0.059 us        |           fsnotify();
1.462 us        |         }
1.960 us        |       }
                |       new_sync_read() {
0.050 us        |         iov_iter_init();
                |         pipe_read() {
                |           mutex_lock() {
0.045 us        |             _cond_resched();
0.581 us        |           }
                |           pipe_wait() {
                |             prepare_to_wait() {
0.052 us        |               _raw_spin_lock_irqsave();
0.054 us        |               _raw_spin_unlock_irqrestore();
1.181 us        |             }
0.053 us        |             mutex_unlock();
                |             schedule() {
```

The Linux CFS scheduler

- performant
- scalable
- robust
- traceable

End of story?


NO

Scheduling in User Space

Rationale

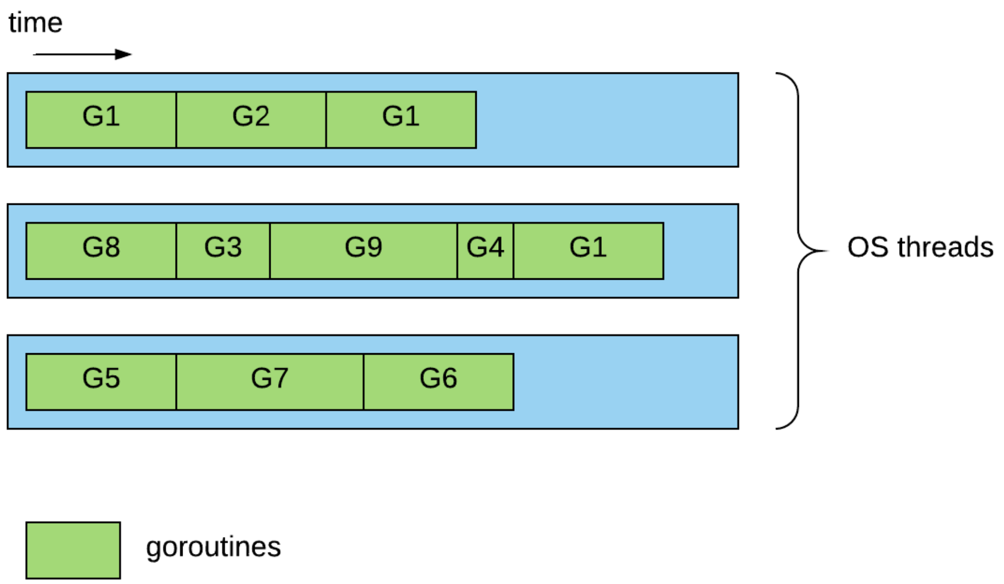
Target different performance characteristics

Decouple concurrency from memory usage

Support managed-memory runtimes

Userspace scheduling: Go

In Go, code runs in *goroutines*, lightweight threads managed by the runtime. Goroutines are multiplexed onto OS threads (this is *M-N* scheduling).



The claim:

"Because OS threads are scheduled by the kernel, passing control from one thread to another requires a full context switch [...]. This operation is slow, due its poor locality and the number of memory accesses required.

[...]

Because it doesn't need a switch to kernel context, **rescheduling a goroutine is much cheaper than rescheduling a thread**."

Donovan & Kernighan, *The Go Programming Language*

If we rerun our ping-pong experiment with goroutines and channels...

```go
func worker(channels [2](chan int), idx int, loops int, wg *sync.WaitGroup) {
	for i := 0; i < loops; i++ {
		channels[idx] <- 1
		<-channels[1-idx]
	}
	wg.Done()
}

func main() {
	var channels = [2]chan int{make(chan int, 1), make(chan int, 1)}
	nloops := 10000000
	start := time.Now()
	var wg sync.WaitGroup
	wg.Add(2)
	go worker(channels, 0, nloops, &wg)
	go worker(channels, 1, nloops, &wg)
	wg.Wait()
	elapsed := time.Since(start).Seconds()
	fmt.Printf("%fs elapsed\n", elapsed)
	fmt.Printf("%f µs per switch\n", 1e6*elapsed/float64(2*nloops))
}
```

If we rerun our ping-pong experiment with goroutines and channels. . .

