RCU in 2019

(from my perspective related to work I have been doing)

Credits

RCU is the great decades-long work of Paul Mckenney and others. I am relatively new on the scene (~ 1.5 years).

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

- Worked on Linux for a decade or so.
- Heard only 5 people understand RCU. Opportunity !!!!
- Tired of reading RCU traces and logs that made no sense.
- Was running into RCU concepts but didn’t know their meaning.

Time to put all mysteries to and end...

- Turned out I actually love it and it is like a puzzle solving.

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

- Striving to be community asset who can help with RCU issues, concepts, improvements, reviewing.
- Started coming up with RCU questions (~1.5 years ago) about why things could not be done a certain way to which PaulMck quickly reminded me that they are "correct" but... RCU is complex since it has to "survive" in the real-world.

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

PaulMck says…

“Here is your nice elegant little algorithm”

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

PaulMck says…

“Here is your nice elegant little algorithm equipped to survive in the Linux kernel”

Agenda

- Introduction
- RCU Flavor consolidation
  - Performance
  - Scheduler Deadlock fixes
- List RCU API improvements (lockdep)

Extra material / slides:

  - kfree_rcu() batching
  - Dynticks and NOHZ_FULL fixes

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:

```
int reader() {

    struct shared_data sd =
        rcu_dereference(global_sd);
    rcu_read_unlock(); // READ
    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; // COPY
    sd->a = 2;
    rcu_assign_pointer(global_sd, sd); // UPDATE
    spin_unlock(&sd->lock);
    synchronize_rcu();
    kfree(old_sd);
}
```

Intro: Typical RCU workflow

- Writer makes a copy of data and modifies it.
- Readers execute in parallel on unmodified data.
- To know all prior readers are done, writer:
  - commit the modified copy (pointer)
  - Waits for a grace period.
  - After the wait, destroy old data.
  - Writer’s wait marks the **start of a grace period.**
- Quiescent state (QS) concept - a state that the CPU passes through which signifies that the CPU is no longer in a read side critical section (tick, context switch, idle loop, use code, etc).
  - All CPUs must report passage through a QS.
  - Once all CPUs are accounted for, the **grace period can end.**

Intro: What’s RCU good for - summary

- Fastest Read-mostly synchronization primitive:
  - Readers are **almost free** regardless of number of concurrent ones. [Caveats]
  - Writes are costly but per-update cost is **amortized**.
    - 1000s or millions of updates can share a grace period, so cost of grace period cycle (updating shared state etc) is divided.
  - Read and Writer sections execute in **parallel.**

[Caveats about ‘literally free”]

- Slide 12: "Readers are almost free": Yes, but caveats do apply:

  (1) CONFIG_PREEMPT=n is free but =y has a small cost (especially unlock()). (2) Possibility of rcu_dereference() affecting compiler optimizations. (3) Given that rcu_read_lock() and rcu_read_unlock() now imply barrier(), they can also affect compiler optimizations.

Intro: What is RCU and what its RCU good for ?

- More use cases:
    - Wait for completion primitive (used in deadlock free NMI registration, BPF maps)
    - Reference counter replacements
    - Per-cpu reference counting
    - Optimized reader-writer locking (percpu_rwsem)
        - Slow / Fast path switch (rcu-sync)
    - Go through RCUUsage paper for many many usage patterns.

