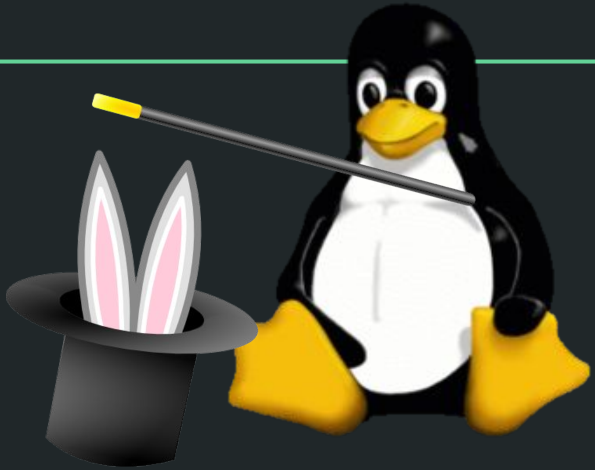
RCU in 2019

Joel Fernandes <joel@joelfernandes.org>
Google.

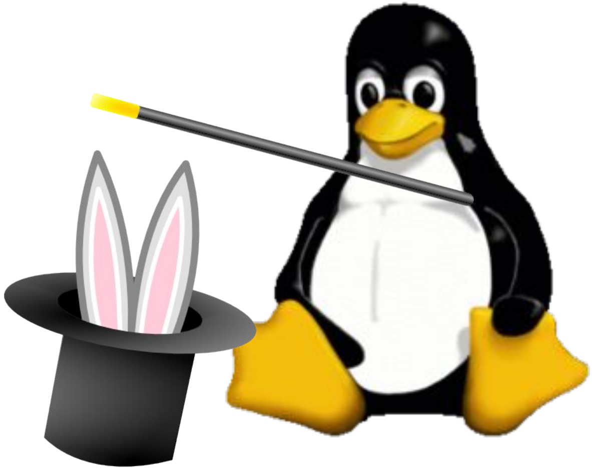


What I do? Recent work history

- Joined Google in 2016 : Task Scheduler , BPF for tracing etc.
  - Complex stuff

What I do? Recent work history

- 2017: Start exploring RCU internals:
    - Very complex stuff



What I do? Recent work history

- 2019: Parenting a 2 year old
  - Very Very complex stuff

How I got started with RCU?

- Worked on Linux for a decade or so.
- People who understand RCU internals ... < 7 : Opportunity!!
- Making sense of RCU traces, logs, concepts.

Time to put mysteries to end.

What am I doing with RCU now?

- Helping community / company with RCU issues, concepts, improvements, reviewing.
- New feature development.

Who am I ; and how I got started with RCU?

Started questioning RCU's internal design (~2 years ago)

Paul McKenney says… "Here is your nice elegant little algorithm"

Who am I ; and how I got started with RCU?

Paul McKenney says… “Here is your nice elegant little algorithm equipped to survive in the Linux Kernel”

Credits

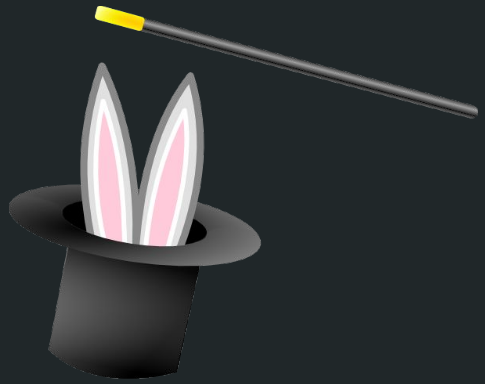
RCU is the great decades-long work of Paul Mckenney and others. I am relatively new on the scene (~ 2 years).

Agenda

- Introduction
- TREE RCU
- RCU Flavor consolidation
  - Performance
  - Scheduler Deadlock fixes
- TASKS RCU
- List RCU API improvements (if time permits)

Introduction

The basic idea of RCU



Intro: Typical RCU workflow

Say you have some data that you have to share between a reader/writer section.

```
struct shared_data {
    int a;
    long b;
};
```

```
int reader(struct shared_data *sd) {
    if (sd->a)
        return sd->b;
    return 0;
}
```

```
int writer(struct shared_data *sd) {
    sd->b = 1;
    sd->a = 2;
}
```

Intro: Typical RCU workflow

One way is to use a reader-writer lock.

```
int reader(struct shared_data *sd) {
    read_lock(&sd->rwlock);
    if (sd->a)
        ret = sd->b;
    read_unlock(&sd->rwlock);
    return ret;
}
```

```
void writer(struct shared_data *sd) {
    write_lock(&sd->rwlock);
    sd->b = 1;
    sd->a = 2;
    write_unlock(&sd->rwlock);
}
```

Some concepts first: RCU read-side critical section

```
struct shared_data *global_sd;

int reader() {
    rcu_read_lock();
    sd = rcu_dereference(global_sd);
    if (sd->a)
        ret = sd->b;
    rcu_read_unlock();
    return ret;                                   }
}
```

Some concepts first: What is a quiescent state?

A state that an entity (CPU or task) passes through that is impossible within an RCU-read side critical section.

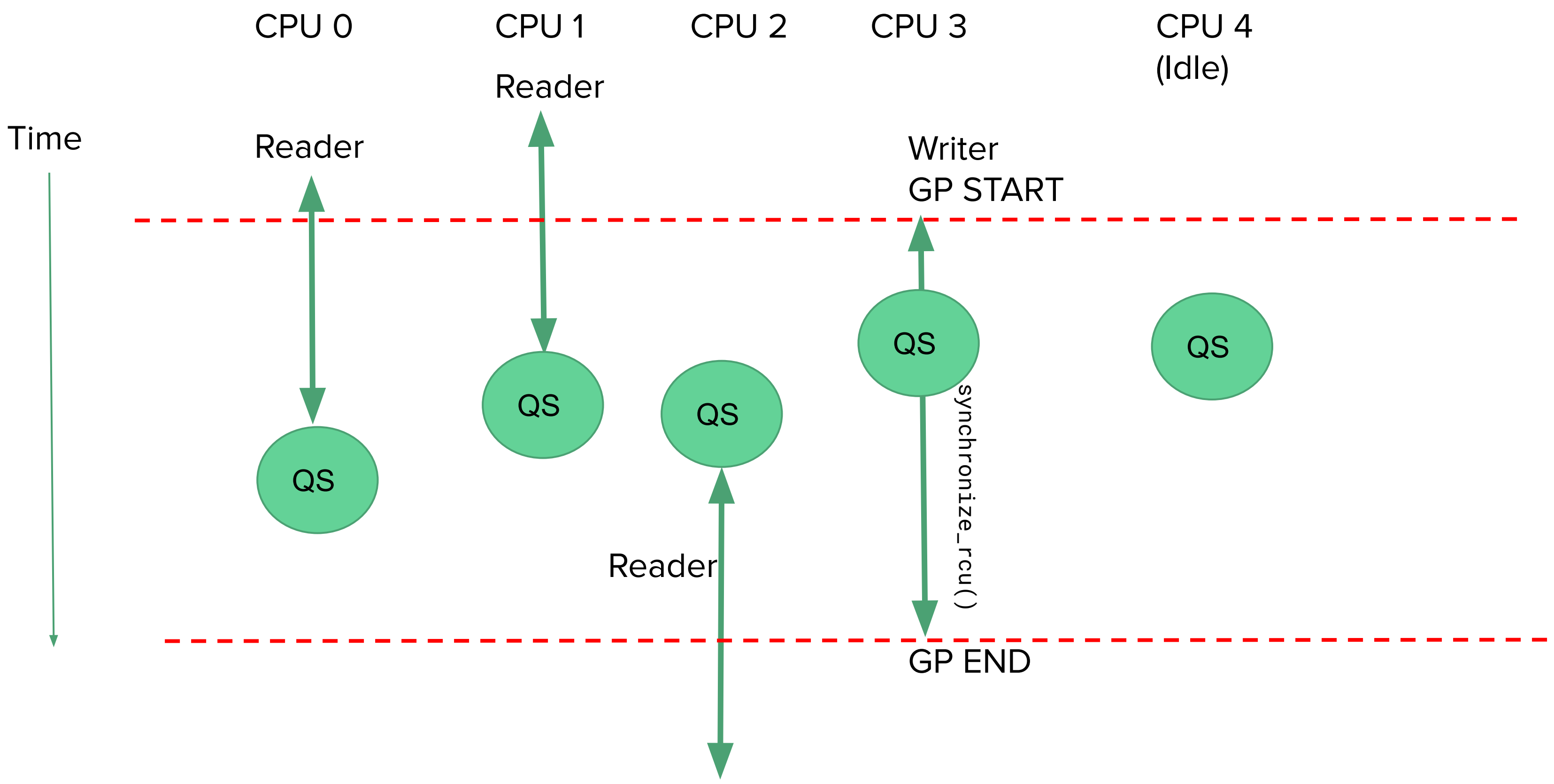
Some concepts first: What is a Grace period?

A waiting period where we :

- start the wait - by writer
- end the wait - all entities have passed through the Quisecent state.

Finish GP wait means all readers STARTED PRIOR TO WAIT have finished.

Some concepts first: What is a Grace period?



Intro: Typical RCU workflow

Say you have some data that you have to share between a reader/writer section.

```
struct shared_data {
    int a;
    long b;
};
```

```
int reader(struct shared_data *sd) {
    if (sd->a)
        return sd->b;
    return 0;
}
```

```
int writer(struct shared_data *sd) {
    sd->b = 1;
    sd->a = 2;
}
```

Intro: Typical RCU workflow

One way is to use a reader-writer lock.

```
int reader(struct shared_data *sd) {
    read_lock(&sd->rwlock);
    if (sd->a)
        ret = sd->b;
    read_unlock(&sd->rwlock);
    return ret;
}
```

```
void writer(struct shared_data *sd) {
    write_lock(&sd->rwlock);
    sd->b = 1;
    sd->a = 2;
    write_unlock(&sd->rwlock);
}
```

Intro: Typical RCU workflow: or use RCU...

```
struct shared_data *global_sd;

int reader() {
    rcu_read_lock();
    struct shared_data sd =
        rcu_dereference(global_sd);

    if (sd->a)
        ret = sd->b;
    rcu_read_unlock();


    return ret;
}
```

```
void writer() {
    struct shared_data *sd, *old_sd;
    spin_lock(&sd->lock);
    old_sd = rcu_dereference(global_sd);
    sd = kmalloc(sizeof(struct shared_data);
    *sd = *old_sd;
    sd->a = 2;
    rcu_assign_pointer(global_sd, sd);
    spin_unlock(&sd->lock);
    synchronize_rcu();
    kfree(old_sd);
}
```

Intro: Fastest Read-mostly Primitive

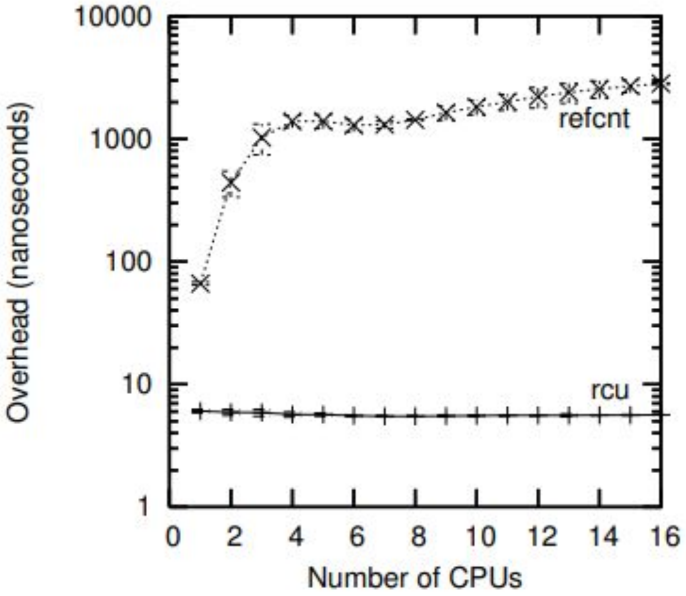


Figure 5: The overhead of entering using RCU as a reference count compared to the overhead of using a shared integer.