```
$ ./pingpong
Elapsed: 4.184381
0.209219 μs per switch
```

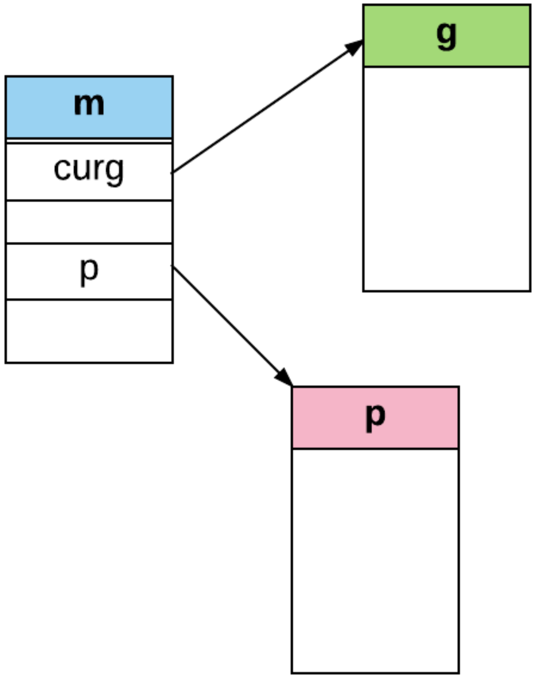
. . . it does look a good bit faster than thread switching.

So what's the Go scheduler doing?

The Go scheduler in a nutshell

Go runtime state is basically described by three data structures:

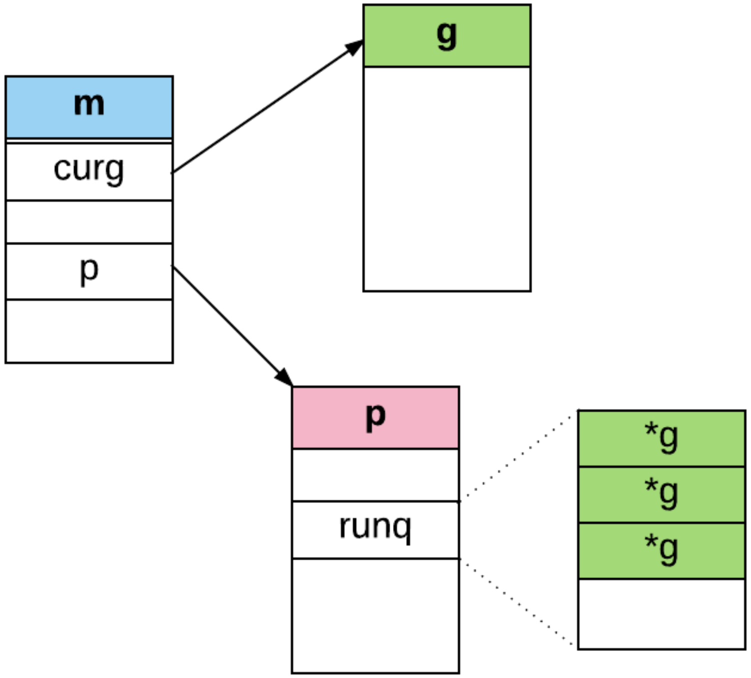
- An *M* represents an OS thread
- A *G* represents a goroutine
- A *P* represents general context for executing Go code.



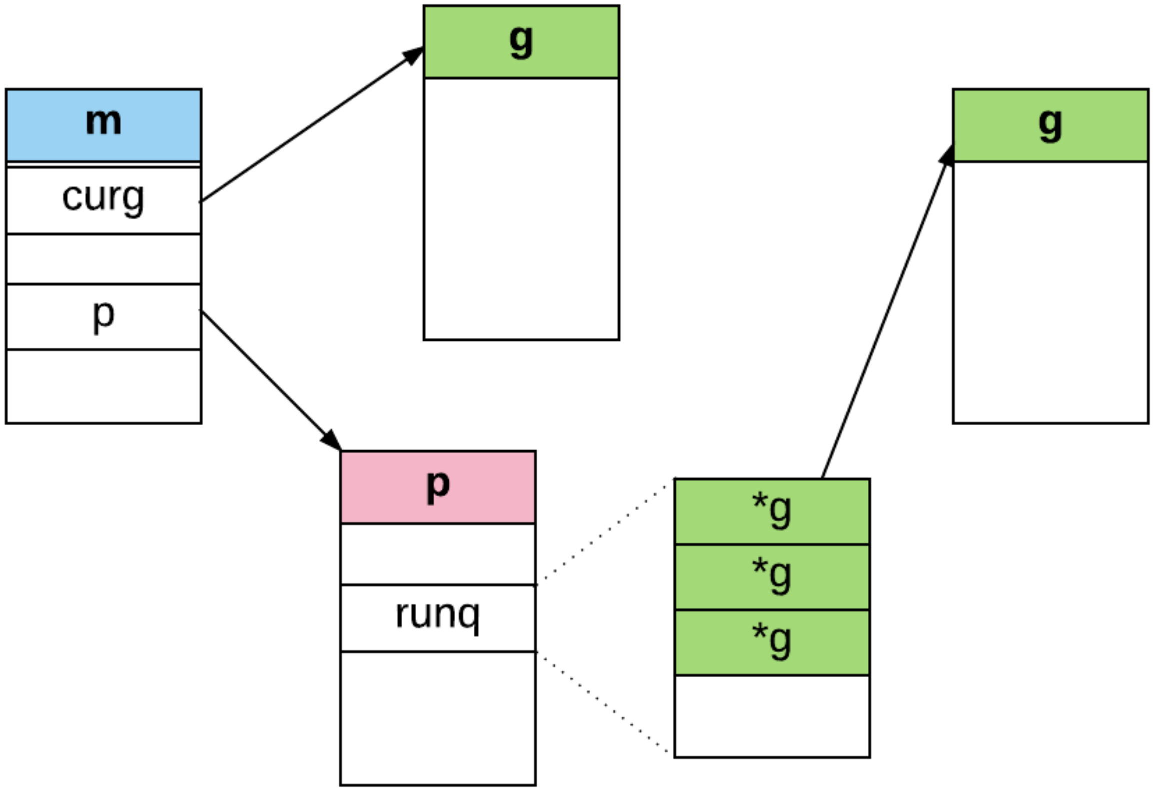
Go runtime state is basically described by three data structures:

- An *M* represents an OS thread
- A *G* represents a goroutine
- A *P* represents general context for executing Go code.

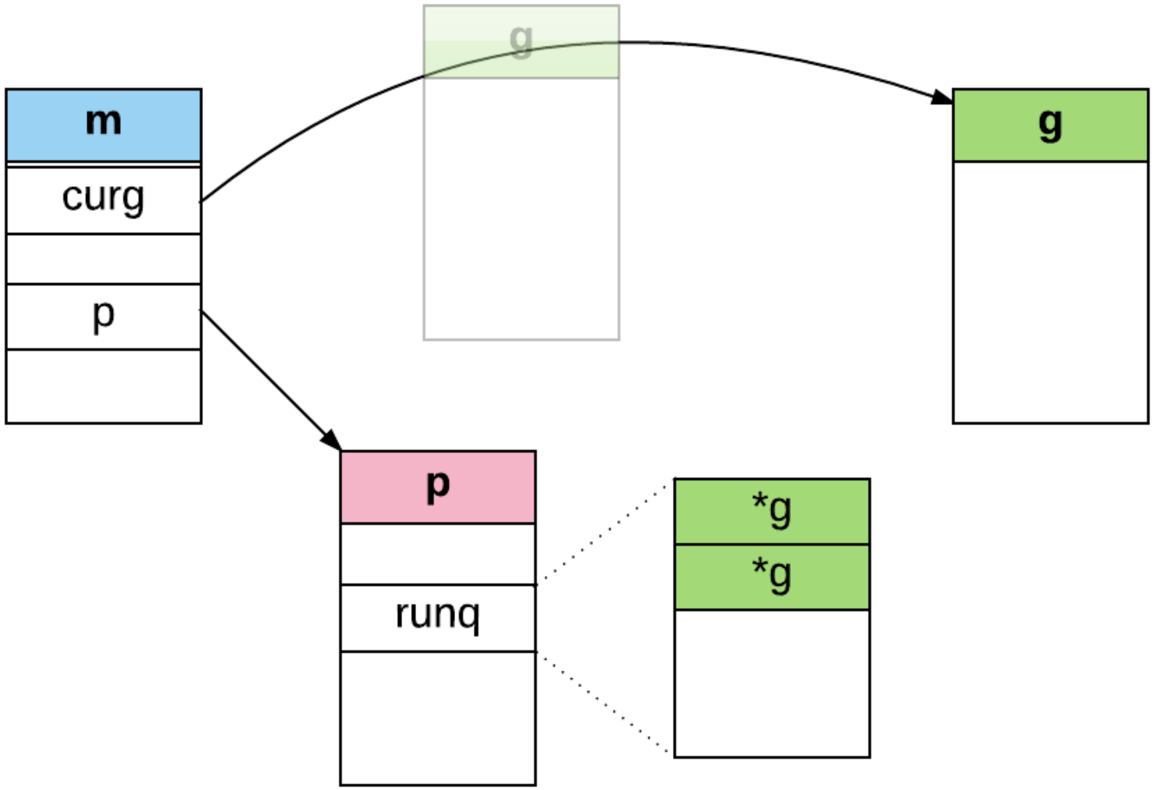
Each *P* contains a queue of runnable goroutines.



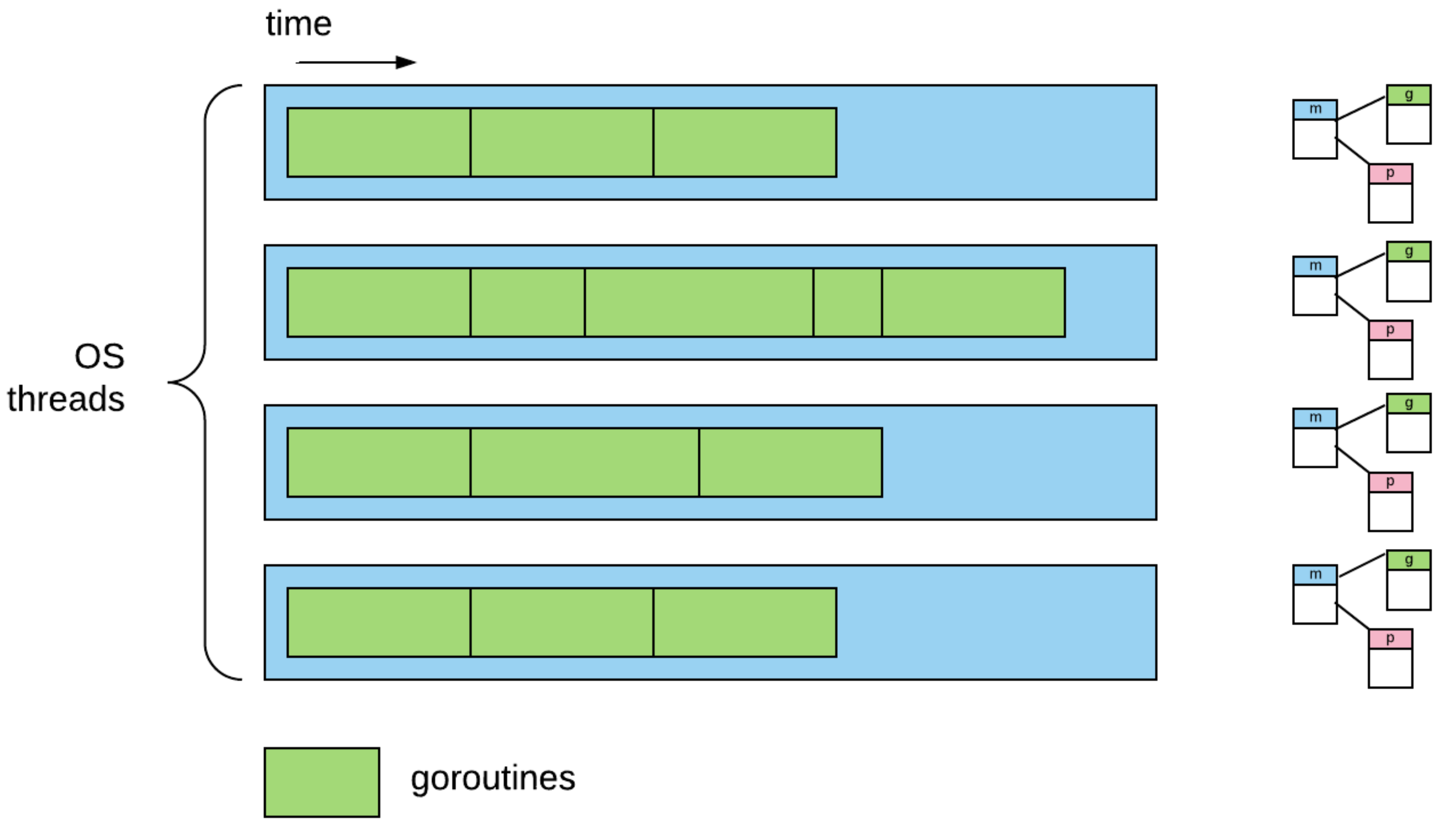