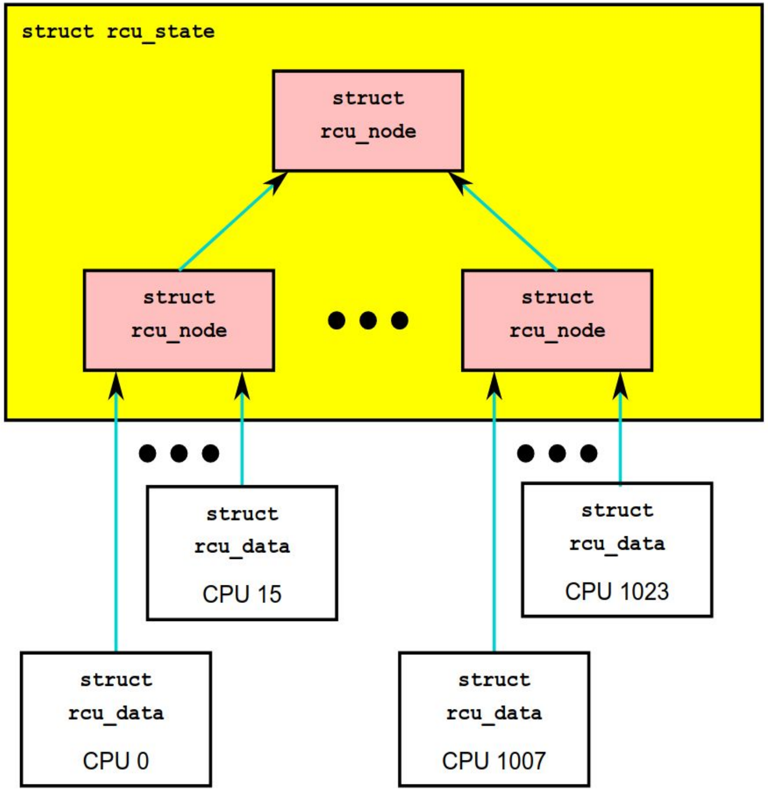
Only talking about TREE_RCU today

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

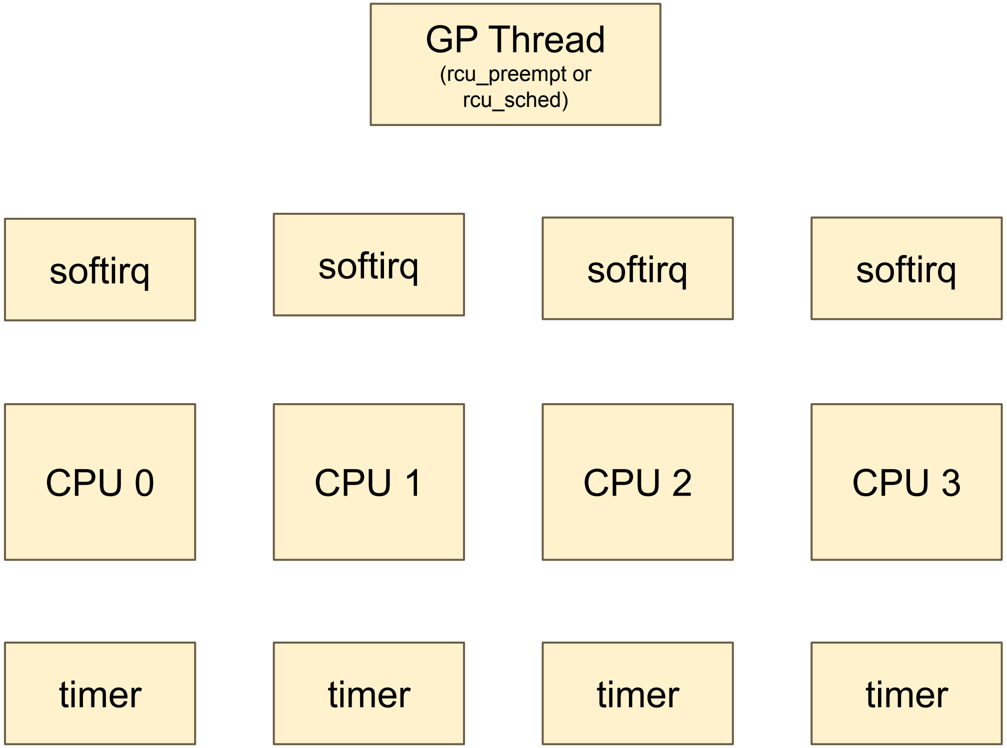
If you are claiming that I am worrying unnecessarily, you are probably right. But if I didn't worry unnecessarily, RCU wouldn't work at all!
— Paul McKenney

There's also TINY RCU but not discussing it.

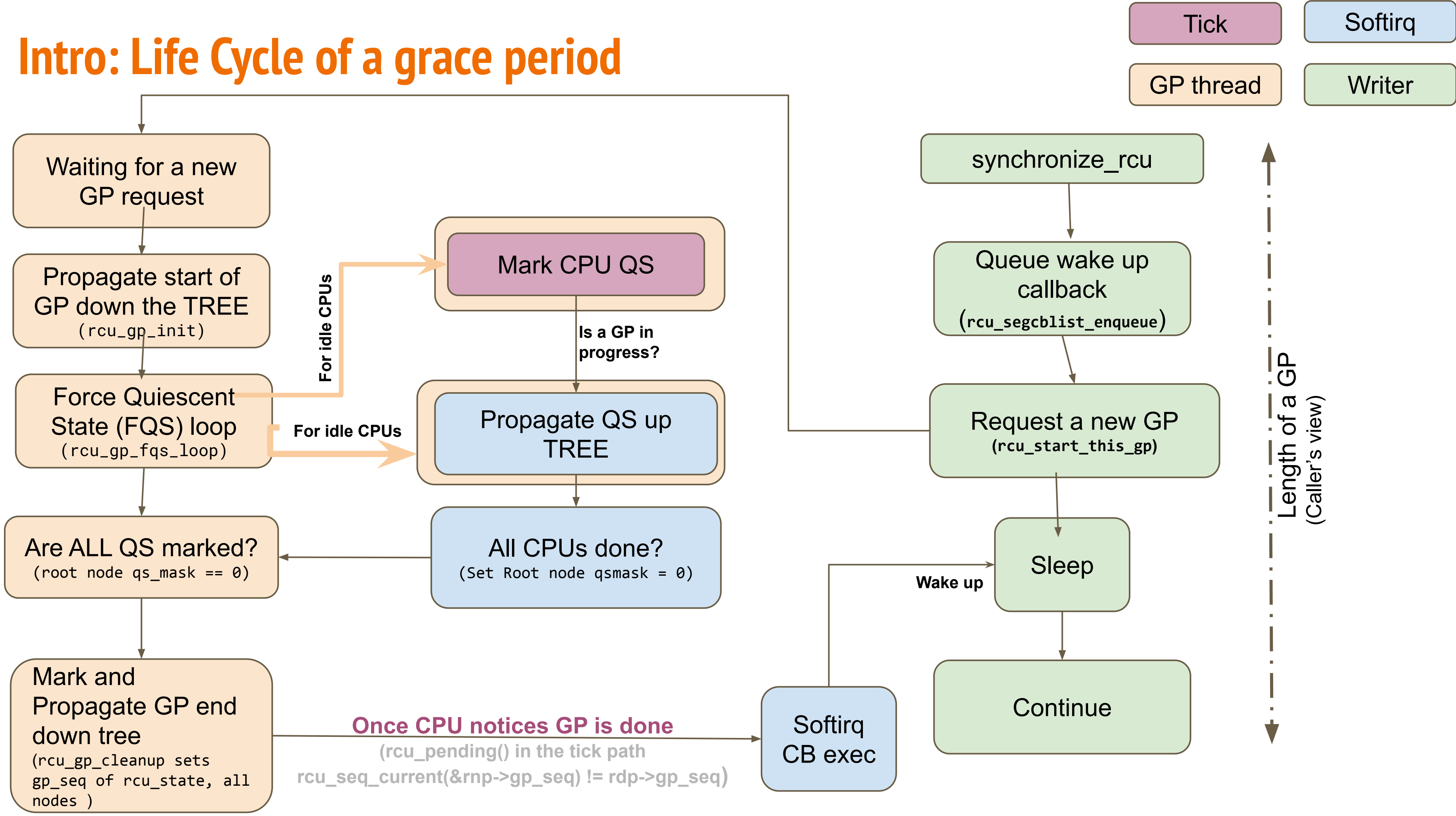
Intro: How TREE_RCU works?