Intro: Writes are costly

What is cost?

- Grace period cycle.
- Time.

But...

- Writes are costly but per-update cost is amortized.
- 1000s or millions of updates can share GP.

Intro: When to use RCU vs something else?

- If data structure is updated less than 10% of time.
- Need it for other special use cases.
  - Check Documentation/RCU/checklist.txt
- Many more use cases:
  - Wait for completion, locking, refcount implementation etc.
  - Check RCU decades later paper:

    https://pdos.csail.mit.edu/6.828/2018/readings/rcu-decade-later.pdf

Toy #1 based on ClassicRCU (Docs: WhatIsRCU.txt)

```
Classic RCU (works only on PREEMPT=n kernels):

#define rcu_dereference(p) READ_ONCE(p);
#define rcu_assign_pointer(p, v) smp_store_release(&(p), (v));

void rcu_read_lock(void) { }
void rcu_read_unlock(void) { }

void synchronize_rcu(void)
{
        int cpu;
        for_each_possible_cpu(cpu)
            run_on(cpu);
}
```

QUIZ: Why will this not work on a preemptible kernel?

QUIZ: What are the drawbacks of this?

Ok.. Now let's see the bear!

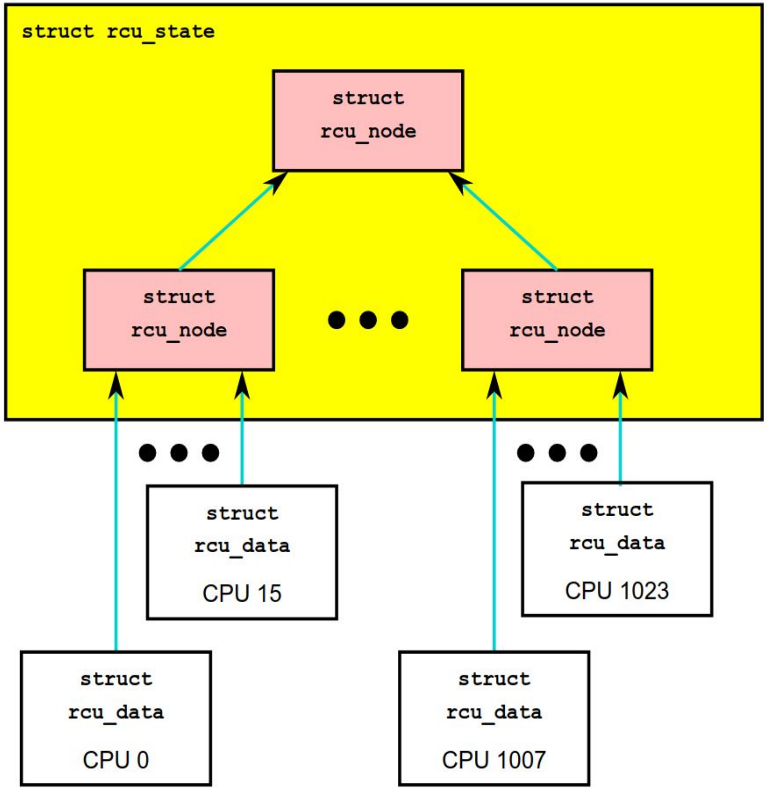
TREE_RCU

TREE_RCU is the most complex and widely used flavor of RCU.

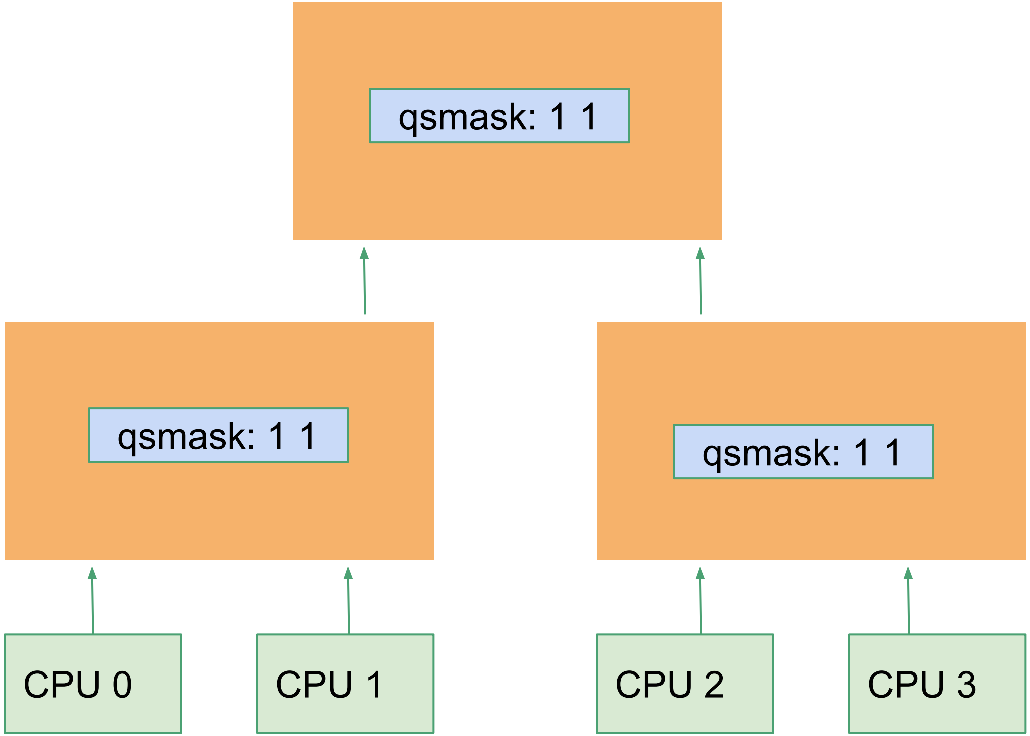
“ If you are claiming that I am worrying unnecessarily, you are probably right. But if I didn't worry unnecessarily, RCU wouldn't work at all! ”
— Paul McKenney

There’s also other specialized flavors: TINY RCU, SRCU, TASKS.

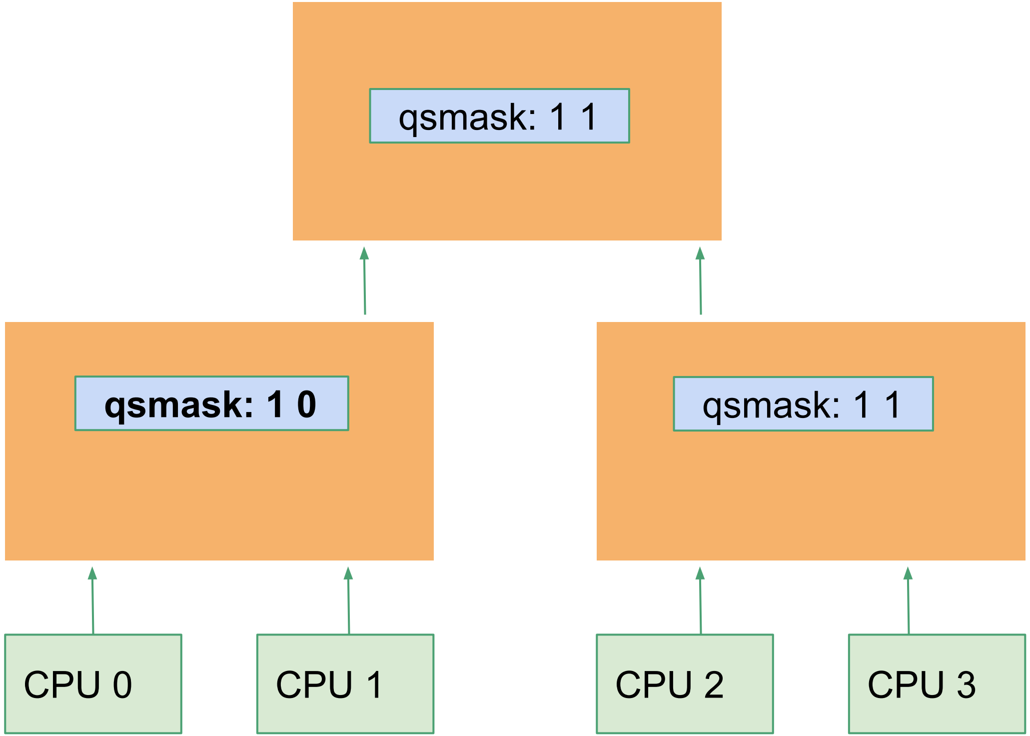
Intro: How TREE_RCU works?