At context switch time, the next goroutine is pulled off the runqueue and run.



At context switch time, the next goroutine is pulled off the runqueue and run.

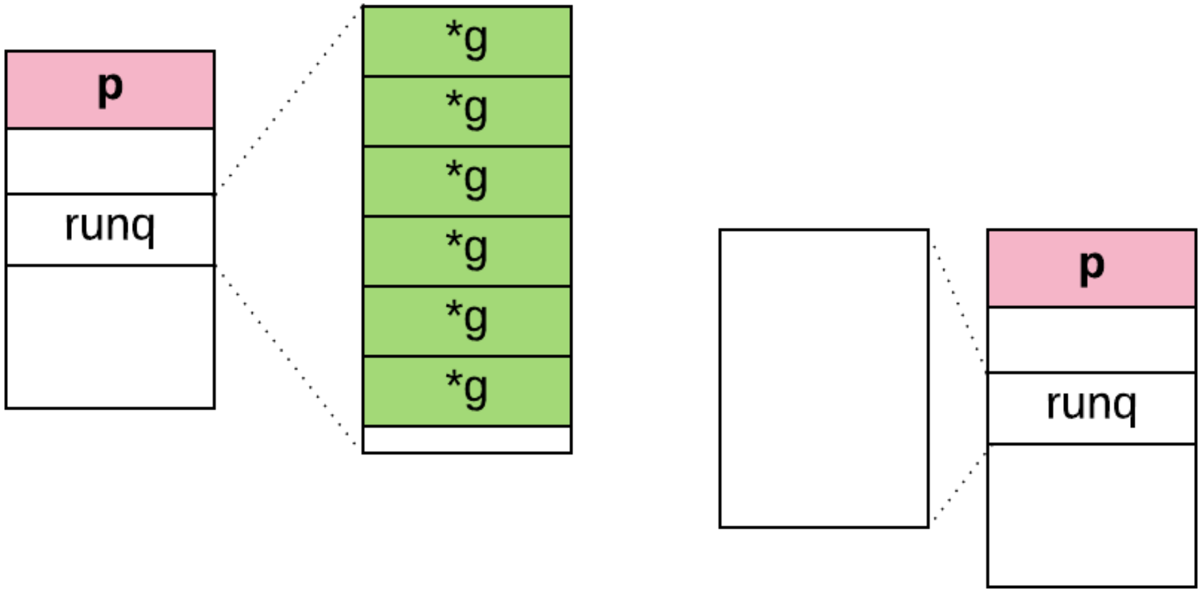


There's one *P* per core (by default). So on an *N*-core machine, up to *N* threads can concurrently execute Go code.



There's no regular inter-*P* runqueue load-balancing. Instead,

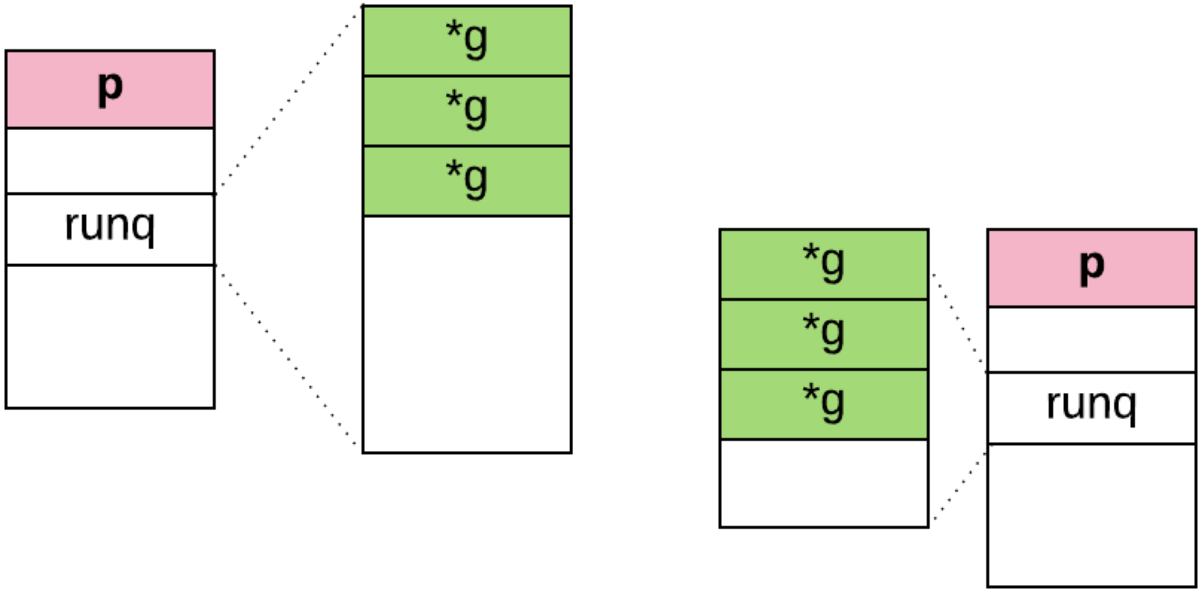
- Goroutines which were preempted or blocked in syscalls[1] go onto a special *global* runqueue.
- A *P* which becomes idle can steal work from another *P*.



[1]This is only true in some cases, but it's not important.

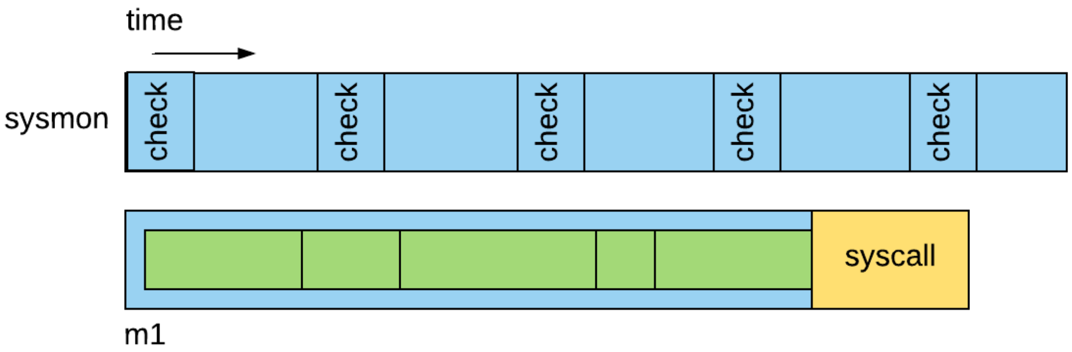
There's no regular inter-*P* runqueue load-balancing. Instead,

- Goroutines which were preempted or blocked in syscalls[1] go onto a special *global* runqueue.
- A *P* which becomes idle can steal work from another *P*.