Intro: Components of TREE RCU (normal grace period)



Intro: Life Cycle of a grace period



Intro: Where/When is Quiescent state reported

- Transition to / from CPU idle and user mode   (a.k.a **dyntick-idle transitions**)
- **Context switch path** (If read-lock nesting is 0) -- NOTE: reports a CPU QS but task still blocks GP.
- In the Scheduler **tick**:
  - Task is not running in IRQ/preempt disable (no more sched/bh readers)
  - Read-lock nesting is 0   (no more preemptible rcu readers)
- In the FQS loop
  - Forcing quiescent states for **idle (nohz_idle), usermode (nohz_full), and offline** CPUs.
  - Forcing quiescent states for **CPUs that do dyntick-idle transitions**.
  - Including dyntick idle transitions faked with rcu_momentary_dyntick_idle() such as by cond_resched() on PREEMPT=n kernels.
  - Give soft hints to CPU's scheduler tick to resched task (rdp.rcu_urgent_qs)

Intro: Which paths do QS reports happen from?

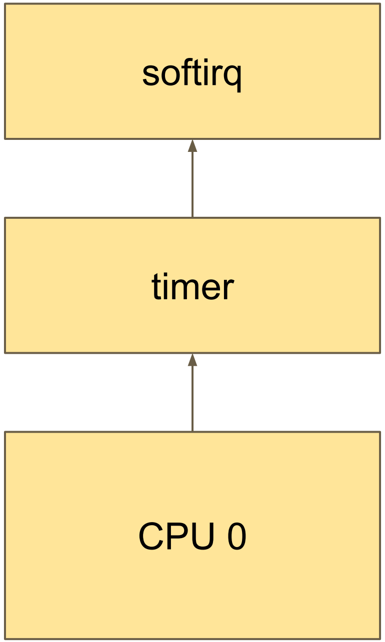
- CPU local update is cheap:
  - Tick path.
  - Context Switch.
  - Help provided from rcu_read_unlock() if needed.
  - Reporting of a deferred QS reporting (when rcu_read_unlock() could not help).

- Global update of CPU QS -- Expensive, but REQUIRED for GP to end:
  - Softirq (once it sees CPU local QS from tick)
  - CPU going online (rcu_cpu_starting)
  - Grace period thread’s FQS loop:
    - dyntick-idle QS report (Expensive, involves using atomic instructions)
      - dyntick-idle transitions (Kernel - > user ; Kernel -> idle)
      - cond_resched() in PREEMPT=n kernels does a forced dyntick-idle transition (HACK).

Intro: What happens in softirq ?

Per-CPU Work:

- **QS reporting for CPU and propagate up tree.**
- **Invoke any callbacks whose GP has completed.**



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.

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())

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

At around 200ms:

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()

Report QS

RCU Read-side critical section

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

At around 300ms for NOHZ_FULL CPUs:

GP THREAD

Send IPIs to CPUs that are still holding up

RCU read-side critical section

IPI handler Turns on TICK

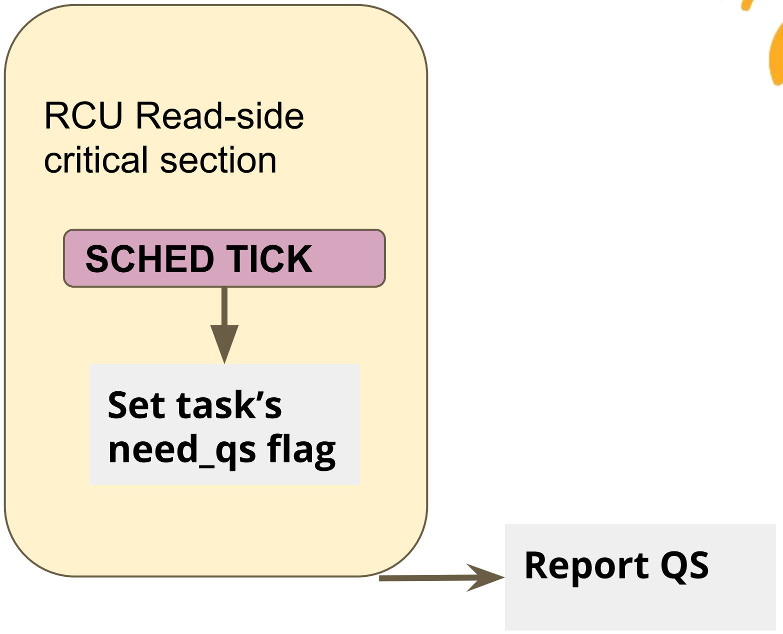
SCHED TICK

SCHED TICK

Report QS

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

At around 1 second of start of GP:



RCU Flavor Consolidation: Why?

In 4.19 and before, there were the 3 RCU flavors (many more but those are special purpose)

- Preemptible RCU
  - In `CONFIG_PREEMPT` kernels
  - `task->read_lock_nesting` keeps track of reader

- Sched RCU
  - Preempt disable sections
  - IRQ disabled sections

- BH RCU
  - Softirq disable sections

Problem: Too many APIs for synchronization. Confusion over which one to use!

- `call_rcu` and `synchronize_rcu` (for preempt)
- `call_rcu_sched` and `synchronize_rcu_sched` (for sched)
- `call_rcu_bh` and `call_rcu_bh` (for bh)

Now: One single `call_rcu` and `synchronize_rcu` to rule them all.

RCU Flavor Consolidation: Changes

3 rcu_state structures before (v4.19)

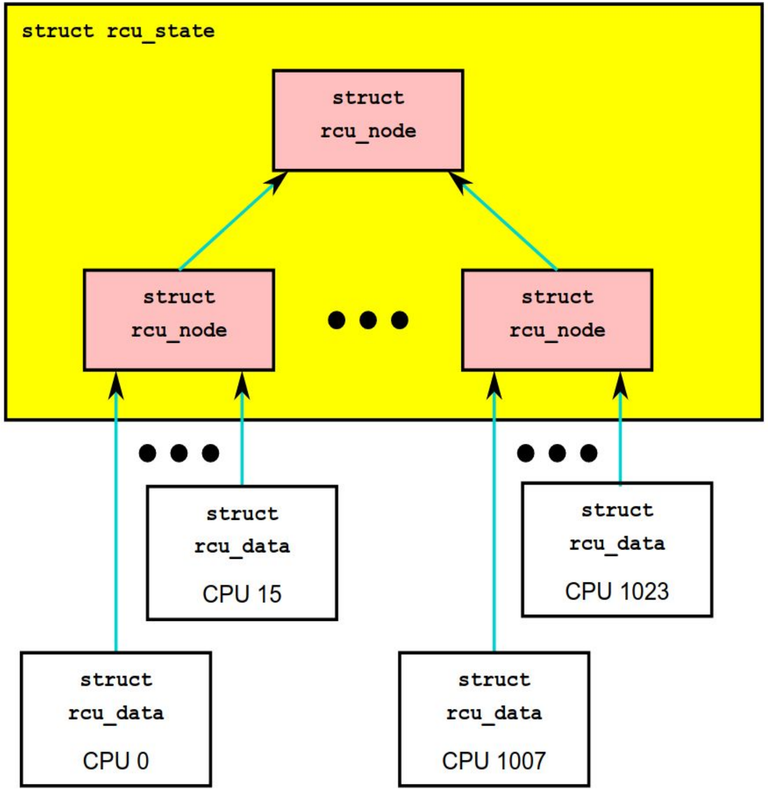
- rcu_preempt
- rcu_sched
- rcu_bh

Now just 1 rcu_state structure to handle all flavors:

- rcu_preempt (Also handles sched and bh)

Number of GP threads also reduced from 3 to 1 since state machines have been merged now.

- Less resources!
- Less code!



RCU Flavor Consolidation: Performance Changes

rcuperf test suite

- Starts N readers and N writers on N CPUs
- Each reader and writer affined to a CPU
- Readers just do `rcu_read_lock/rcu_read_unlock` in a loop
- Writers start and end grace periods by calling synchronize_rcu repeatedly.
- Writers measure time before/after calling synchronize_rcu.

Locally modified test to do on reserved CPU continuously:

`preempt_disable + busy_wait + preempt_enable` in a loop

Note: This is an additional reader-section variant now (sched-RCU) running in parallel to the existing rcu_read_lock readers (preemptible-RCU).