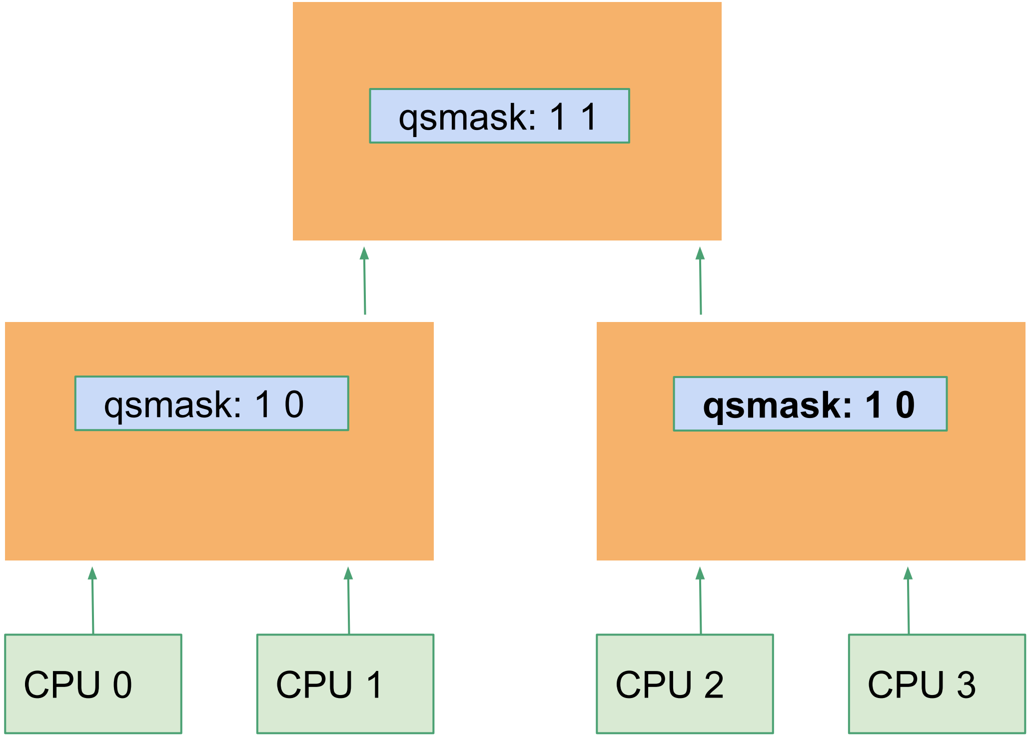
TREE_RCU example: Initial State of the tree



TREE_RCU example: CPU 1 reports QS

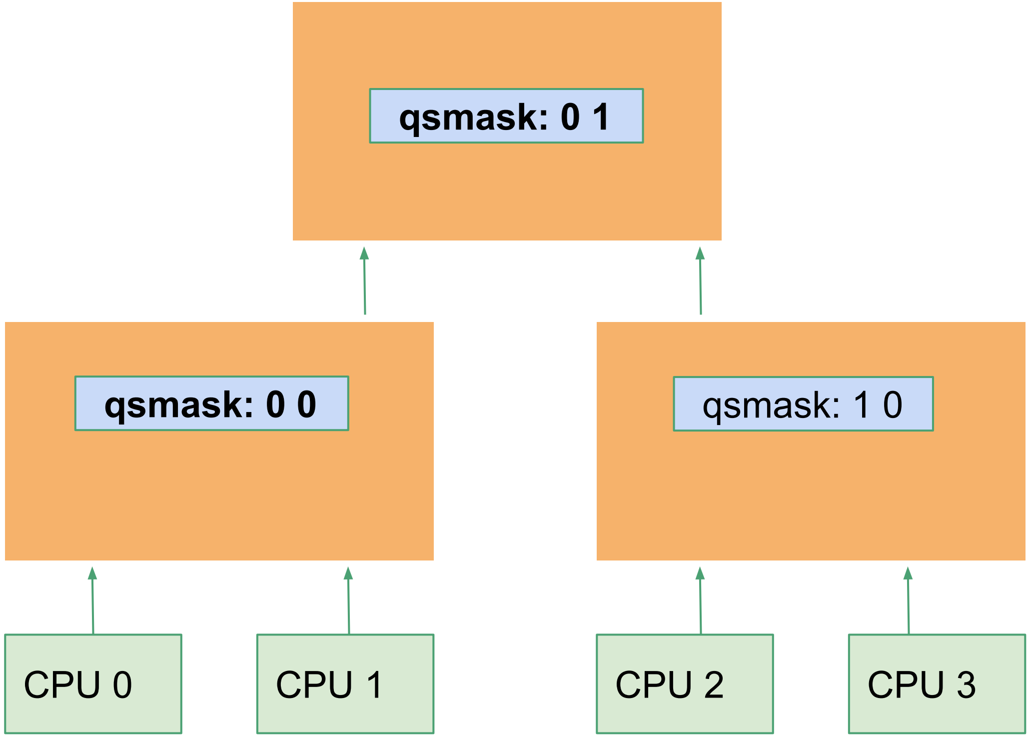


TREE_RCU example: CPU 3 reports QS



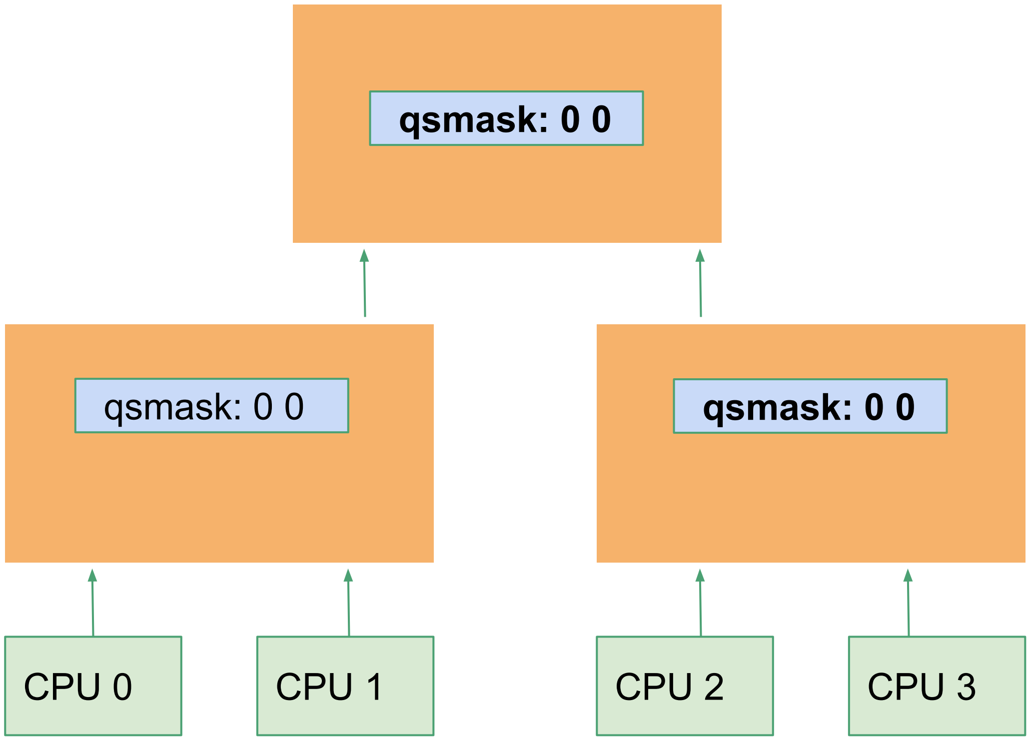
(Notice that the 2 QS updates have proceeded without any synchronization needed)

TREE_RCU example: CPU 0 reports QS



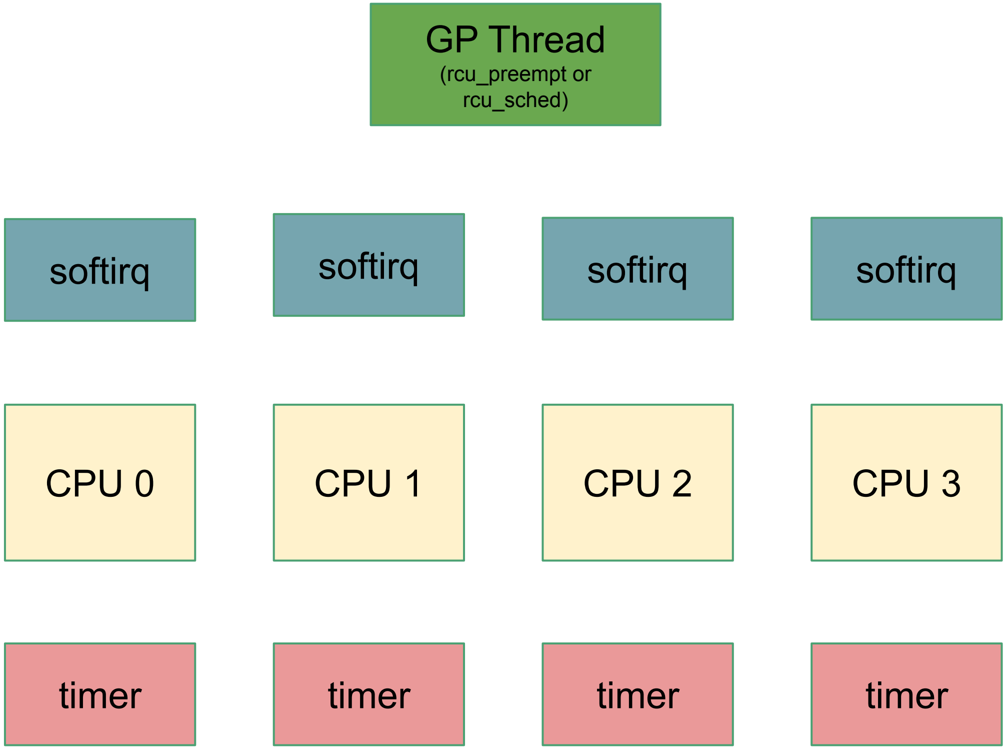
(Now there has been an update at the root node)