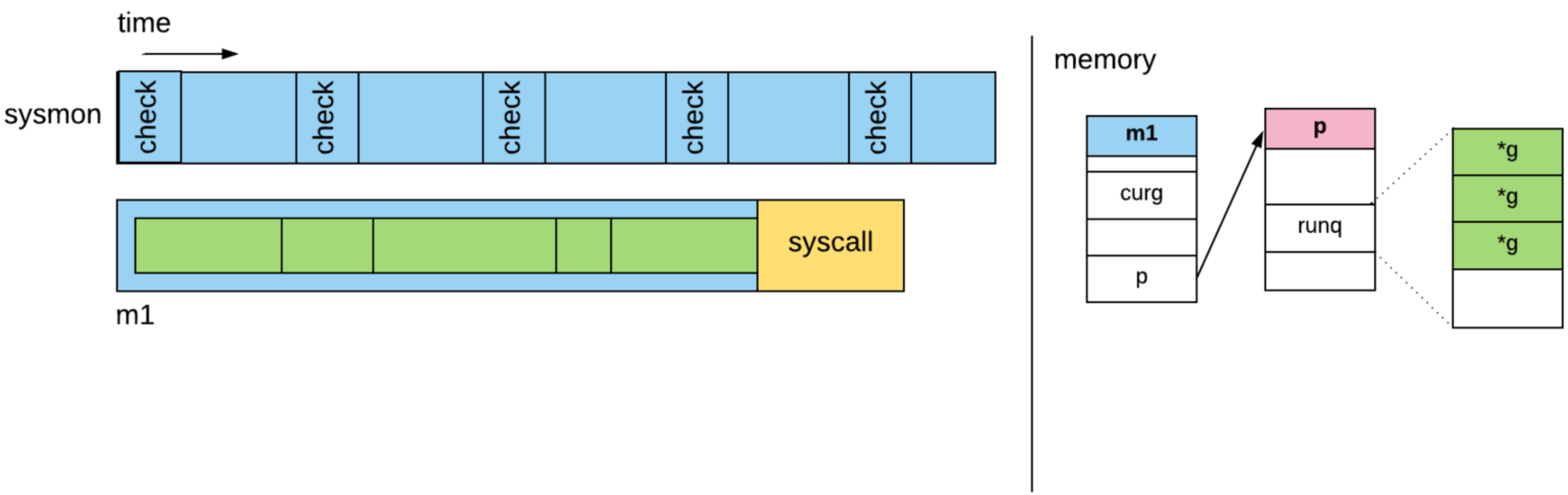


[1]This is only true in some cases, but it's not important.

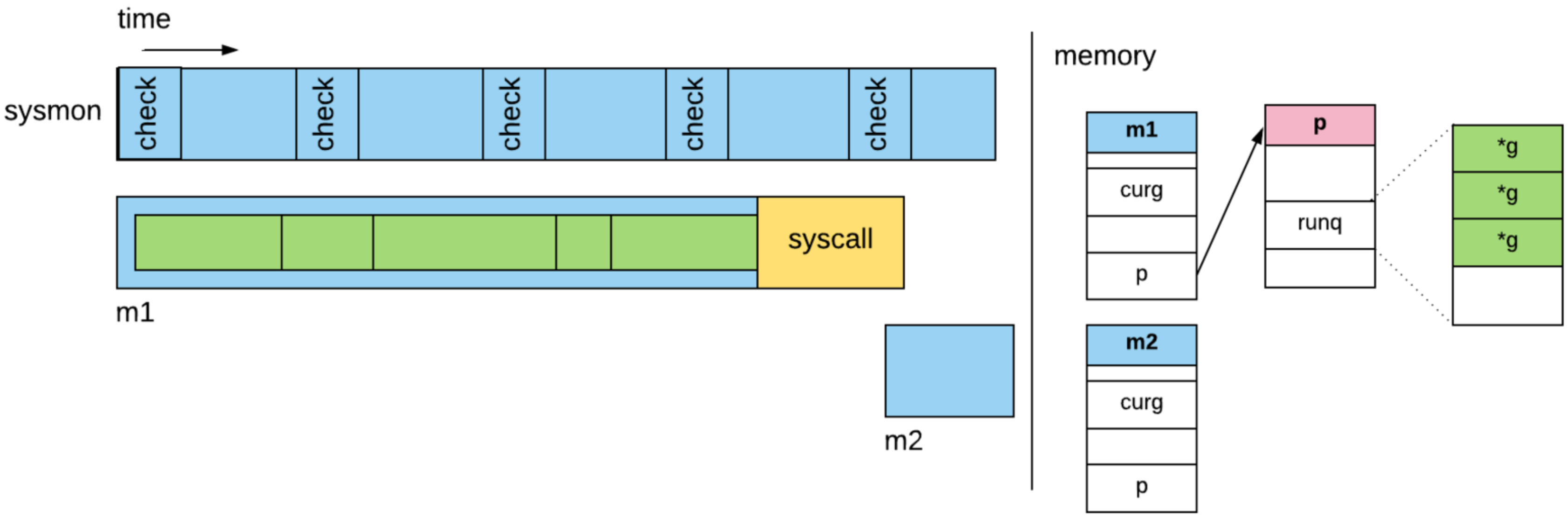
A separate *sysmon* thread implements *p* handoff if an *m* blocks in a syscall.



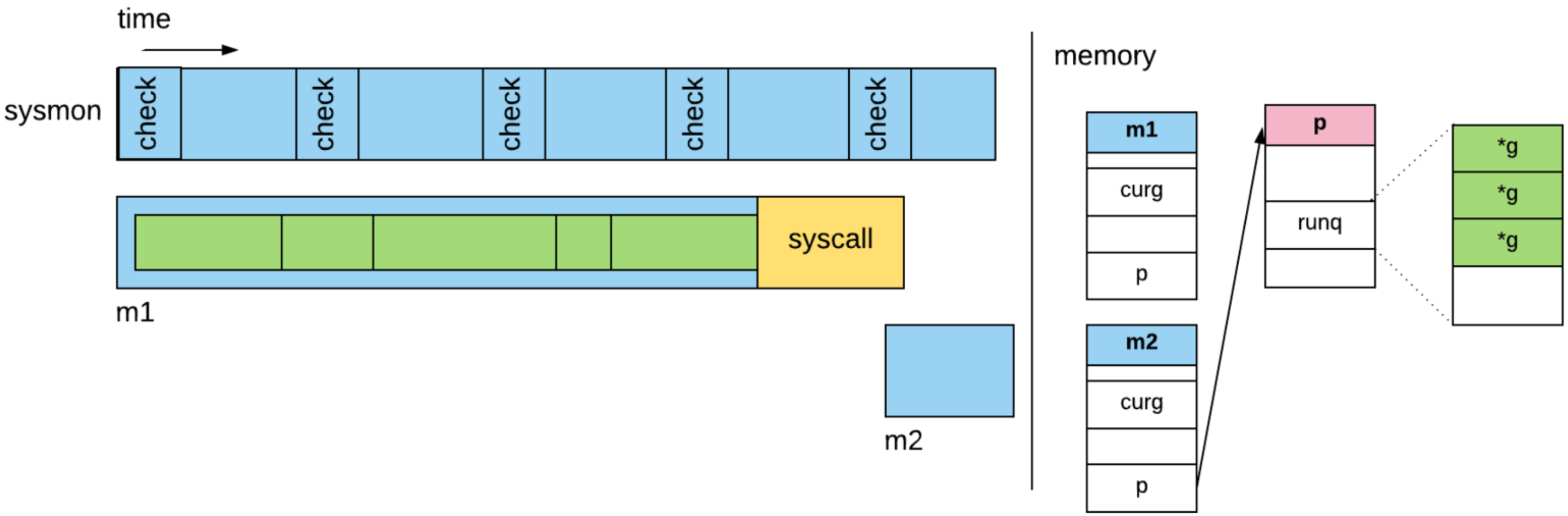
A separate *sysmon* thread implements *p* handoff if an *m* blocks in a syscall.



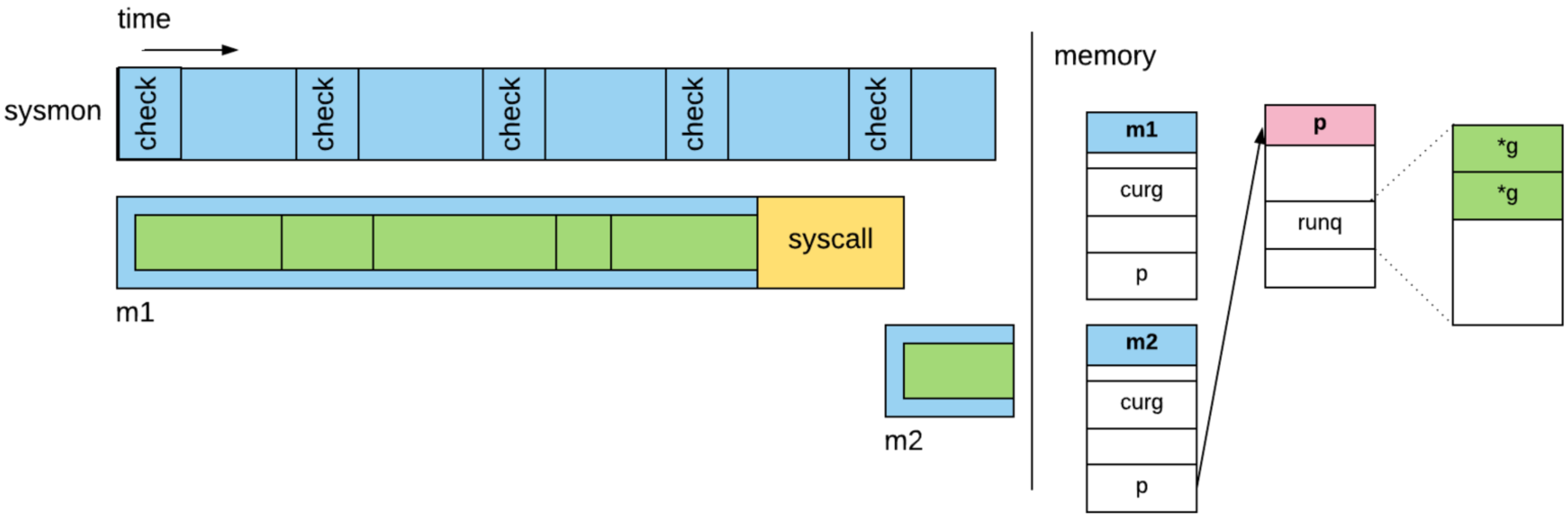
A separate *sysmon* thread implements *p* handoff if an *m* blocks in a syscall.



A separate *sysmon* thread implements *p* handoff if an *m* blocks in a syscall.



A separate *sysmon* thread implements *p* handoff if an *m* blocks in a syscall.



The sysmon thread also checks for long-running goroutines that should be preempted.

However, preemption can only happen at Go function entry, so tight loops can potentially block arbitrarily.

In Go, context switches are fast by virtue of simplicity.

This design supports lots of concurrent goroutines (millions),

but omits features (goroutine priorities, strong preemption).

Userspace scheduling: Erlang

Erlang's concurrency primitive is called a *process*.

Processes communicate via asynchronous message passing (no shared state).

Erlang code is compiled to bytecode and executed by a virtual machine.

This architecture enables a simple preemption mechanism
(not timer- or watcher-based).

It uses the notion of a *reduction budget*.

Reductions

Every Erlang process gets a reduction count (default 2000).

Every operation costs reductions:

- calling a function
- sending a message to another process
- I/O
- garbage collection
- etc.

After you use up your reduction budget, you get preempted.

The core of the VM is a bytecode dispatch loop.

For example, to call a function

