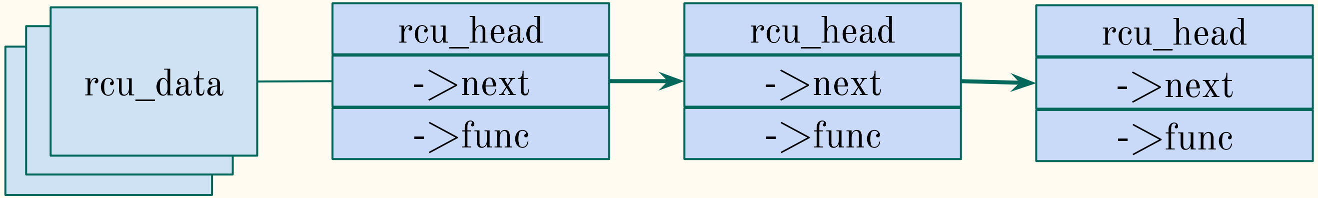
Reduce synchronize_rcu() latency

Paul McKenney (Meta)
Uladzislau Rezki (Sony)
Neeraj Upadhyay (AMD)
LPC 2024, Vienna, Austria

What is RCU(Read-Copy update)

Think of RCU as something that defers work, with one work item per callback

- each callback has a function pointer and an argument;
- callbacks are queued on per-CPU lists, invoked after grace period;
- allow fast and scalable read-side access to shared data.



synchronize_rcu() overview

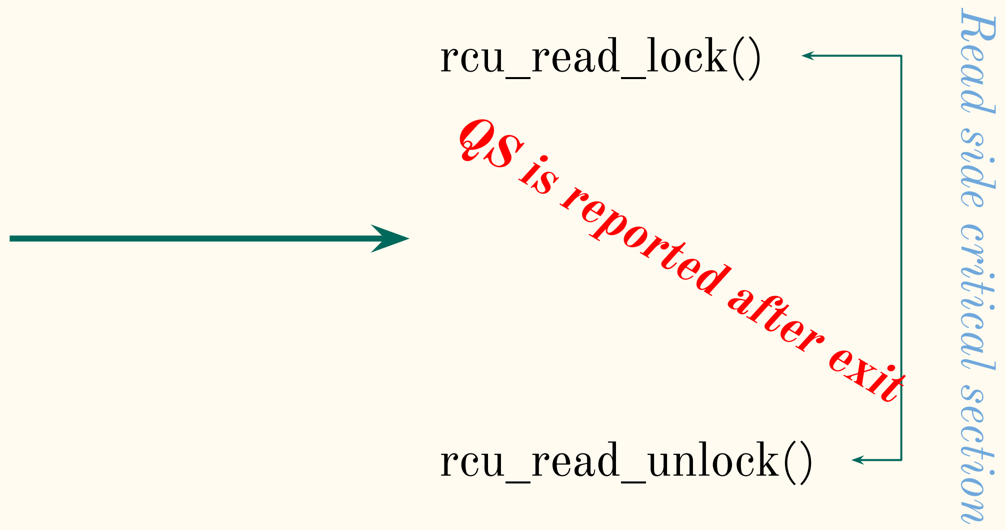
- synchronize_rcu() initialize and wait until a grace period has elapsed;
- A calling context is blocked;
- Depending on a workload it can take milliseconds;
- There are ~500 hundreds direct calls within the kernel(6.9.0-rc2);
- The latency of synchronize_rcu() depends strongly on kernel configuration:

  For example how RCU is configured in your kernel:

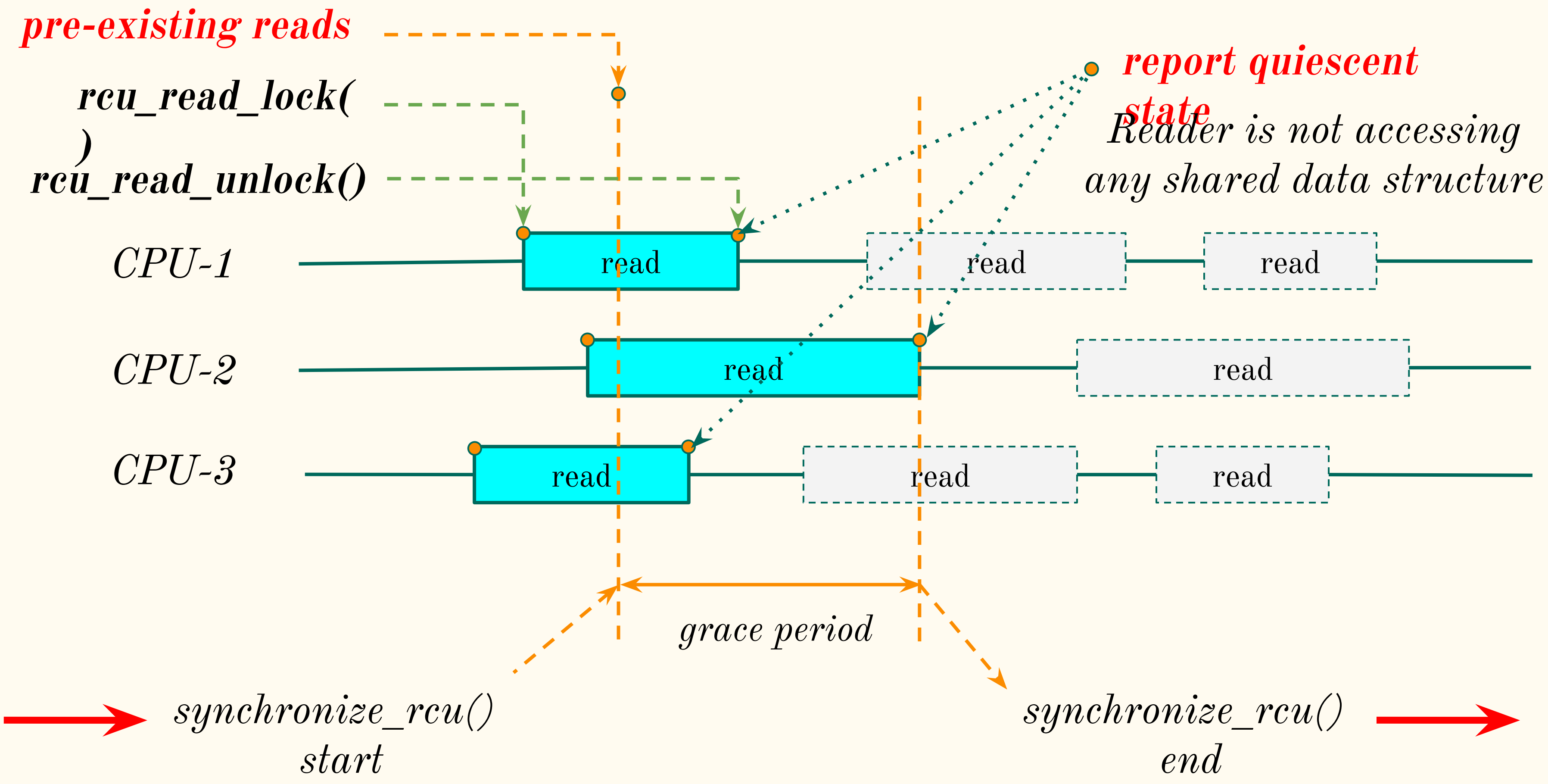
  - enabled/disabled **CONFIG_RCU_NOCB_CPU**;
  - enabled/disabled **CONFIG_RCU_LAZY**;
  - a boot parameter(**rcupdate.rcu_expedited**) that converts a normal synchronize_rcu() into an expedited version. A drawback of such approach is a need to send out IPIs what can affect a low-latency workloads and RT tasks.

synchronize_rcu() overview cont.