TREE_RCU example: CPU 2 reports QS

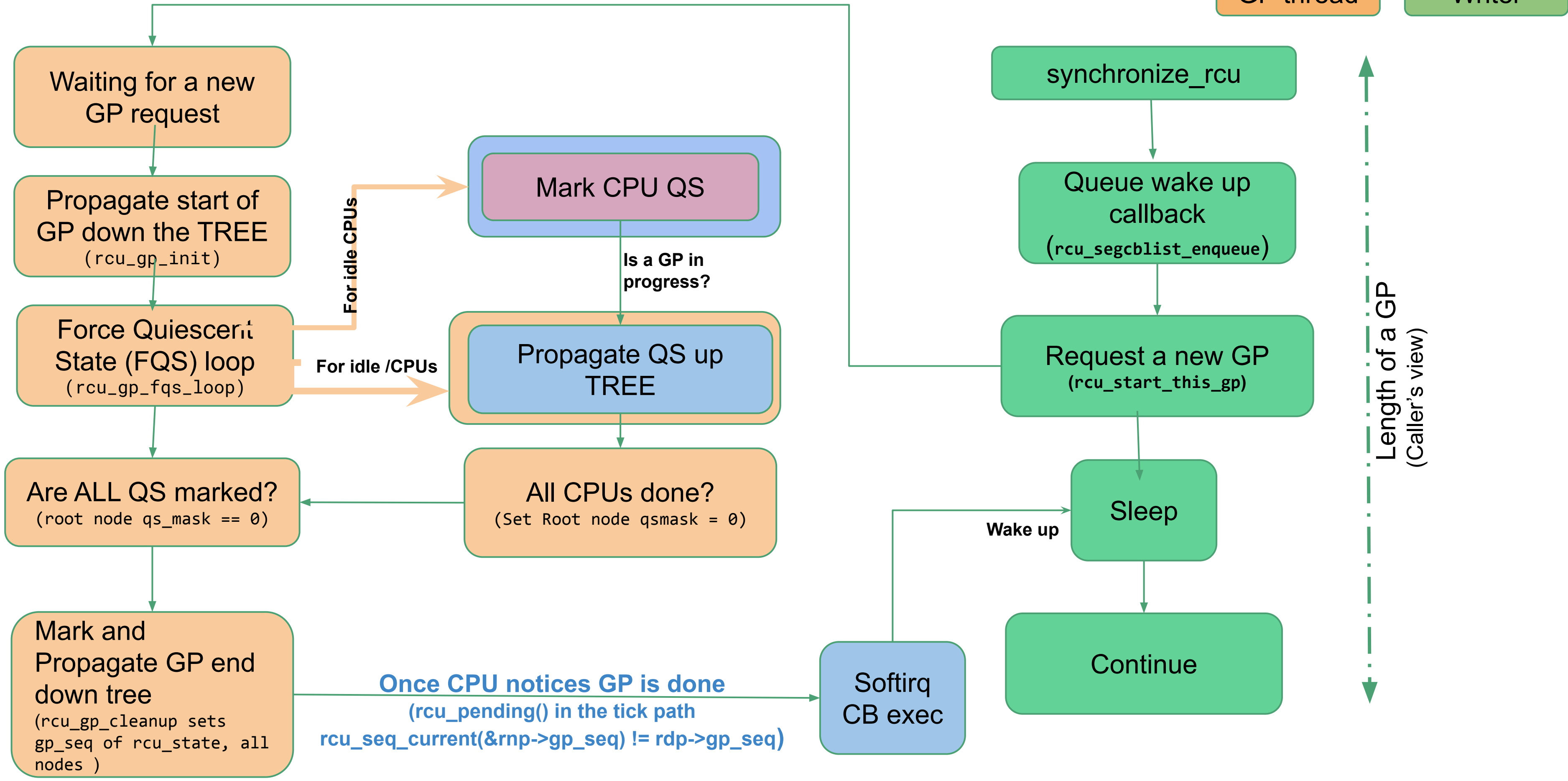


(notice that only 2 global updates were needed instead of 4. On a system with 1000s of CPUs, this will be at most 64)

Intro: Components of TREE RCU (normal grace period)



Intro: Life Cycle of a grace period



Implied QS

- CPU is already in a certain state:
  - IDLE
  - OFFLINE
  - USER MODE

Light weight QS

- Does not end the grace period yet.
- Just marks CPU-**locally** and someone ELSE reports up the tree LATER.

What happens?

- Start of GP sets **`rcu_data::cpu_no_qs`**
- Lightweight QS reporting clears it which says CPU is DONE.

Where does it happen?

- Scheduler tick
- Context switch

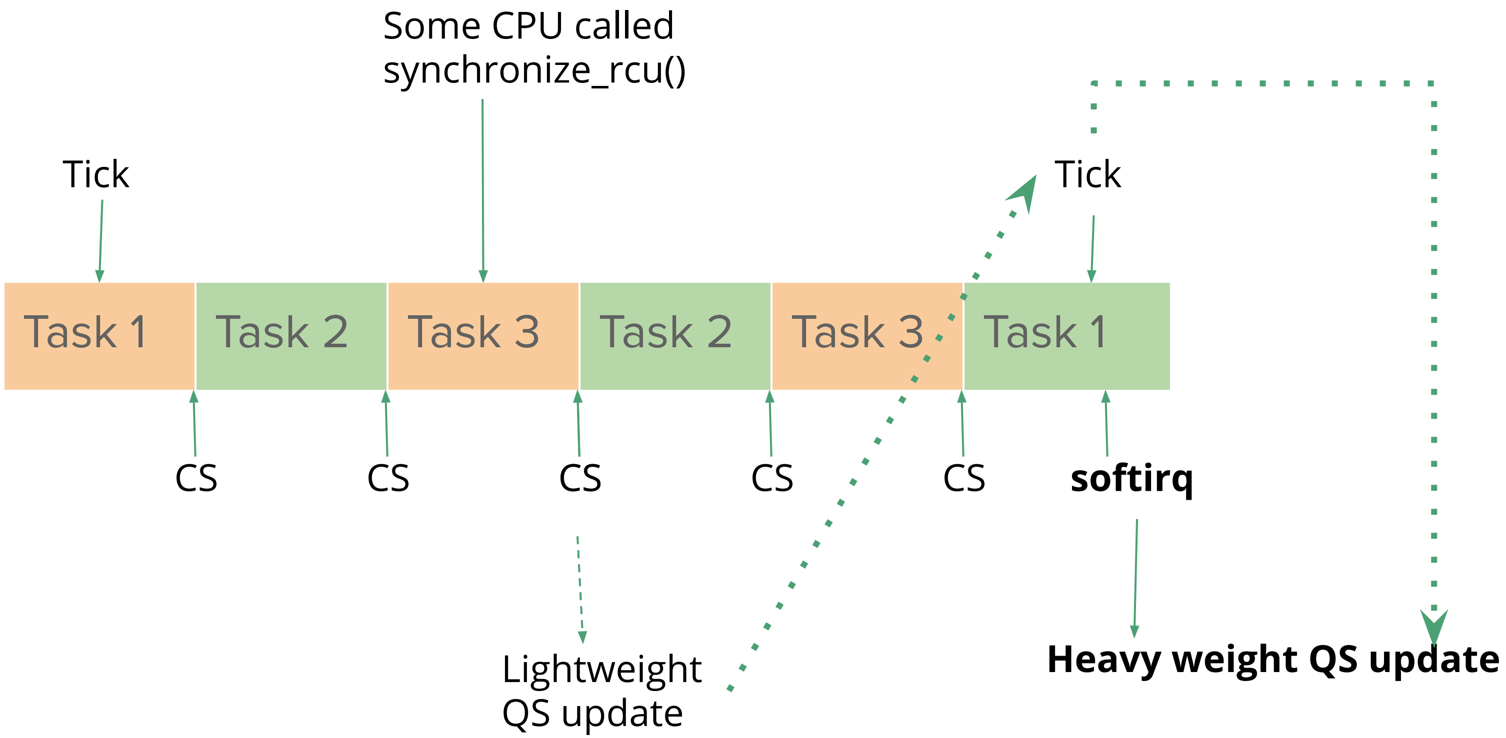
Heavy weight QS

- Can end the grace period due to tree report.
- Happens less often : Uses mem barriers, atomics, locking etc.
- Happens only AFTER the light weight QS.

Where does it happen?

- softirq
- fqs_loop
  - Due to transition to NOHZ - idle/user mode
  - cond_resched() in PREEMPT=n kernels
- rcu_read_unlock_special() in some cases.

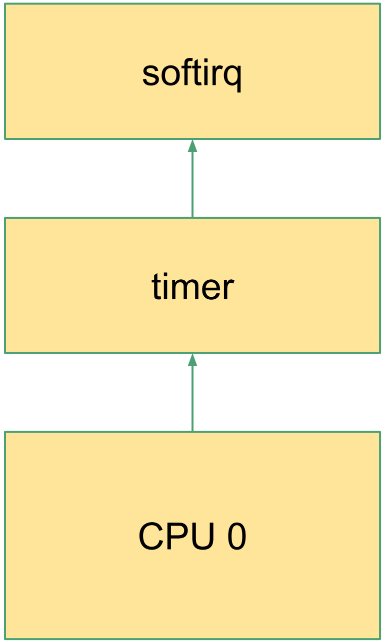
Example of light weight and heavy weight QS



Intro: What happens in softirq ?

Per-CPU Work:

- **QS reporting for CPU and propagate up tree.**
- **Invoke any callbacks whose GP has completed.**
    - **(TODO: Check that if there are no callbacks queued on CPU, can we skip softirq?)**

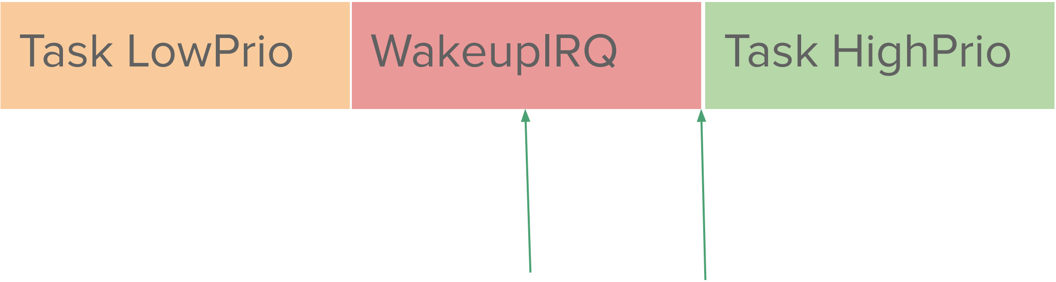


Caveat about callbacks queued on offline CPUs:

PaulMck says:

> And yes, callbacks do migrate away from non-offloaded CPUs that go
> offline. But that is not the common case outside of things like
> rcutorture.

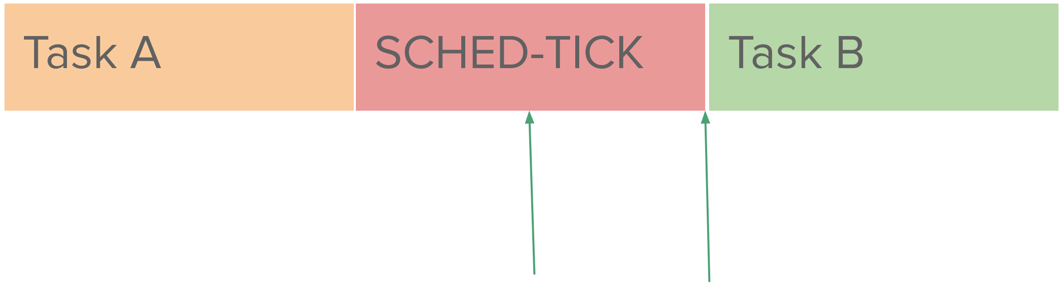
The magic of {TIF,PREEMPT}_NEED_RESCHED



TaskHighPrio wakes up in IRQ handler and task’s **TIF_NEED_RESCHED** flag now set (low<hi)