```
// from the BEAM emulator source

emulator_loop:
  switch(Go) {       // 3700-line switch stateme
    // ...
    OpCase(i_call_f): {
      SET_CP(c_p, I+2);
      I = (BeamInstr *) Arg(0));
      Dispatch();
    }
    // ...
  }



#define Dispatch()
  do {
     dis_next = (BeamInstr *) *I;
     if (REDUCTIONS > 0 ||
           REDUCTIONS > -reduction_budget) {
        REDUCTIONS--;
        Go = dis_next;
        goto emulator_loop;
     } else {
        goto context_switch;
     }
  } while (0)
```

The core of the VM is a bytecode dispatch loop.

For example, to call a function,

(1) set the continuation pointer,

(2) advance the instruction pointer

(3) call *Dispatch()*

```
// from the BEAM emulator source

emulator_loop:
  switch(Go) {      // 3700-line switch statemen
    // ...
    OpCase(i_call_f): {
      SET_CP(c_p, I+2);
      I = (BeamInstr *) Arg(0));
      Dispatch();
    }
    // ...
  }



#define Dispatch()
  do {
     dis_next = (BeamInstr *) *I;
     if (REDUCTIONS > 0 ||
           REDUCTIONS > -reduction_budget) {
        REDUCTIONS--;
        Go = dis_next;
        goto emulator_loop;
     } else {
        goto context_switch;
     }
  } while (0)
```

The core of the VM is a bytecode dispatch loop.

For example, to call a function,

(1) set the continuation pointer,

(2) advance the instruction pointer

(3) call *Dispatch()*

```
// from the BEAM emulator source

emulator_loop:
  switch(Go) {       // 3700-line switch stateme
    // ...
    OpCase(i_call_f): {
      SET_CP(c_p, I+2);
      I = (BeamInstr *) Arg(0));
      Dispatch();
    }
    // ...
  }
```

If we still have reductions,

decrement the reduction counter

and go through the emulator loop

for the next instruction.

```
#define Dispatch()
  do {
     dis_next = (BeamInstr *) *I;
     if (REDUCTIONS > 0 ||
           REDUCTIONS > -reduction_budget) {
        REDUCTIONS--;
        Go = dis_next;
        goto emulator_loop;
     } else {
        goto context_switch;
     }
 } while (0)
```

The core of the VM is a bytecode dispatch loop.

For example, to call a function,
(1) set the continuation pointer,
(2) advance the instruction pointer
(3) call *Dispatch()*

```
// from the BEAM emulator source

emulator_loop:
  switch(Go) {      // 3700-line switch stateme
    // ...
    OpCase(i_call_f): {
      SET_CP(c_p, I+2);
      I = (BeamInstr *) Arg(0));
      Dispatch();
    }
    // ...
  }
```

If we still have reductions, decrement the reduction counter and go through the emulator loop for the next instruction.

Otherwise, context-switch.

```
#define Dispatch()
  do {
     dis_next = (BeamInstr *) *I;
     if (REDUCTIONS > 0 ||
            REDUCTIONS > -reduction_budget) {
       REDUCTIONS--;
       Go = dis_next;
       goto emulator_loop;
     } else {
       goto context_switch;
     }
 } while (0)
```

Why does this matter?

Why does this matter?
Let's try an experiment.

```
func main() {
  for i := 0; i < 4; i++ {
    go func() { for { time.Now() }}();
  }

  for i := 0; i < 1000; i++ {
    target_delay_ns := rand.Intn(1000 * 1000 * 1000)
    ts := time.Now()
    time.Sleep(time.Duration(target_delay_ns) * time.Nanosecond)
    actual_delay_ns = time.Since(ts).Nanoseconds()
    jitter = actual_delay_ns - target_delay_ns
    fmt.Printf("%d\n", target_delay_ns)
  }
}
```

Why does this matter?

Let's try an experiment.

busy tight loop (saturate cores)

A small Go program:

```go
func main() {
  for i := 0; i < 4; i++ {
    go func() { for { time.Now() }}();
  }

  for i := 0; i < 1000; i++ {
    target_delay_ns := rand.Intn(1000 * 1000 * 1000)
    ts := time.Now()
    time.Sleep(time.Duration(target_delay_ns) * time.Nanosecond)
    actual_delay_ns = time.Since(ts).Nanoseconds()
    jitter = actual_delay_ns - target_delay_ns
    fmt.Printf("%d\n", target_delay_ns)
  }
}
```

Why does this matter?

Let's try an experiment.

busy tight loop (saturate cores)

A small Go program:

```
func main() {
  for i := 0; i < 4; i++ {
    go func() { for { time.Now() }}();
  }