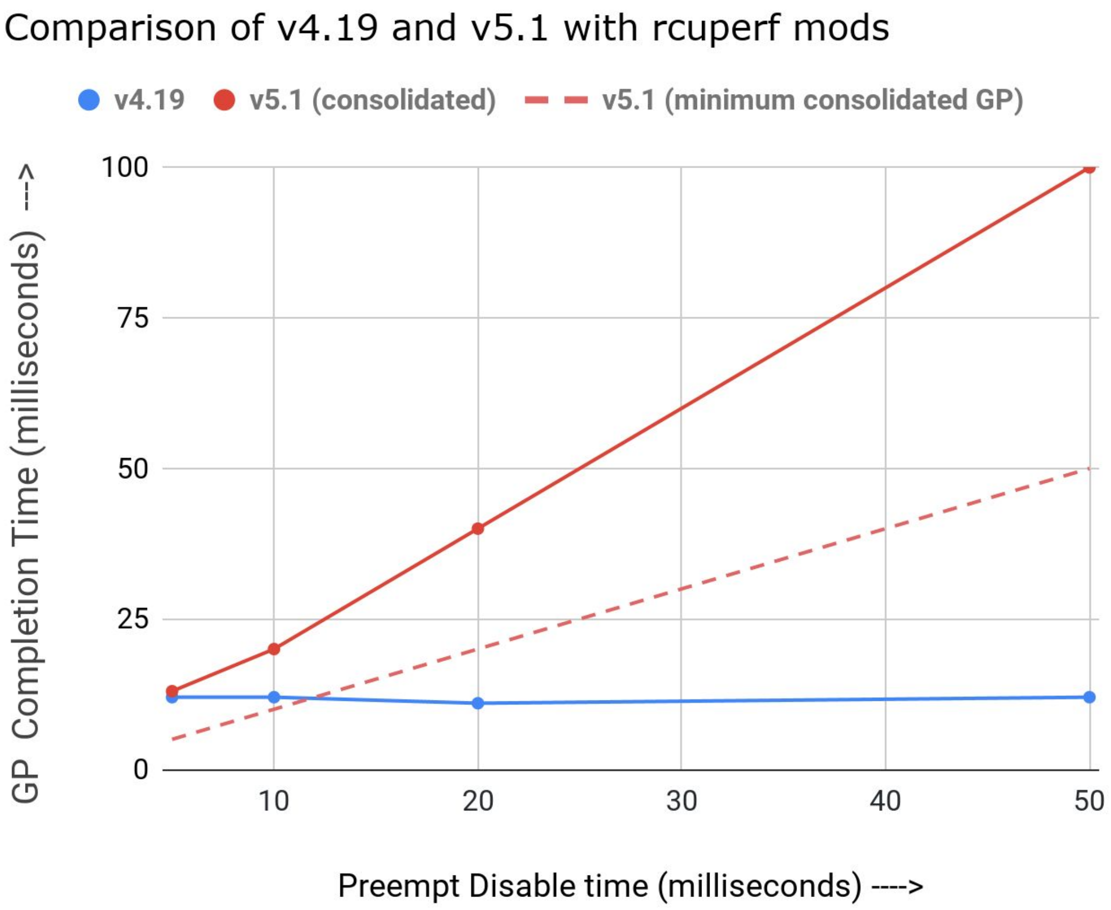
What could be the expected Results?

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();        // Now need special processing from
                          // rcu_read_unlock_special()
preempt_enable();
```

Consolidated RCU - The different cases to handle

RCU-preempt reader nested in RCU-sched

Before:

```
preempt_disable();
rcu_read_lock();
do_some_long_activity();
rcu_read_unlock()
        -> rcu_read_unlock_special(); // Report the QS
preempt_enable();
```

Now:

```
preempt_disable();
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
preempt_enable(); // 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()
        -> rcu_read_unlock_special(); // Report the QS
local_bh_enable(); /* 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
                                   // rcu_read_unlock_special.deferred_qs
                                   // bit & set TIF_NEED_RESCHED
local_bh_enable(); /* softirq exit */    // Report the QS
```

Consolidated RCU - The different cases to handle

RCU-preempt reader nested in local_irq_disable (IRQoff)

(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
                                   // rcu_read_unlock_special.deferred_qs
                                   // bit & set TIF_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 (IRQoff)

(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-bh reader nested in RCU-preempt reader

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-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

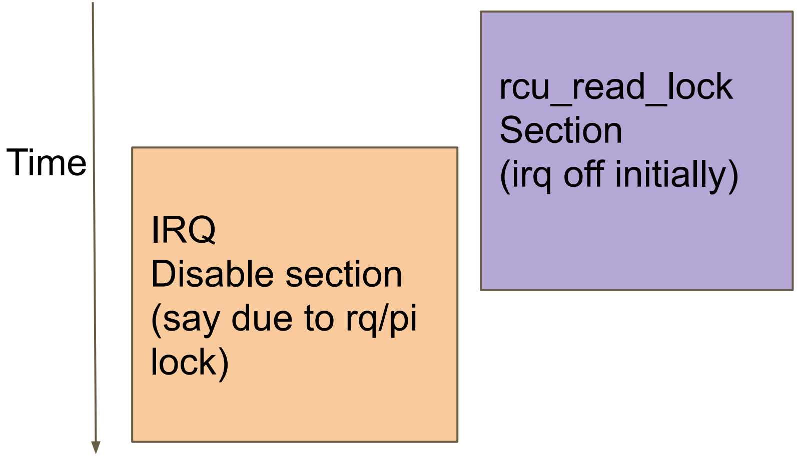
Solution: In case of denial of attack, ksoftirqd’s loop with report QS.
No reader sections expected there:

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

Consolidated RCU - Fixing scheduler deadlocks...

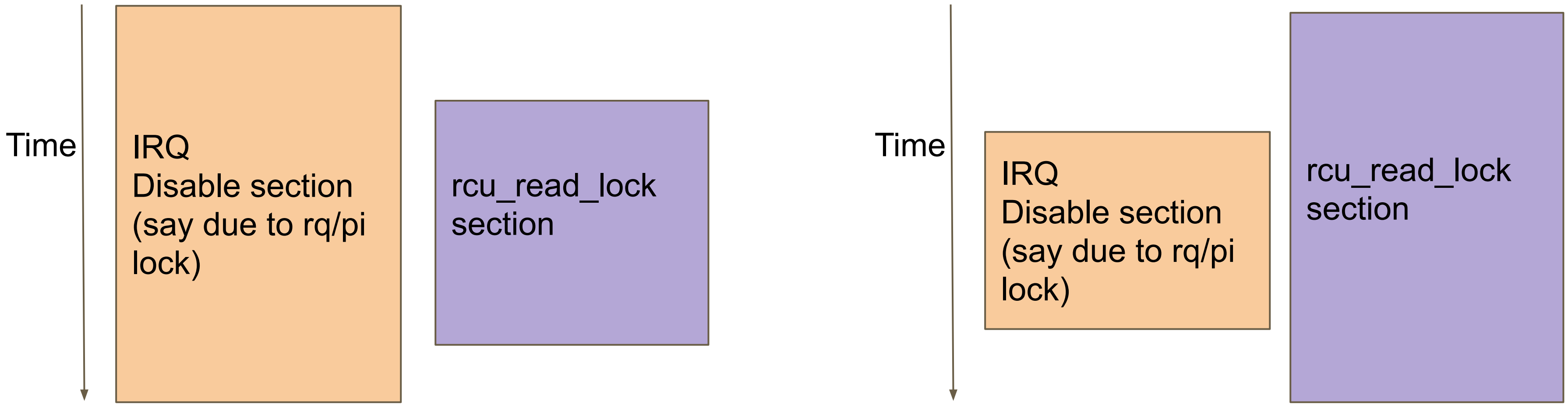
The forbidden scheduler rule... This is NOT allowed (https://lwn.net/Articles/453002/)

“Thou shall not hold RQ/PI locks across an rcu_read_unlock() if thou not holding it or disabling IRQ across both both the 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/

List RCU usage - hardening (TODO: Change to be from user POV)

- RCU “consumer” APIs have built in checking.
    - `rcu_read_lock()`
    - `rcu_read_lock_sched()`
    - `rcu_read_lock_bh()`
- However listRCU APIs don’t have anything like that…
- list_for_each_entry_rcu() does no checking at all …
- Can hide subtle RCU related bugs and list corruption

List RCU usage - hardening

- Why traditionally, been no checking?
- Because it is hard… Can’t know which RCU flavor the calling code intended:

```
rcu_read_lock();
list_for_each_entry_rcu() { .. }
rcu_read_unlock();
```

Vs

```
rcu_read_lock_sched();
list_for_each_entry_rcu() { .. }
rcu_read_unlock_sched();
```

List RCU usage - hardening

- Option 1: Introduce new APIs:
  - `list_for_each_entry_rcu`
  - `list_for_each_entry_rcu_bh`
  - `list_for_each_entry_rcu_sched`

- Drawback:
  - Proliferation of APIs...

List RCU usage - hardening

- Option 2: Can do better since RCU backend is now consolidated.
    - So checking whether ANY flavor of RCU reader is held, is sufficient in list_for_each_entry_rcu()
        - sched
        - bh
        - preemptible RCU

List RCU usage - hardening

Final solution:

```
int rcu_read_lock_any_held(void)
{
        bool ret;

        if (rcu_read_lock_held_common(&ret))
                return ret;
        if (lock_is_held(&rcu_lock_map) ||
            lock_is_held(&rcu_bh_lock_map) ||
            lock_is_held(&rcu_sched_lock_map))
                return 1;
        return !preemptible();
}
EXPORT_SYMBOL_GPL(rcu_read_lock_any_held);
```

Called from…