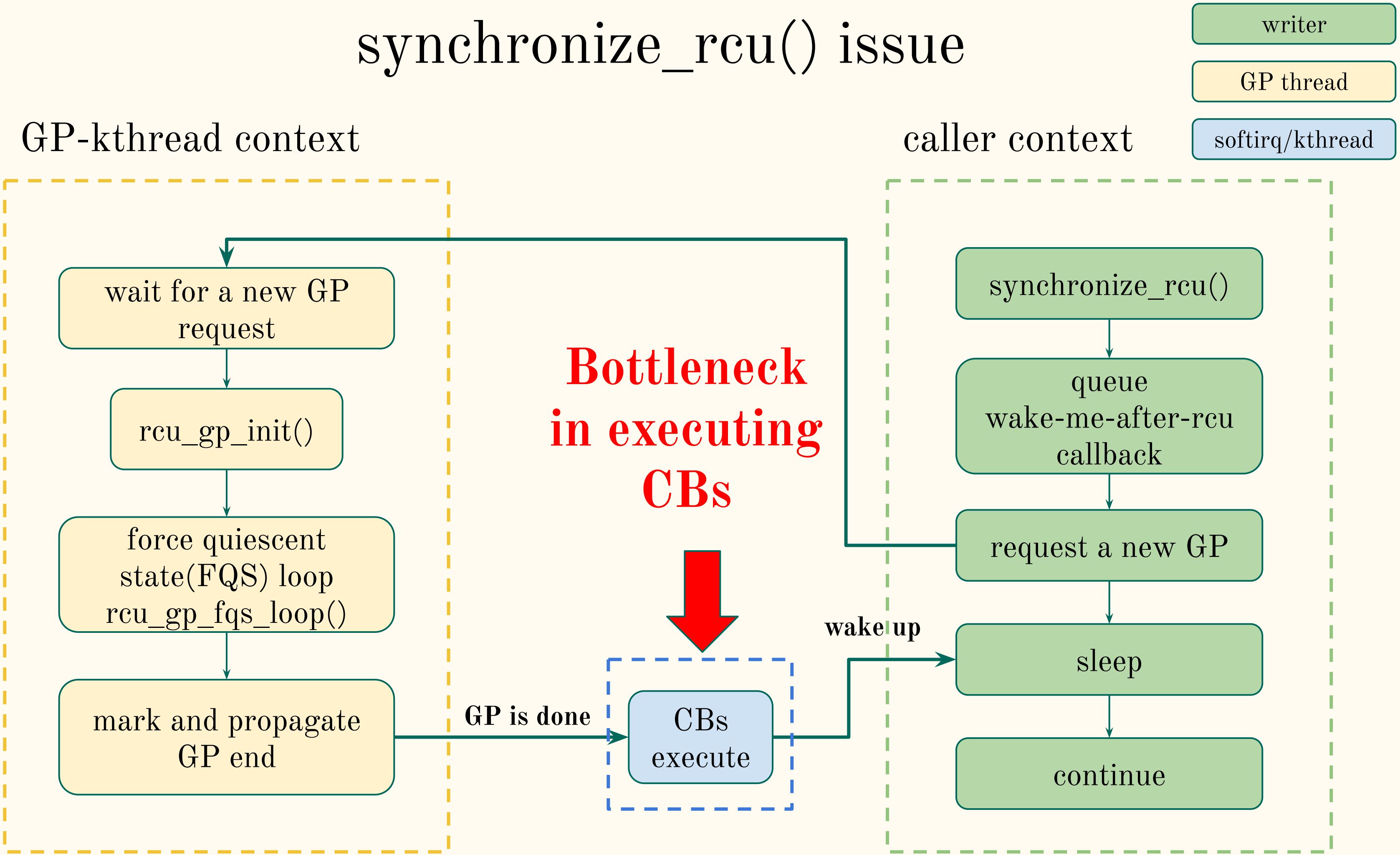
- A quiescent state(**QS**) concept - a state that CPU pases through, which means a CPU is no longer in a read side critical section:
  - tick;
  - context switch;
  - idle loop;
  - user code;
  - etc.



RCU GP basics

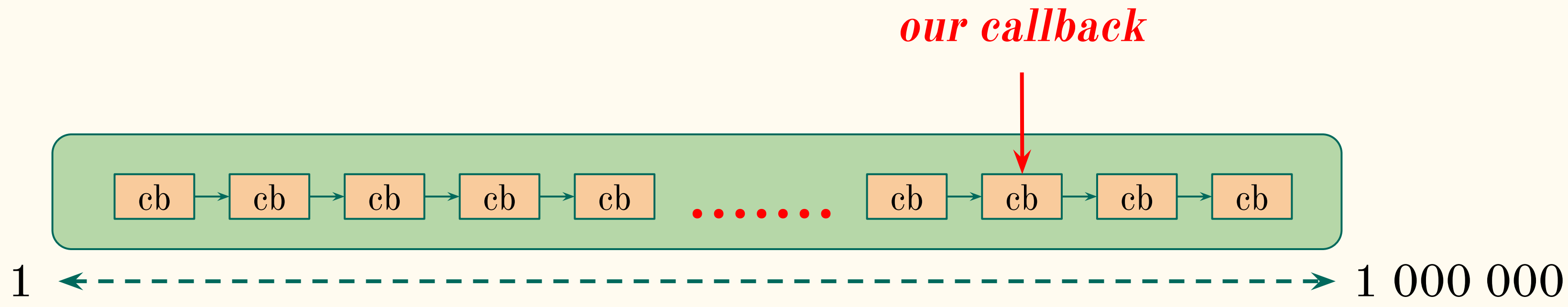


synchronize_rcu() issue



synchronize_rcu() issue(cont.)