  for i := 0; i < 1000; i++ {
    target_delay_ns := rand.Intn(1000 * 1000 * 1000)
    ts := time.Now()
    time.Sleep(time.Duration(target_delay_ns) * time.Nanosecond)
    actual_delay_ns = time.Since(ts).Nanoseconds()
    jitter = actual_delay_ns - target_delay_ns
    fmt.Printf("%d\n", target_delay_ns)
  }
}
```

sleep

Why does this matter?

Let's try an experiment.

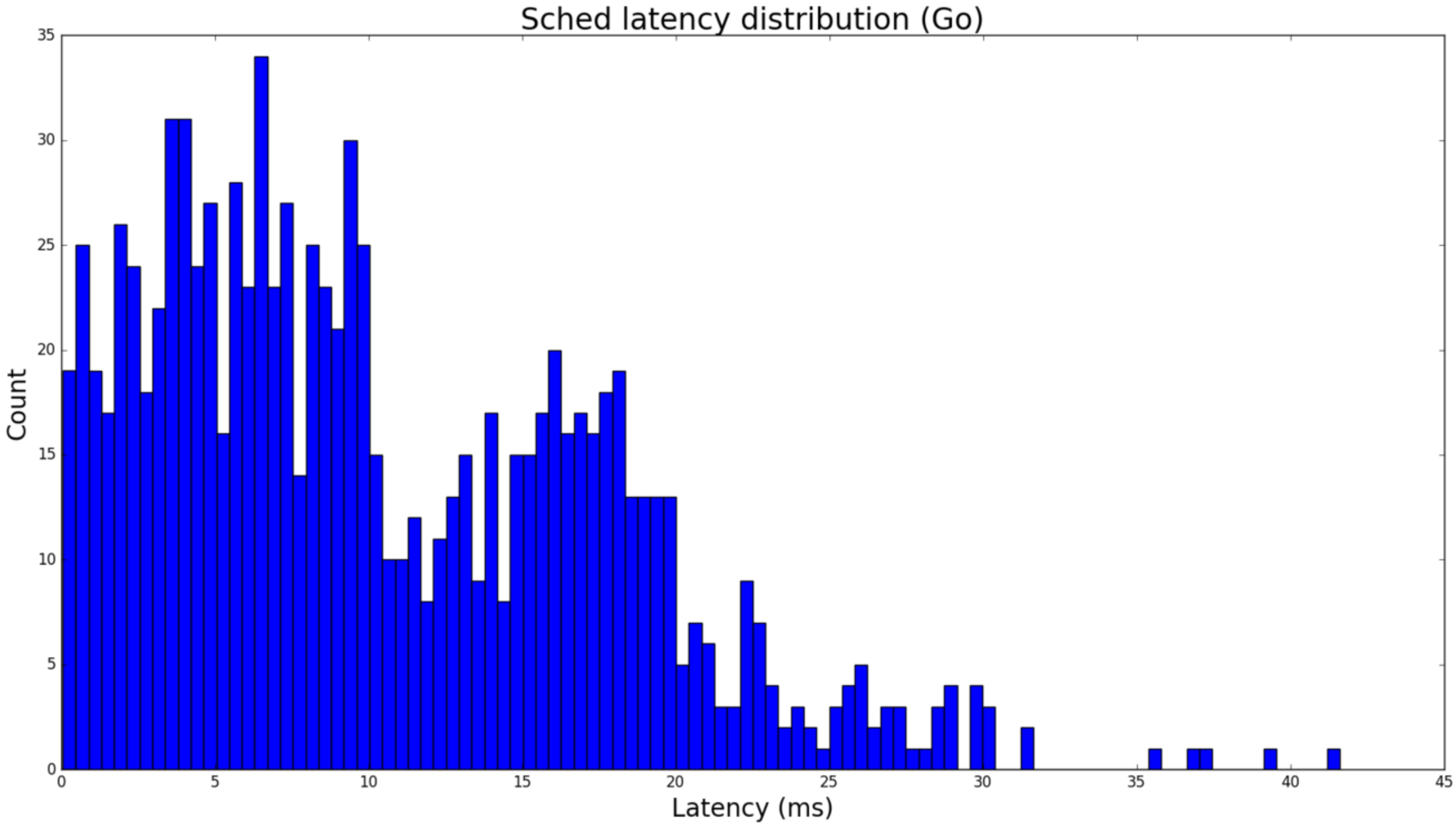
busy tight loop (saturate cores)

A small Go program:

```
func main() {
  for i := 0; i < 4; i++ {
    go func() { for { time.Now() }}();
  }