IRQ return causes Entry into scheduler And CONTEXT SWITCH

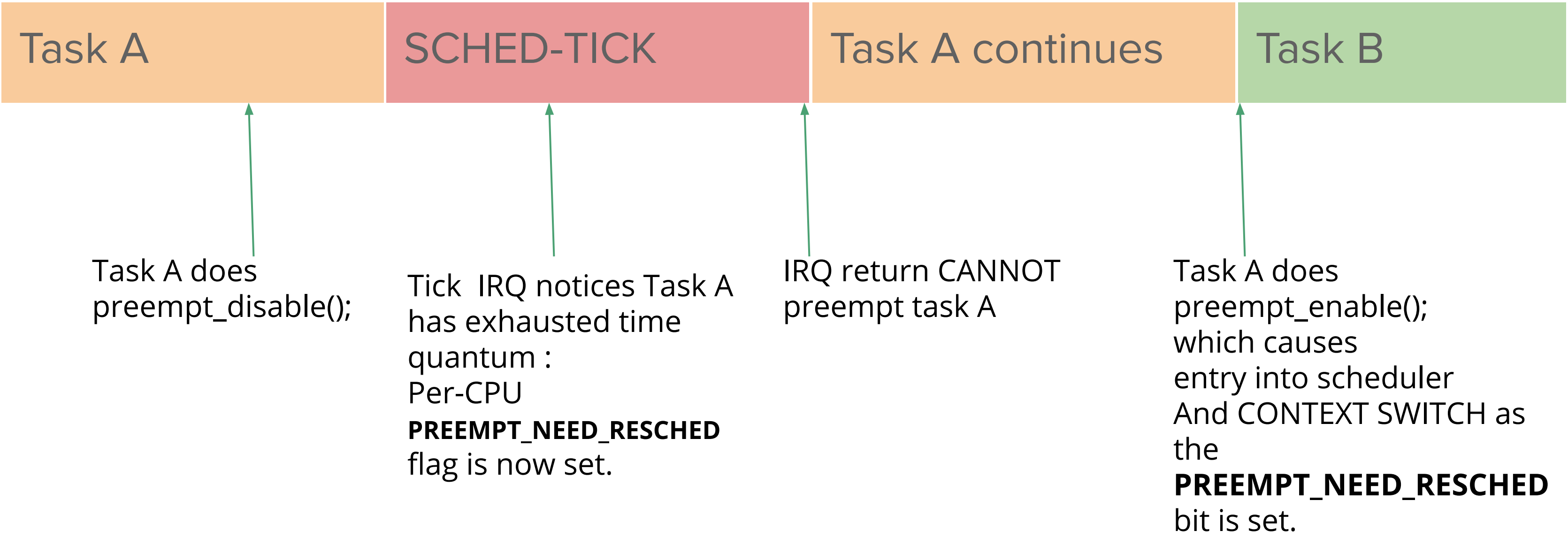
The magic of {TIF,PREEMPT}_NEED_RESCHED



Tick IRQ notices
Task A has
exhausted time
quantum :
Task’s
TIF_NEED_RESCHED
flag now set

IRQ return causes
Entry into scheduler
And CONTEXT SWITCH

The magic of {TIF,PREEMPT}_NEED_RESCHED



Intro: Grace Period has started, what's RCU upto?

At around 100ms:

GP THREAD

Set Per-CPU urgent_qs flag

Task is in kernel mode

Sched-Tick

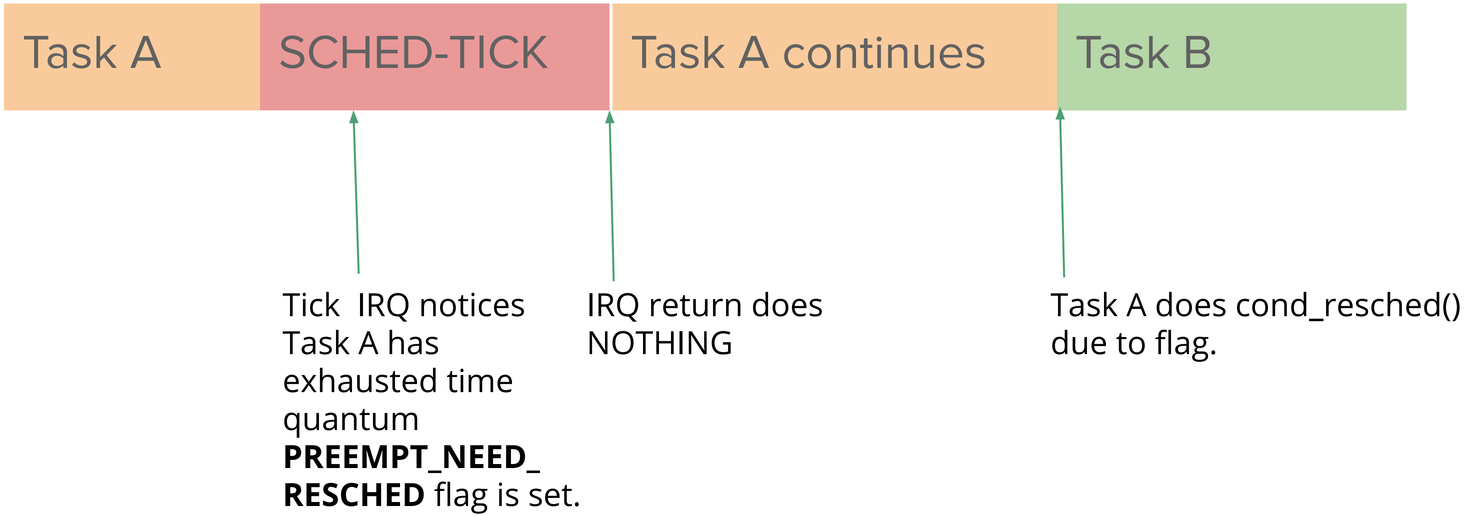
Set task's need_resched flag.

Enter scheduler

Report QS

(Note: Scheduler entry can happen either in next TICK or next preempt_enable())

!CONFIG_PREEMPT kernels and cond_resched() :



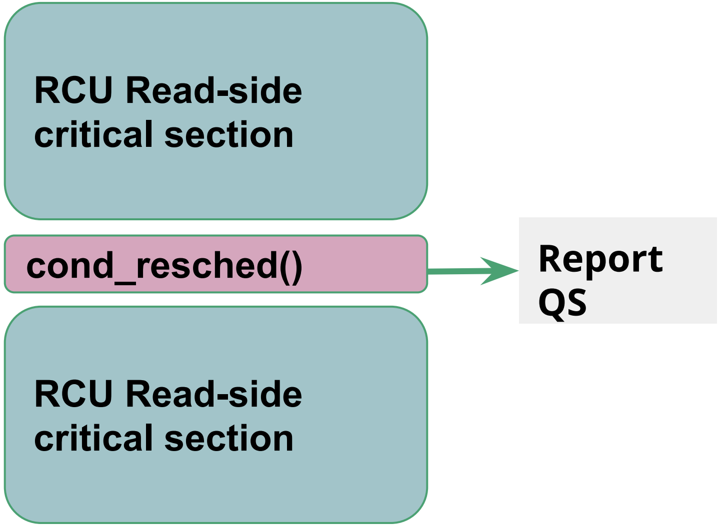
RULE:

cond_resched() cannot be in rcu reader section.

BAD:

```
rcu_read_lock();
cond_resched();
rcu_read_unlock();
```

We can use that to our advantage:



Intro: Grace Period has started, what's RCU upto?

At around 200ms: Put cond_resched() on steroids:

GP THREAD

Request help from cond_resched() for PREEMPT=n

(by setting Per-cpu need_heavy_qs flag)

RCU Read-side critical section

cond_resched()

RCU Read-side critical section

Report HEAVY-WEIGHT QS by "fake" dyntick idle transition forcing the fqs_loop to mark CPU to be in QS.

Intro: Grace Period has started, what’s RCU upto?

`At around 300ms turn on TICK for nohz_full kernel mode:`

GP THREAD

Send IPIs to CPUs that are still holding up

RCU read-side critical section

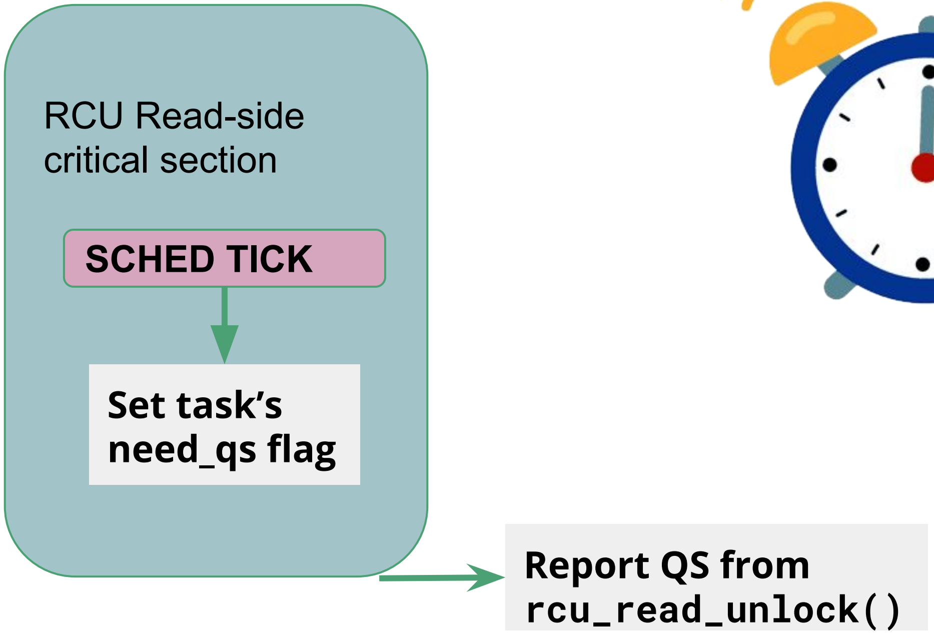
IPI handler Turns on TICK

SCHED TICK

SCHED TICK → Report QS

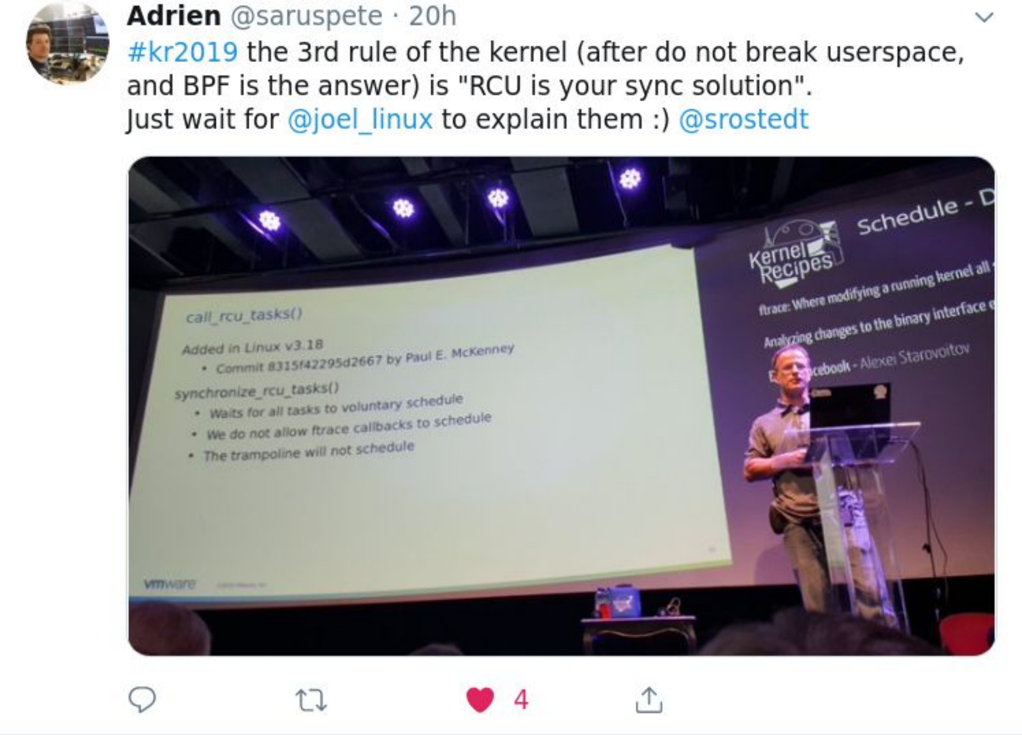
Intro: Grace Period has started, what’s RCU upto?