```
list_for_each_entry_rcu()
```

List RCU - hardening

Patches:

https://lore.kernel.org/lkml/20190716184656.GK14271@linux.ibm.com/T/#t

https://lore.kernel.org/patchwork/project/lkml/list/?series=401865

Next steps:

- Make CONFIG_PROVE_RCU_LIST the default and get some more testing

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.

Thank you...

- Due to time constraint, rest of the slides are not covered, however feel free to come up with any questions:
  - Please email the list: rcu@vger.kernel.org
  - Follow us on Twitter: paulmckrcu@ joel_linux@ boqun_feng@

kfree_rcu() improvements

Problem: Too many kfree_rcu() calls can overwhelm RCU

- Concerns around vhost driver calling kfree_rcu often:
  - https://lore.kernel.org/lkml/20190721131725.GR14271@linux.ibm.com/
  - https://lore.kernel.org/lkml/20190723035725-mutt-send-email-mst@kernel.org/
- Similar bug was fixed by avoiding kfree_rcu() altogether in access()
  - https://lore.kernel.org/lkml/20190729190816.523626281@linuxfoundation.org/

Can we improve RCU to do better and avoid future such issues?

Current kfree_rcu() design (TODO: Add state diagram)

kfree_rcu -> queue call back -> wait for GP -> run kfree from RCU core

- Flooding kfree_rcu() can cause lot of RCU work and grace periods.
- All kfree() happens from softirq as RCU callbacks can execute from softirq
    - Unless nocb or nohz_full is passed of course, still running more callbacks means other RCU work also has to wait.

kfree_rcu() re-design - first pass

kfree_rcu batching:

- 2 per-CPU list (Call these A and B)
- Each kfree_rcu() request queued on a per-CPU list (no RCU)
- Within KFREE_MAX_JIFFIES (using schedule_delayed_work())
    - Dequeue all requests on A, and Queue on B
    - Call queue_rcu_work() to run a worker to free all B-objects after GP
    - Meanwhile new requests can be queued on A

kfree_rcu() re-design - first pass

Advantages:

- 5X reduction in number of grace periods in rcuperf testing:
  - GPs from **~9000 to ~1700** (per-cpu kfree_rcu flood on 16 CPU system)
- All kfree() happens in workqueue context and systems schedules work on the different CPUs.

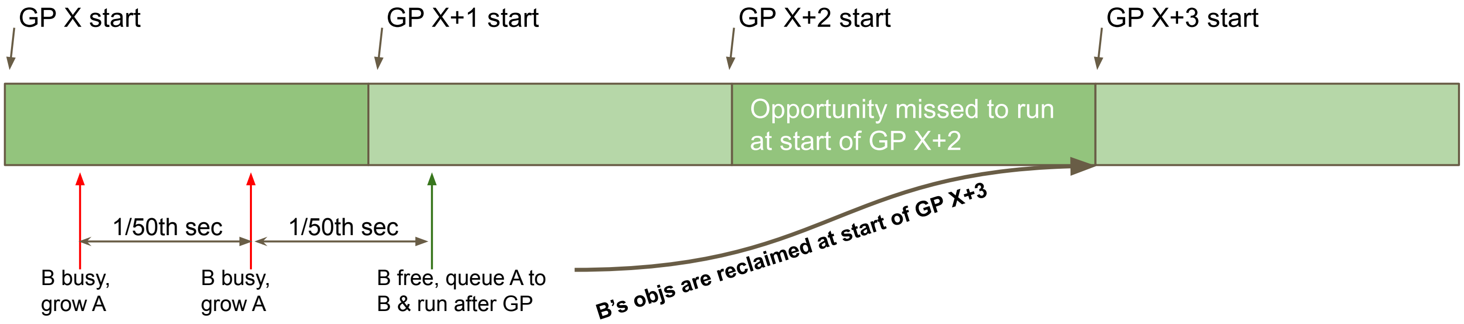
Drawbacks:

- Since list B cannot be queued into until GP ends, list A keeps growing
- Causes high memory consumption (300-350) MB.

kfree_rcu() re-design - first pass

Drawbacks:

- Since list B cannot be queued into until GP, list A keeps growing
- Miss opportunities and has to wait for future GPs on busy system
- Causes higher memory consumption (300-350) MB.

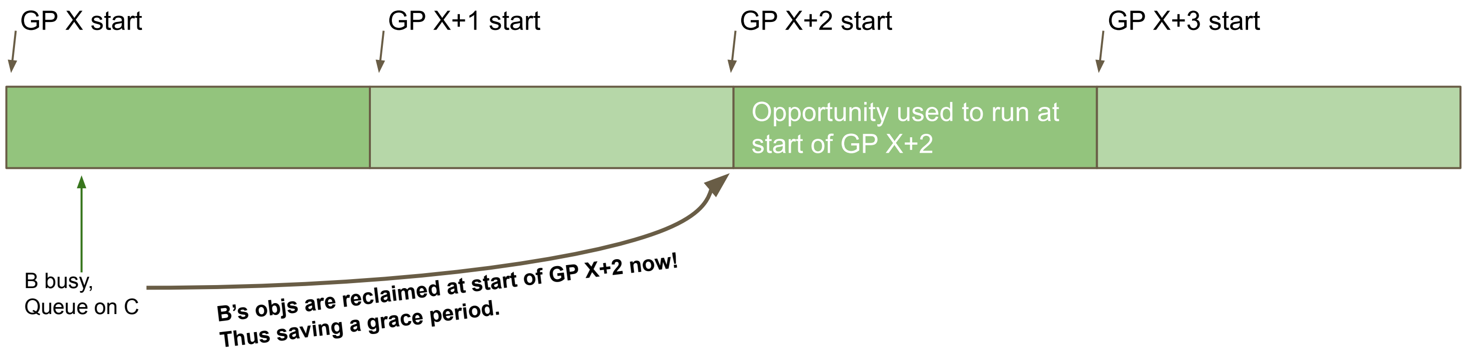


B is busy, so any attempts to Queue A to B have to be delayed.

So we keep retrying every 1/50th of a second; and A keeps growing…

kfree_rcu() re-design - second pass

- Introduce a new list C.
    - If B is busy just queue on C and schedule that after GP



- Brings down memory consumption by 100-150MB during kfree_rcu flood
- Maximize bang for GP-bucks!

dyntick RCU counters : intro (TODO: Condense these 9 slides into one slide)

- RCU needs to know what state CPUs are in
- If we have scheduler tick, no problem! Tick path tells us.

But…

- NOHZ_FULL can turn off tick in both user mode and idle mode.
- NOHZ_IDLE can turn off tick in idle mode

Implemented with a per-CPU atomic counter:

- rcu_data :: atomic_t dynticks : Even for RCU-idle, odd non-RCU-idle

dyntick RCU counters : intro (TODO: Add state-diagrams)

- RCU transitions non-idle <-> idle:
  - kernel <-> user
  - kernel <-> idle
  - irq <-> user
  - irq <-> idle

Each of these transitions cause rcu_data::dynticks to be incremented.

- RCU remains non-idle:
  - irq <-> nmi
  - nmi <-> irq
  - kernel <-> irq
  - Irq <-> kernel
- **Each of these transitions cause rcu_data::dynticks to NOT be incremented.**

nohz CPUs tick turn off : intro

Scheduler tick is turn off is attempted:

- From cpu idle loop: `tick_nohz_idle_stop_tick();`
- At the end of IRQs:

```
irq_exit()
    -> tick_nohz_irq_exit()
            -> tick_nohz_full_update_tick()
```

tick_nohz_full_update_tick() checks `tick_dep_mask`. If any bits are set, tick is not turned off.

nohz CPUs : reasons to keep tick ON

Introduced in commit
d027d45d8a17 ("nohz: New tick dependency mask")

- POSIX CPU timers
- Perf events
- Scheduler
  - (for nohz_full, tick may be turned off if RQ has only 1 task)

The problem: nohz_full and RCU (TODO: Diagram)

Consider the scenario:

- A nohz_full CPU with tick turned off is looping in kernel mode.
- The GP kthread on another CPU notices CPU is taking too long.
- Sets per-cpu hints: rcu_urgent_qs to true and sends resched_cpu().
- IPI runs on nohz_full CPU.
- It enters the scheduler, still nothing happens. Why??
- Grace period continues to stall… resulting in OOM.

Reason:

- Entering the scheduler only means the per-CPU quiescent state is updated
- Does not mean the global view of the CPU’s quiescent state is updated…

The problem: nohz_full and RCU

What does updating global view of a CPU QS mean?
Either of the following:

- A. Quiescent state propagates up the RCU TREE to the root
- B. The rcu_data :: dynticks counter of the CPU is updated.
    - This causes the FQS loop to do A on dyntick transition of CPU.

Either of these operations will result in a globally updated view of CPU QS

Both of these are expensive and need to be done carefully.

The problem: nohz_full and RCU

The global view of a CPU QS is updated only from a few places..

- From the RCU softirq (which is raised from scheduler tick)
    - This does A (tick).

- Kernel/IRQ -> Idle / User transition
    - This does B (dyntick transition).

- cond_resched() in PREEMPT=n kernels
    - This does B (dyntick transition).

- Other special cases where a GP must not be stalled and an RCU reader is not possible: (rcutorture, multi_cpu_stop() etc.)

The problem: nohz_full and RCU

Possible Solution:

Use option B (forced dynticks transition) in the IPI handler.

- This would be rather hackish. Currently we do this only in very special cases.
- This will only solve the issue for the current GP, future GPs may still suffer.

The problem: nohz_full and RCU

Final Solution: Turn back the tick on.

- Add a new “RCU” tick dependency mask.
- RCU FQS loop sets rcu_urgent_qs to true and does resched_cpu() on the stalling CPU as explained.
- IPI handler arrives at CPU.
- On IPI entry, we set tick dependency mask.
- On IPI exit, the sched-tick code turns the tick back on.

This solution prevents the OOM splats.
https://git.kernel.org/pub/scm/linux/kernel/git/paulmck/linux-rcu.git/commit/?h=dev&id=5b451e66c8f78f301c0c6eaacb0246eb13e09974
https://git.kernel.org/pub/scm/linux/kernel/git/paulmck/linux-rcu.git/commit/?h=dev&id=fd6d2b116411753fcaf929e6523d2ada227aa553

dyntick counter cleanup attempts

Why?

In the previous solution, found a bug where dyntick_nmi_nesting was being set to (DYNTICK_IRQ_NONIDLE | 2)

However, the code was checking (dyntick_nmi_nesting == 2)

Link to buggy commit:
https://git.kernel.org/pub/scm/linux/kernel/git/paulmck/linux-rcu.git/commit/?h=dev&id=13c4b07593977d9288e5d0c21c89d9ba27e2ea1f

However, Paul wants the existing mechanism redesigned rather than cleaned up since it's too complex: http://lore.kernel.org/r/20190828202330.GS26530@linux.ibm.com