  for i := 0; i < 1000; i++ {
    target_delay_ns := rand.Intn(1000 * 1000 * 1000)
    ts := time.Now()
    time.Sleep(time.Duration(target_delay_ns) * time.Nanosecond)
    actual_delay_ns = time.Since(ts).Nanoseconds()
    jitter = actual_delay_ns - target_delay_ns
    fmt.Printf("%d\n", target_delay_ns)
  }
}
```

sleep

estimate preemption latency


Sched latency distribution (Go)
Count
Latency (ms)
0
5
10
15
20
25
30
35
40
45

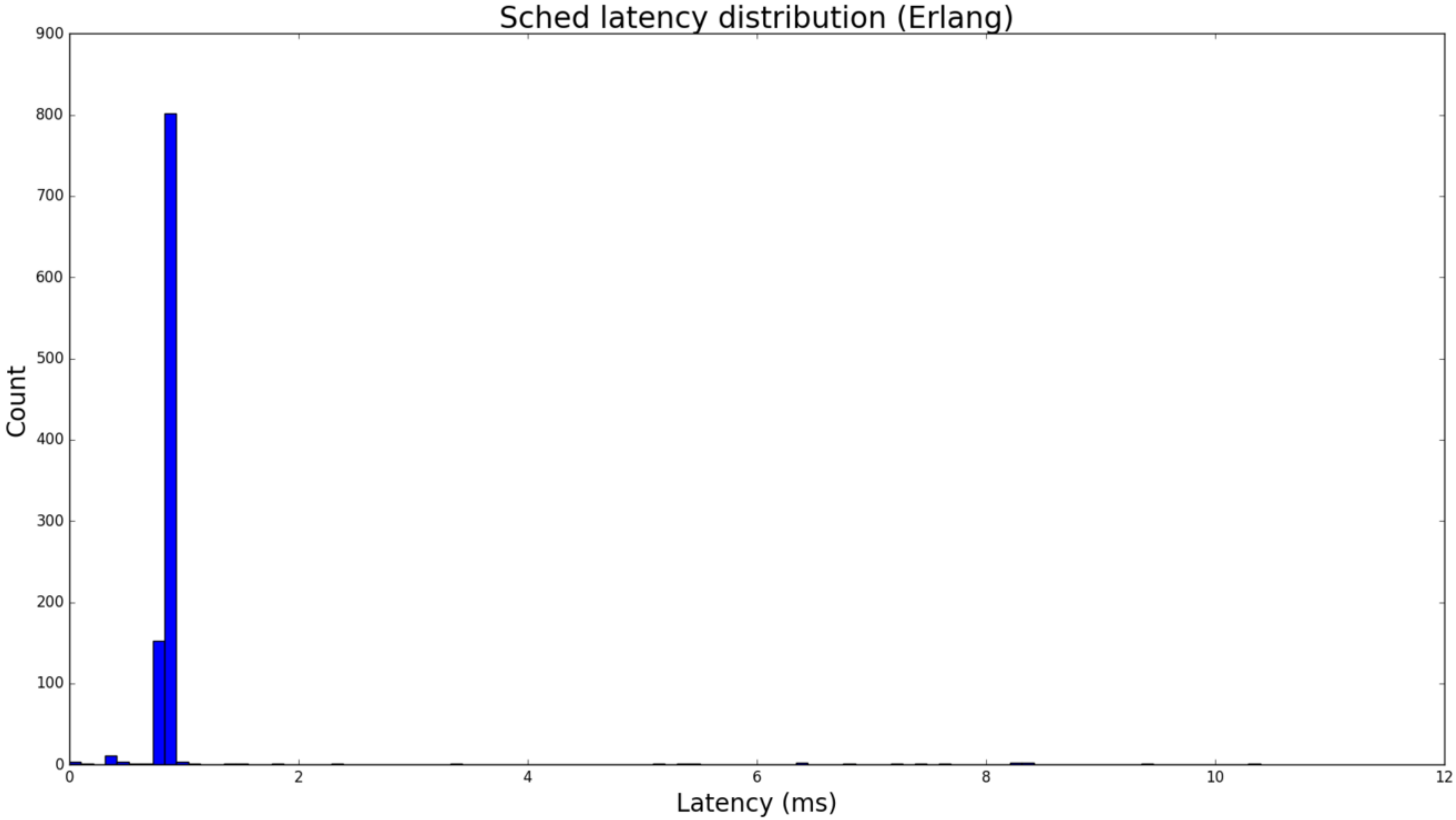
Same deal (Erlang) (okay actually Elixir whatever)

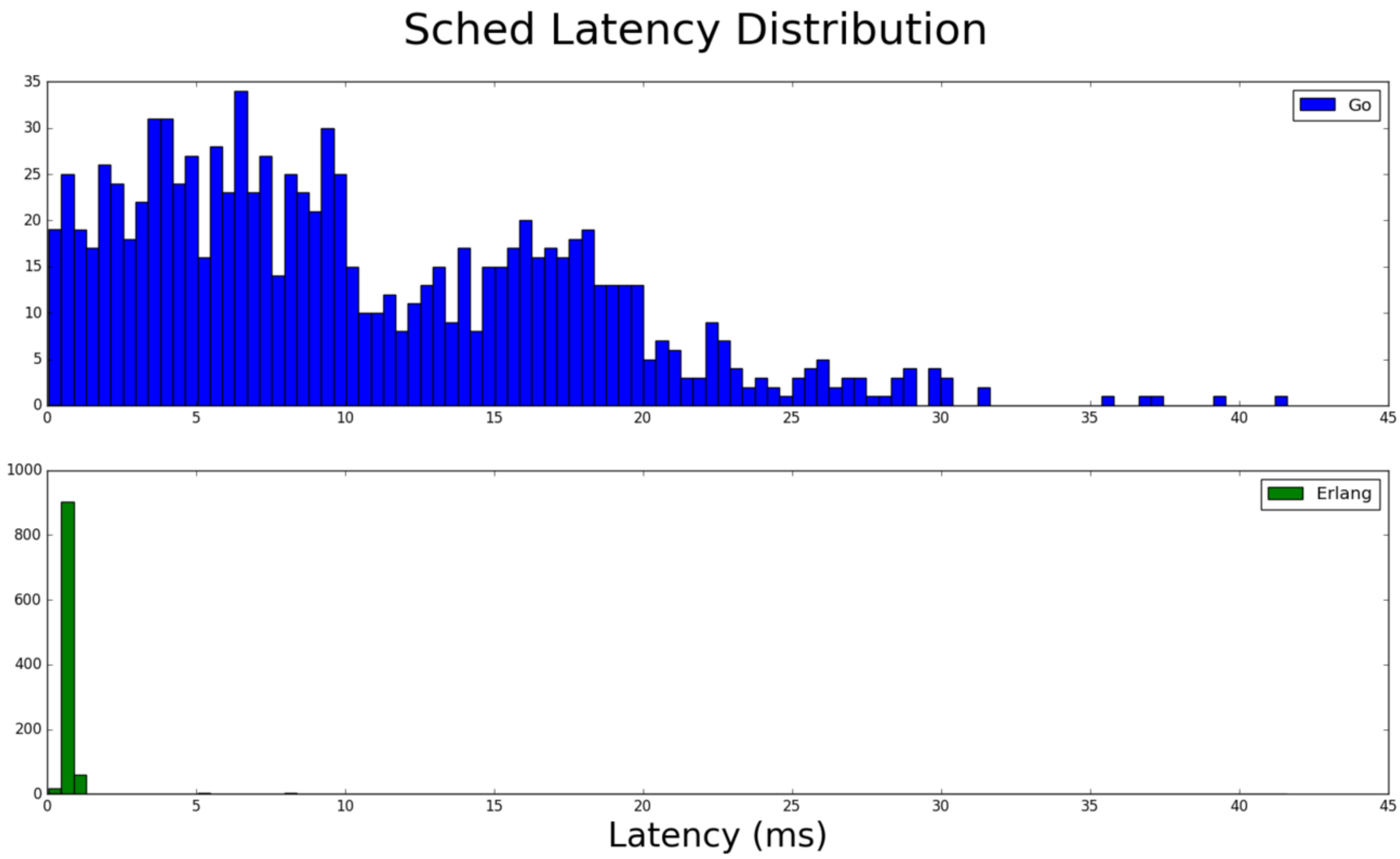
busy tight loop (saturate cores)

sleep

estimate preemption latency

```
def block(n) do
  n = n + 1
  block n
end

spawn(Preempter, :block, [0])
spawn(Preempter, :block, [0])
spawn(Preempter, :block, [0])
spawn(Preempter, :block, [0])

def preempter(n) when n <= 0 do end
def preempter(n)
  delay_ms = round(:rand.uniform() * 1000)
  start = :os.system_time(:nano_seconds)
  :timer.sleep(delay_ms)
  now = :os.system_time(:nano_seconds)
  IO.puts((now-start) - 1000000 * delay_ms)
  preempter n - 1
end

preempter(1000)
```


Sched latency distribution (Erlang)
Count
Latency (ms)
0
100
200
300
400
500
600
700
800
900
0
2
4
6
8
10
12



Erlang trades throughput for predictable latency.

Go does the opposite.

Lessons

Scalable scheduling: not that mysterious!

Patterns

- Independent runqueues
- Load balancing
- Preemption at safepoints

Decisions

- Granular priorities vs implementation simplicity
- Latency predictability vs baseline overhead

Thank you!
Any questions?

slides: speakerdeck.com/emfree/runtime-scheduling

freemaneben@gmail.com

Scheduler observability

- *GODEBUG=schedtrace* : periodically output scheduler statistics

```
$ GODEBUG=schedtrace=100 go run main.go
SCHED 0ms: gomaxprocs=4 idleprocs=3 threads=5 spinningthreads=0 idlethreads=3 runqueue=0 [0 0 0 0]
SCHED 103ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=0 runqueue=20 [49 10 9 8]
SCHED 204ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=0 runqueue=40 [44 5 4 3]
SCHED 305ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=0 runqueue=33 [39 0 11 13]
SCHED 405ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=0 runqueue=43 [34 5 6 8]
SCHED 506ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=0 runqueue=63 [29 0 1 3]
SCHED 606ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=0 runqueue=40 [24 12 10 10]
SCHED 707ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=0 runqueue=60 [19 7 5 5]
SCHED 807ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=0 runqueue=80 [14 2 0 0]
SCHED 908ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=0 runqueue=49 [9 11 16 11]
SCHED 1009ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=1 runqueue=70 [4 6 11 6]
SCHED 1109ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=1 runqueue=67 [22 1 6 1]
SCHED 1210ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=1 runqueue=50 [18 16 1 12]
SCHED 1310ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=1 runqueue=53 [13 11 13 7]
SCHED 1411ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idlethreads=1 runqueue=71 [9 7 8 2]
```

runqueue depths

Scheduler observability

- *go tool trace*: Multipurpose program execution tracer

```
$ go tool test -trace trace.out                           # Trace tests, or
$ curl -o trace.out http://localhost/debug/pprof/trace?seconds/5   # Trace a running program
$ go trace trace.out                                      # Run trace viewer
```