`At around 1 second of start of GP:`



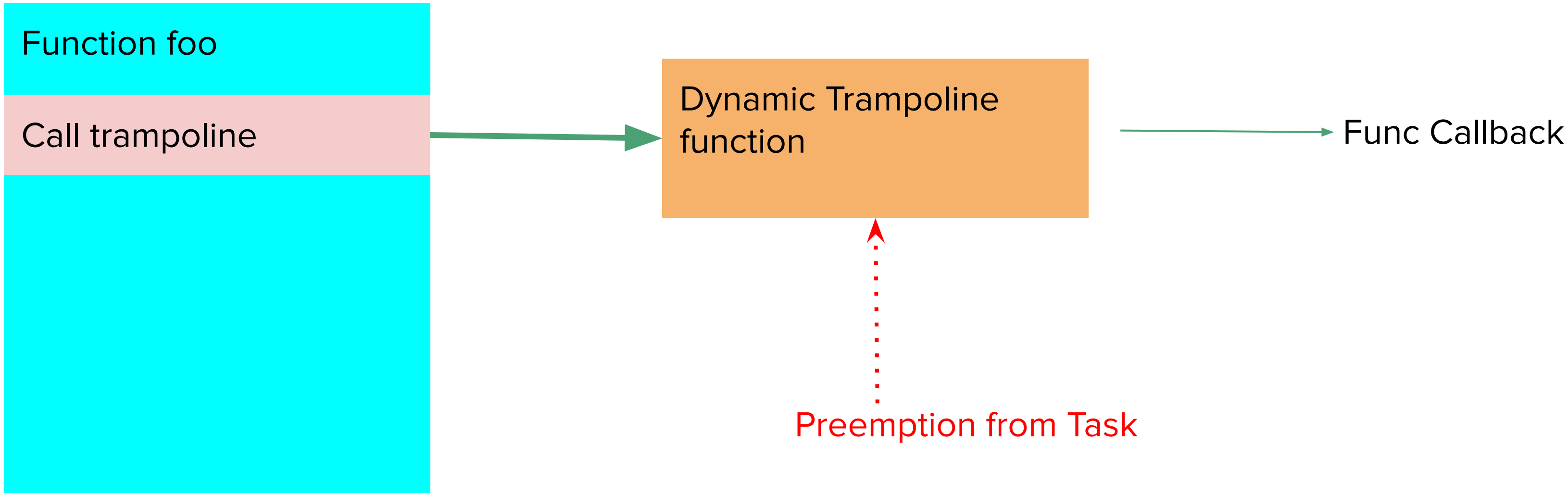
Tasks-RCU

“We all jump on a ~~yellow submarine~~ dynamic trampoline” -- Beatles



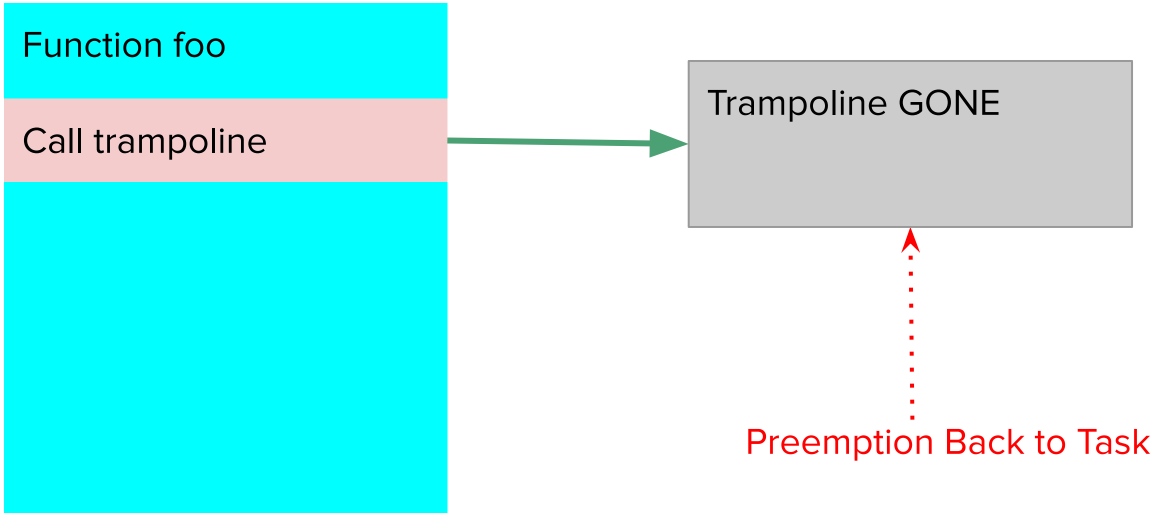
TasksRCU : For dynamic trampolines

Problem: Ftrace allocates dynamic trampolines for callbacks.



TasksRCU : For dynamic trampolines

Problem: Ftrace allocates dynamic trampolines for callbacks.



BOOM!

Solution: TasksRCU

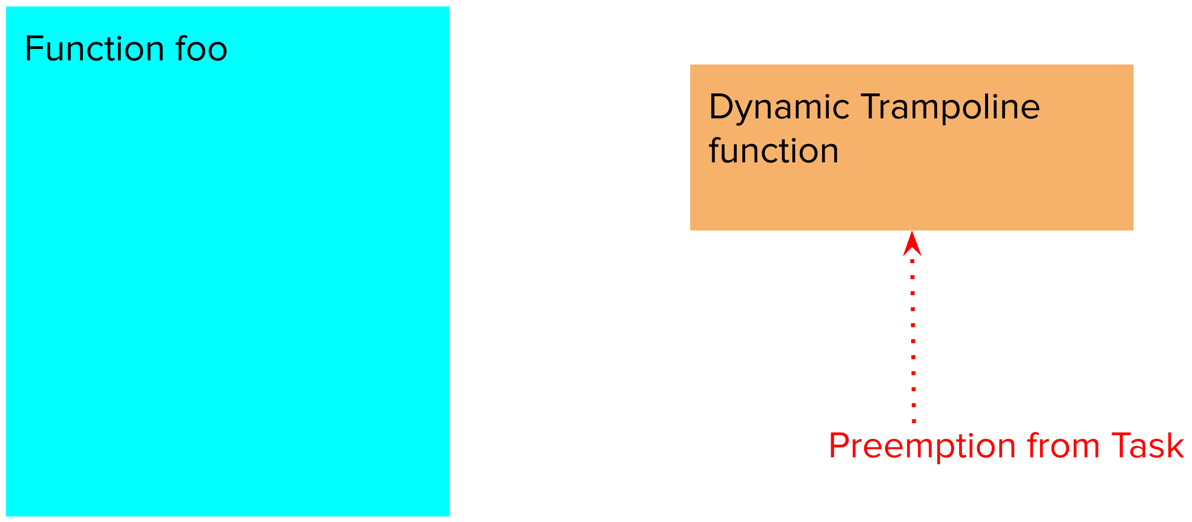
Read-side critical section: Trampoline

Quiescent state: Task blocking

Grace Period: Wait for all tasks to block

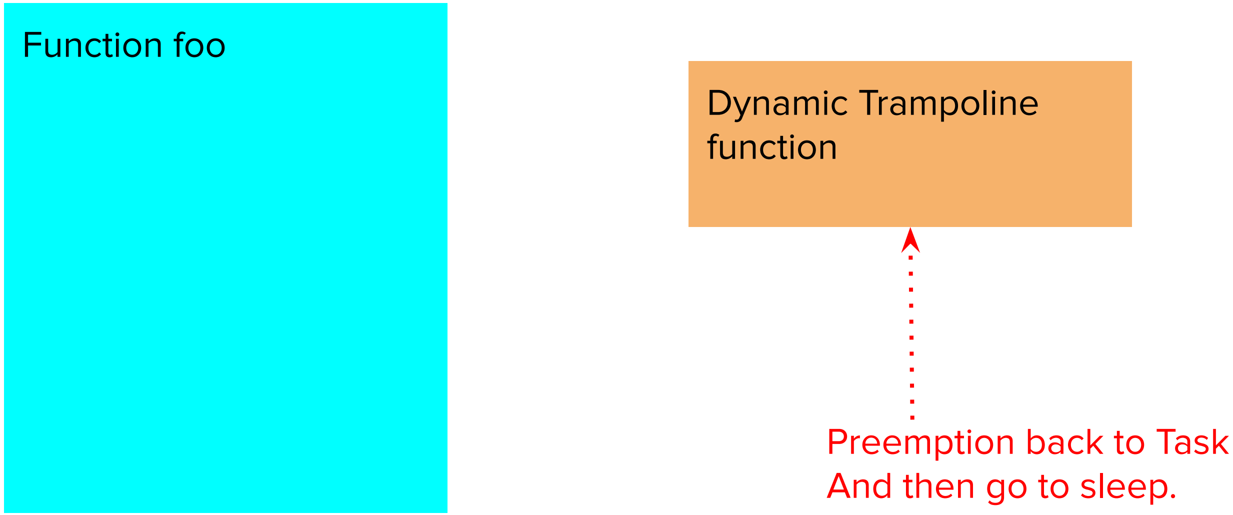
TasksRCU : For dynamic trampolines

Solution: Disconnect trampoline, but don't free it yet.