- An executing time of callbacks depends on:
  - length of CB-list;
  - how fast previous callbacks were completed;
  - number of times callback invocation is paused;
  - where in a list our wake-me-after-rcu callback is located;



synchronize_rcu() issue(cont.)

- On our mobile devices i can easily trigger the scenario when a callback is last in the list out of ~3600 callbacks:

```
<snip>
 <...>-29     [001] d..1. 21950.145313: rcu_batch_start: rcu_preempt CBs=3613 bl=28
...
 <...>-29     [001] ..... 21950.152578: rcu_invoke_callback: rcu_preempt rhp=00000000b2d6dee8 func=__free_vm_area_struct.cfi_jt
 <...>-29     [001] ..... 21950.152579: rcu_invoke_callback: rcu_preempt rhp=00000000a446f607 func=__free_vm_area_struct.cfi_jt
 <...>-29     [001] ..... 21950.152580: rcu_invoke_callback: rcu_preempt rhp=00000000a5cab03b func=__free_vm_area_struct.cfi_jt
 <...>-29     [001] ..... 21950.152581: rcu_invoke_callback: rcu_preempt rhp=0000000013b7e5ee func=__free_vm_area_struct.cfi_jt
 <...>-29     [001] ..... 21950.152582: rcu_invoke_callback: rcu_preempt rhp=000000000a8ca6f9 func=__free_vm_area_struct.cfi_jt
 <...>-29     [001] ..... 21950.152583: rcu_invoke_callback: rcu_preempt rhp=000000008f162ca8 func=wakeme_after_rcu.cfi_jt
 <...>-29     [001] d..1. 21950.152625: rcu_batch_end: rcu_preempt CBs-invoked=3612 idle=....
<snip>
```

Summary

- The synchronize_rcu()function's implementation depends on kernel configuration
- The behaviour depends on how your kernel is configured
- Per-cpu lists can be too long (almost 1 000 000 CBs)
    - run "rm -rf" on folder with small files on fast SSD storage + linux kernel compiling

    rcuop/1-30 [008] D..1. **13483.560898**: rcu_batch_start: rcu_preempt **CBs=871001** bl=4200
    rcuop/1-30 [008] D..1. **13483.820768**: rcu_batch_end: rcu_preempt **CBs-invoked=537691** idle=...R
    ...
- Execution path has limitations:
    - time(how long we execute callbacks)
    - reschedule points(to prevent hogging CPU)
    - batch threshold(how many CBs already executed)
    - where in a list "wakeme-after-rcu" callback is located

New approach of normal synchronize_rcu() call

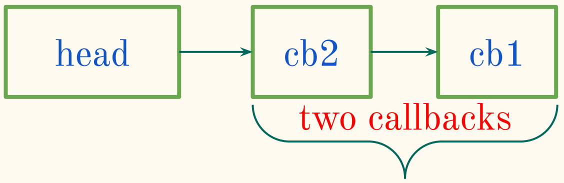
- Decouple a “sync” callback from others
- Bypass common per-cpu cb-lists
- Maintain a separate track of “sync” callers only
- Do limited direct wake-ups from GP-kthread
- The rest is deferred to a dedicated worker to perform a final flush
- Unify the call. So, the behaviour does not depend on kernel configuration
- A new approach can be enabled/disabled in runtime

New approach of normal synchronize_rcu() call(cont.)