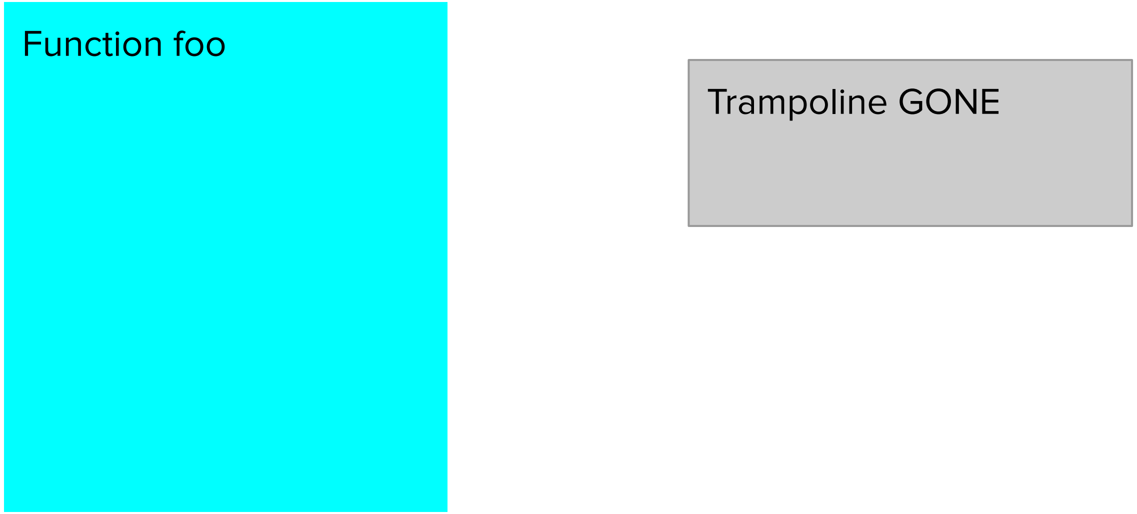
TasksRCU : For dynamic trampolines

Solution: Wait for all tasks to block (synchronize_rcu_tasks()).



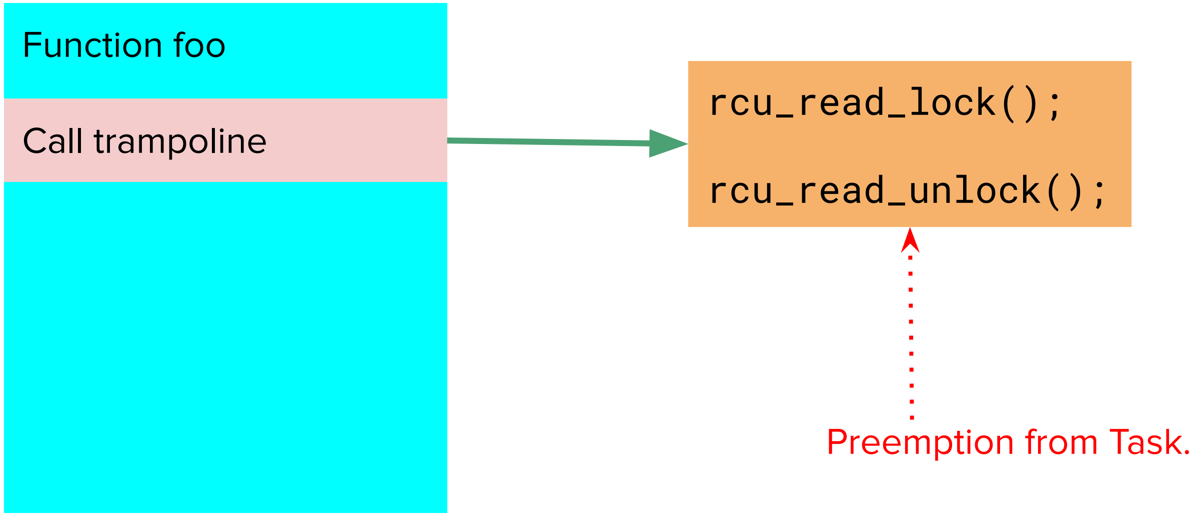
TasksRCU : For dynamic trampolines

Solution: Free trampoline



TasksRCU : For dynamic trampolines

Why wouldn't rcu_read_lock() with synchronize_rcu() work?



RCU Flavor consolidation

Different RCU “flavors”

RCU-sched

Reader Section: !preemptible();

Entry into RCU read-side critical section:

a. `rcu_read_lock_sched();`
b. `preempt_disable();`
c. `local_irq_disable();`
d. IRQ entry.

Different RCU "flavors"

RCU-bh

Reader Section: Bottom half disable

Entry into RCU read-side critical section:

a. `rcu_read_lock_bh();`

b. `local_bh_disable();`

c. SoftIRQ entry.

Different RCU “flavors”

RCU-preempt

Reader section:

Marked by `rcu_read_lock() and rcu_read_unlock()` pair.

Preemption allowed in reader , blocking not allowed (unless RT patchset).

RCU Flavor Consolidation: Why? Reduce APIs

Problem:

1. Too many APIs for synchronization. Confusion over which one to use!
   a. `For preempt flavor: call_rcu()` and `synchronize_rcu()`.
   b. `For sched: call_rcu_sched()` and `synchronize_rcu_sched()`.
   c. `For bh flavor: call_rcu_bh()` and `synchronize_rcu_bh()`.
2. Duplication of RCU state machine for each flavor …
3. Too many GP threads.

Now after flavor consolidation: Just `call_rcu()` and `synchronize_rcu()`.

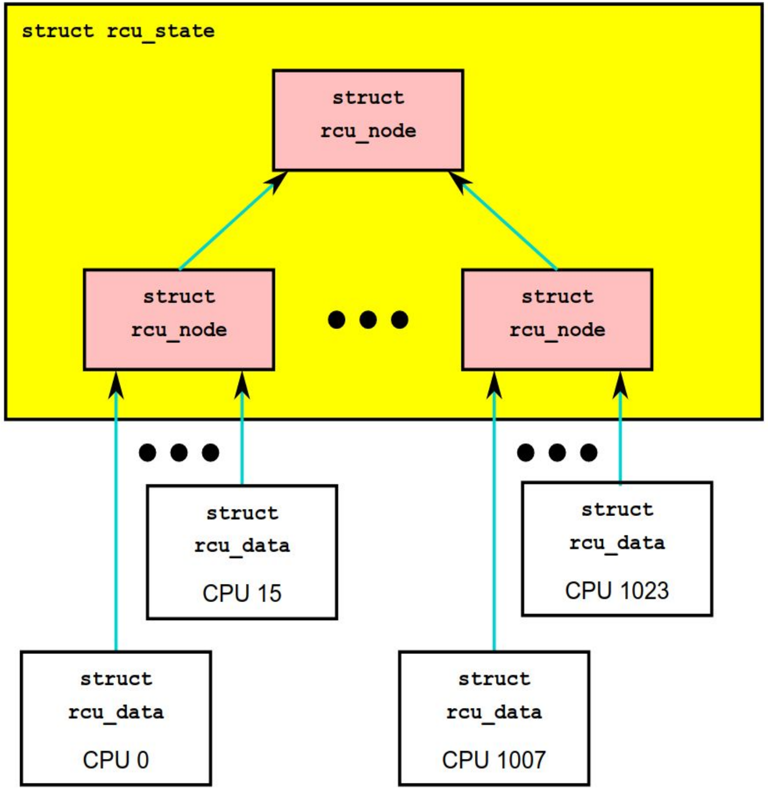
RCU Flavor Consolidation: Why? Changes to rcu_state

Why?

- 3 -> 1 rcu_state structures.
- 3 -> 1 GP thread and state machines.

Advantages:

- Less resources!
- Less code!



Remember : an RCU reader taking a long time can delay a grace period

CPU 0

```
/* This is start of an RCU reader! */
rcu_read_lock();


/* This is end of an RCU reader! */
rcu_read_unlock();
```

CPU 1

```
/* Called after CPU 0’s preempt_disable() */
synchronize_rcu();


/* Executes only much later! */
some_func();
```

Before consolidation: Grace periods were separated, for example...

CPU 0

```
/* This is start of an RCU reader! */
preempt_disable();

/* This is end of an RCU reader! */
preempt_enable();
```

CPU 1

```
/* Called after CPU 0's preempt_disable() */
synchronize_rcu();

/* Can exec before CPU 0 preempt_enable() */
some_func();
```

After consolidation: synchronize_rcu() has to wait

CPU 0

```
/* This is start of an RCU reader! */
preempt_disable();


/* This is end of an RCU reader! */
preempt_enable();
```

CPU 1

```
/* Called after CPU 0’s preempt_disable() */
synchronize_rcu();


/* Executes only much later! */
some_func();
```

rcuperf can prove it.

What does the rcuperf test do?

- Starts N readers and N writers on N CPUs
- Readers just do `rcu_read_lock() + rcu_read_unlock()` in a loop.
- Writers call and measure wall-clock time of synchronize_rcu() repeatedly.

What I did (HACK) : Modified test to busy loop for N ms on reserved CPU:

```
void reserved_thread() {
     preempt_disable();
     busy_loop_ms(N);
     preempt_enable();
}
```

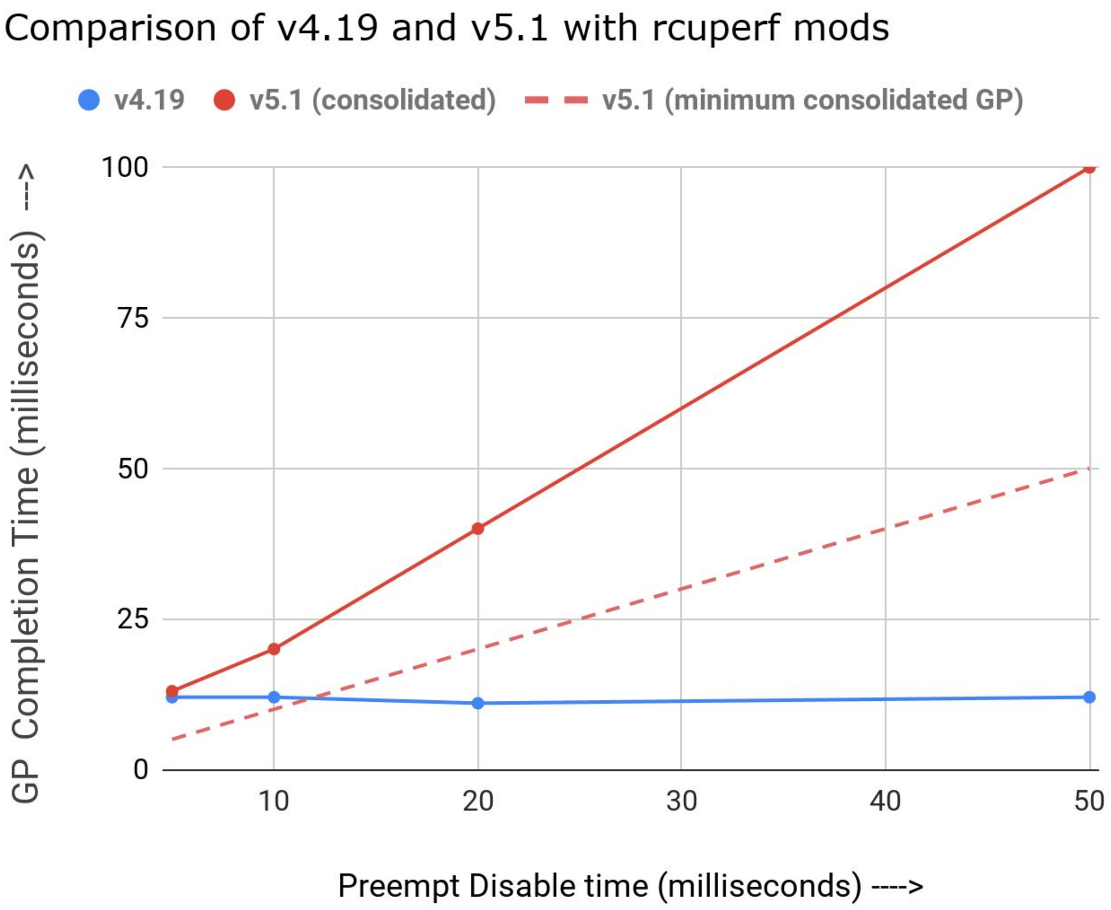
What could be the expected Results?

RCU Flavor Consolidation

Performance Changes

This is still **within RCU specification!**

Also note that disabling preemption for so long is most not acceptable by most people anyway.



RCU Flavor Consolidation

Notice that `synchronize_rcu` time was 2x the `preempt_disable` time, that's cos:

```
                 synchronize_rcu Wait      synchronize_rcu Wait
                |<--------------------->|  |---------------------|
                v                       v  v                     v
<----------> <----------> <----------> <----------> <---------->
     GP            GP           GP          GP           GP

GP = long preempt disable duration
```

Consolidated RCU - The different cases to handle

Say RCU requested special help from the reader section unlock that is holding up a GP for too long....

```
preempt_disable();
rcu_read_lock();
do_some_long_activity();  // TICK sets per-task ->need_qs bit
rcu_read_unlock();        // ... so need help from rcu_read_unlock();
preempt_enable();
```

Consolidated RCU - The different cases to handle

RCU-preempt reader nested in RCU-sched due to preempt_disable()

Before:

```
preempt_disable();
rcu_read_lock();
do_some_long_activity();
rcu_read_unlock();       // Report QS ASAP
preempt_enable();
```

Now:

```
preempt_disable();
rcu_read_lock();
do_some_long_activity();
rcu_read_unlock();
     -> rcu_read_unlock_special(); // Defer the QS and set
                                   // bit & set PREEMPT_NEED_RESCHED
preempt_enable(); // Report the QS
```

Consolidated RCU - The different cases to handle

RCU-preempt reader nested in RCU-sched due to local_irq_disable()

(This is a special case where previous reader requested deferred special processing by setting ->deferred_qs bit)

Before:

```
local_irq_disable();
rcu_read_lock();
rcu_read_unlock()
        -> rcu_read_unlock_special(); // Report the QS
local_irq_enable();
```

Now:

```
local_irq_disable();
rcu_read_lock();
rcu_read_unlock();
     -> rcu_read_unlock_special(); // Defer the QS and set
                                   // bit & set PREEMPT_NEED_RESCHED
local_irq_enable();  // CANNOT Report the QS, still deferred.
```

Consolidated RCU - The different cases to handle

RCU-preempt reader nested in RCU-sched due to IRQ entry :

(This is a special case where previous reader requested deferred special processing by setting ->deferred_qs bit)

Before:

```
/* hardirq entry */
rcu_read_lock();
rcu_read_unlock()
        -> rcu_read_unlock_special(); // Report the QS
/* hardirq exit */
```

Now:

```
/* hardirq entry */
rcu_read_lock();
do_some_long_activity();
rcu_read_unlock();
     -> rcu_read_unlock_special(); // Defer the QS and set
                                   // rcu_read_unlock_special.deferred_qs
                                   // bit & set TIF_NEED_RESCHED
/* hardirq exit */    // Report the QS
```

Consolidated RCU - The different cases to handle

RCU-preempt reader nested in RCU-bh

Before:

```
local_bh_disable(); /* or softirq entry */
rcu_read_lock();
do_some_long_activity();
rcu_read_unlock(); // Report QS ASAP
local_bh_enable(); /* or softirq exit */
```

Now:

```
local_bh_disable(); /* or softirq entry */
rcu_read_lock();
do_some_long_activity();
rcu_read_unlock();
     -> rcu_read_unlock_special(); // Defer the QS and set
                                   // bit & set PREEMPT_NEED_RESCHED
local_bh_enable(); /* or softirq exit */    // Report the QS
```

Consolidated RCU - The different cases to handle

RCU-bh reader nested in RCU-preempt or RCU-sched

Before:

```
preempt_disable();
/* Interrupt arrives */
/* Raises softirq */
/* Interrupt exits */
__do_softirq();
   -> rcu_bh_qs();       /* Reports a BH QS */
preempt_enable();
```

Now:

```
preempt_disable();
/* Interrupt arrives */
/* Raises softirq */
/* Interrupt exits */
__do_softirq();          /* Do nothing -- preemption still disabled */
preempt_enable();
```

Consolidated RCU - The different cases to handle

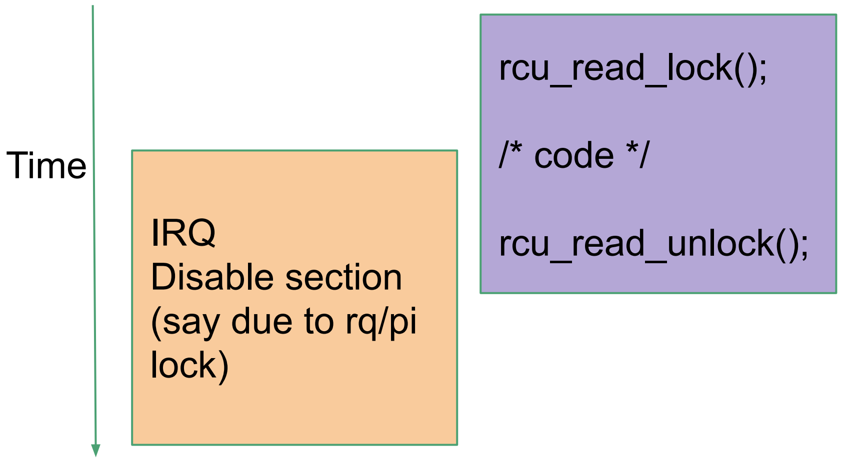
Solution: In case of denial of attack, ksoftirqd’s loop will report QS.
No reader sections expected there:

See commit: d28139c4e967 ("rcu: Apply RCU-bh QSes to RCU-sched and RCU-preempt when safe")

Consolidated RCU - Fixing scheduler deadlocks...

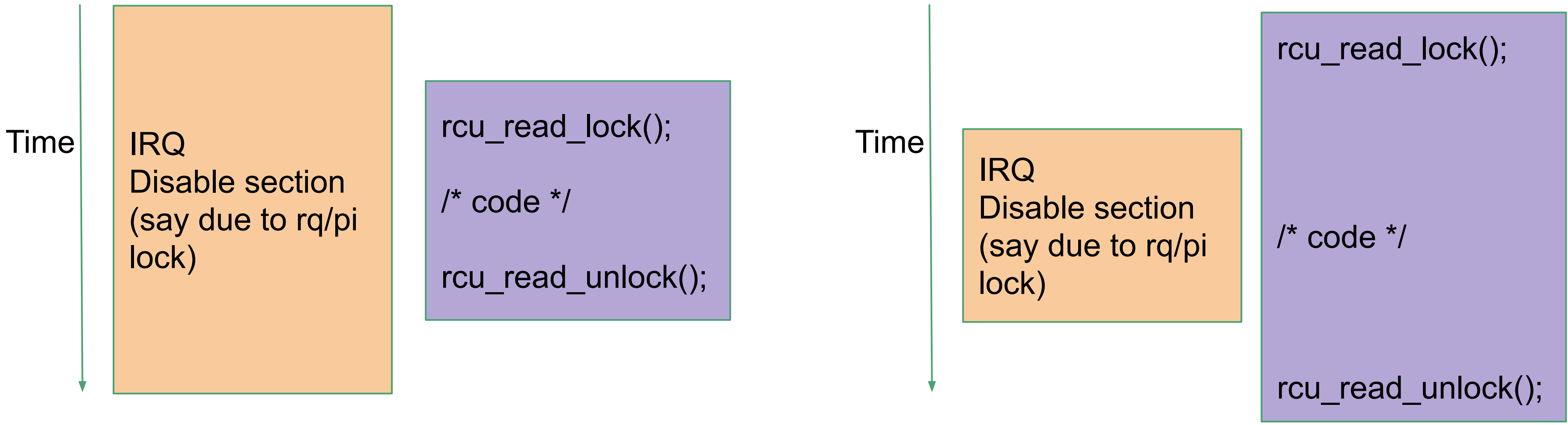
The forbidden scheduler rule... This is NOT allowed (https://lwn.net/Articles/453002/)

"Thou shall not hold RQ/PI locks across rcu_read_unlock() if thou not holding it or disabling IRQ across both rcu_read_lock() + rcu_read_unlock()"



Consolidated RCU - Fixing scheduler deadlocks...

The forbidden scheduler rule... This is ALLOWED:



Consolidated RCU - Fixing scheduler deadlocks...

But we have a new problem… Consider case: future rcu_read_unlock_special() might be called due to a previous one being deferred.

```
previous_reader()
{
	rcu_read_lock();
	do_something();      /* Preemption happened here (so need help from rcu_read_unlock_special. */
	local_irq_disable(); /* Cannot be the scheduler as we discussed! */
	do_something_else();
	rcu_read_unlock();  // As IRQs are off, defer QS report but set deferred_qs bit in rcu_read_unlock_special
	do_some_other_thing();
	local_irq_enable();
}

current_reader() /* QS from previous_reader() is still deferred. */
{
	local_irq_disable();  /* Might be the scheduler. */
	do_whatever();
	rcu_read_lock();
	do_whatever_else();
	rcu_read_unlock();  /* Must still defer reporting QS once again but safely! */
	do_whatever_comes_to_mind();
	local_irq_enable();
}
```

Consolidated RCU - Fixing scheduler deadlocks...

Fixed in commit: 23634eb (“rcu: Check for wakeup-safe conditions in rcu_read_unlock_special()”)

Solution: Intro rcu_read_unlock_special.b.deferred_qs bit. (Which is set in previous_reader() in previous example).

Raise softirq from _special() only when either of following are true:

- in_irq() (later changed to in_interrupt) - because ksoftirqd wake-up impossible.
- deferred_qs is set which happens in previous_reader() in previous example.

This makes the softirq raising not wake ksoftirqd thus avoiding a scheduler deadlock.

Made detailed notes on scheduler deadlocks:

https://people.kernel.org/joelfernandes/making-sense-of-scheduler-deadlocks-in-rcu

https://lwn.net/Articles/453002/

Future work

- More Torture testing on arm64 hardware
- Re-design dynticks counters to keep simple
- List RCU checking updates
- RCU scheduler deadlock checking
- Reducing grace periods due to kfree_rcu().
- Make possible to not embed rcu_head in object
- More RCU testing, experiment with modeling etc.
- More systematic study of __rcu sparse checking.

- For questions, please email the list: rcu@vger.kernel.org
- Follow us on Twitter:
  - @paulmckrcu
  - @joel_linux
  - @boqun_feng
  - @srostedt

Thank you!