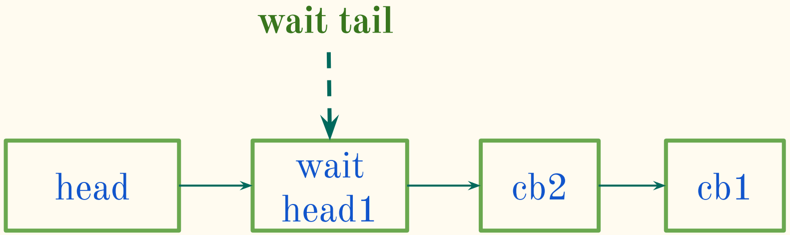
- There is a single lockless list
- It is used for handing synchronize_rcu() users
- rcu_synchronize nodes are enqueued to the llist
- At every GP init, a new wait-node is added:
  - it allows adding users and processing at the same time
- Within the llist, there are two tail pointers
  - **wait tail** - tracks the set of nodes, which need to wait for the current GP to complete
  - **done tail** - tracks the set of nodes, for which a GP has elapsed. These nodes processing will be done as part of cleanup work executed by a kworker

A state machine and cases

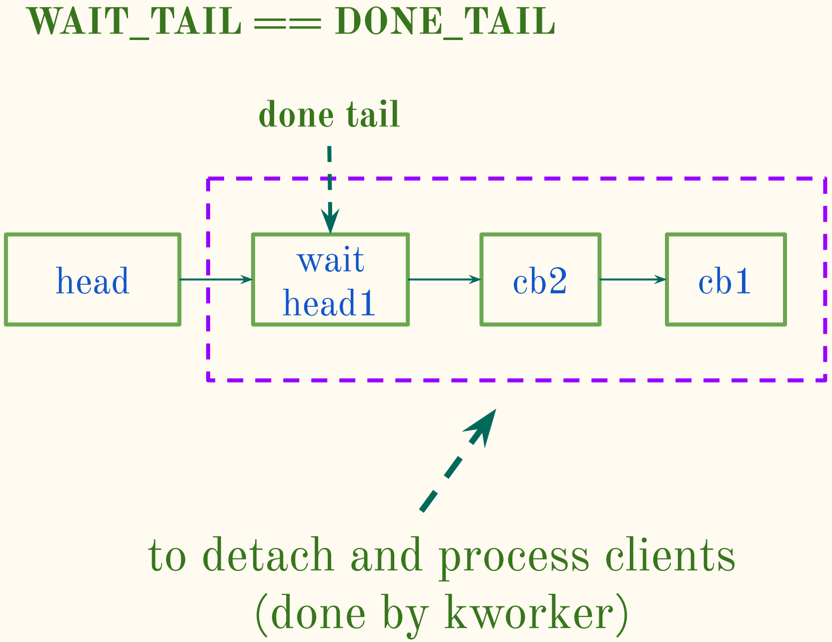
a. initial llist callbacks list



b. new GP1 starts

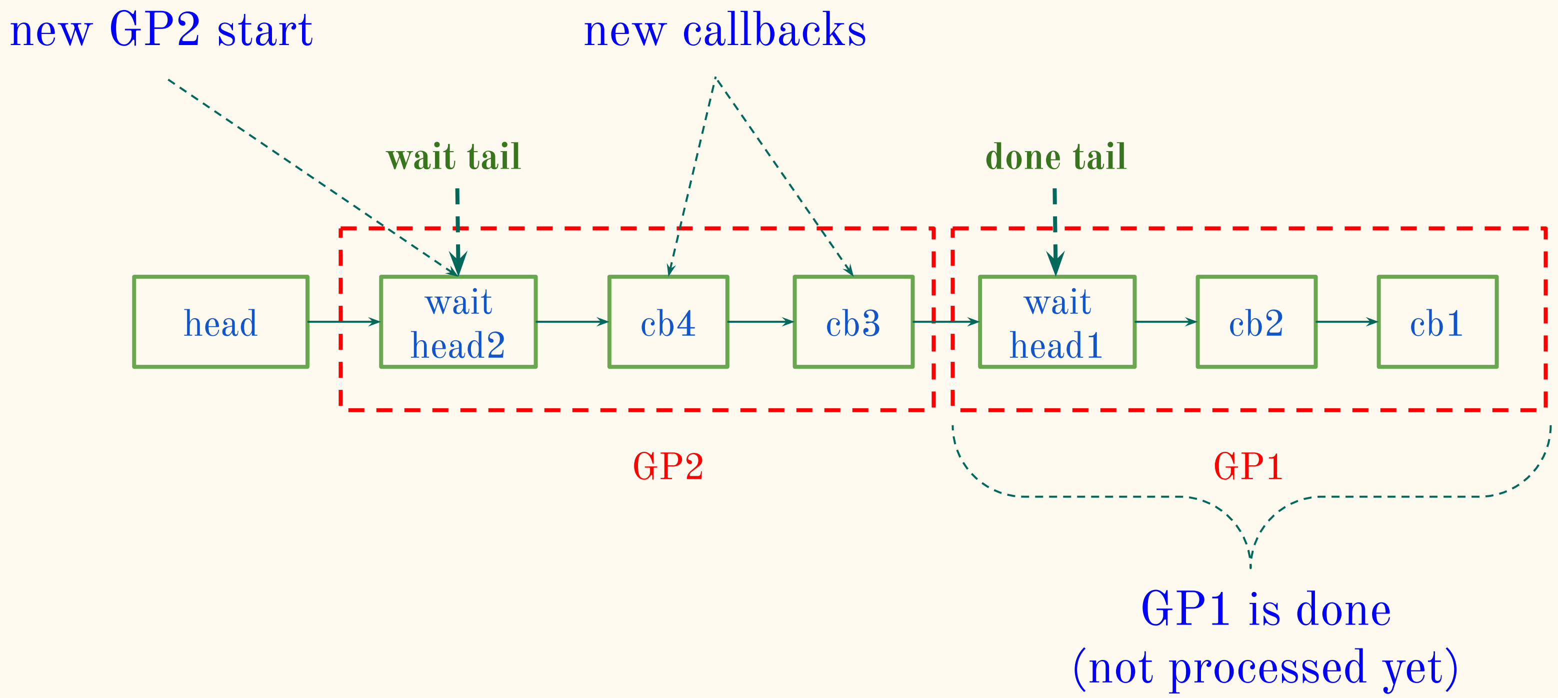


c. GP completion



A state machine and cases(cont.)

d. New callbacks and GP2 start



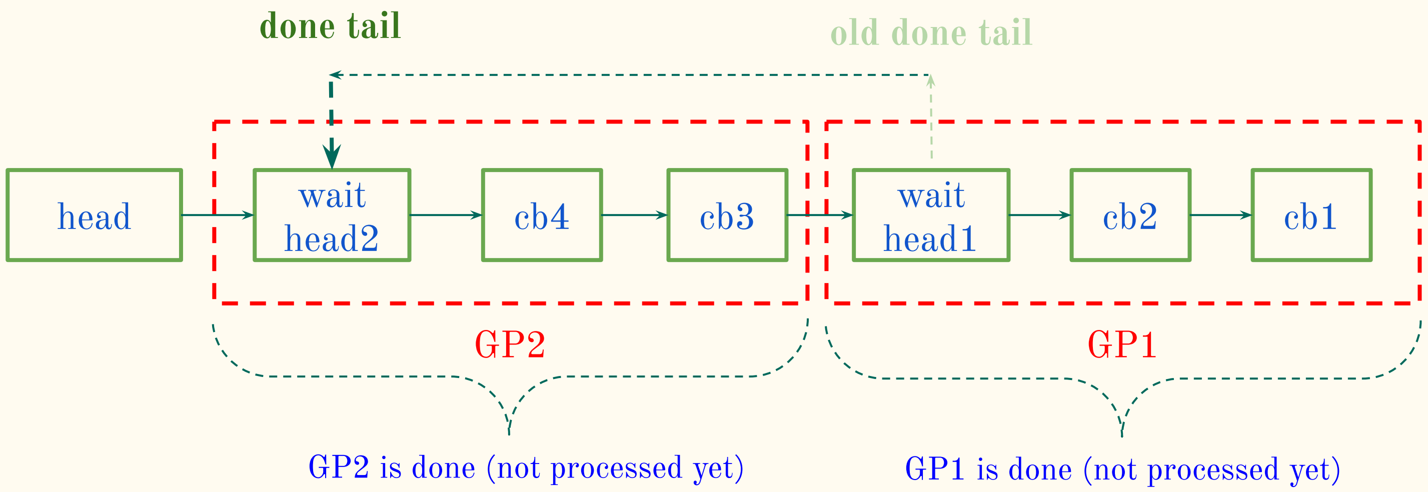
A state machine and cases(cont.)

e. **GP2 completion**

WAIT_TAIL == DONE_TAIL

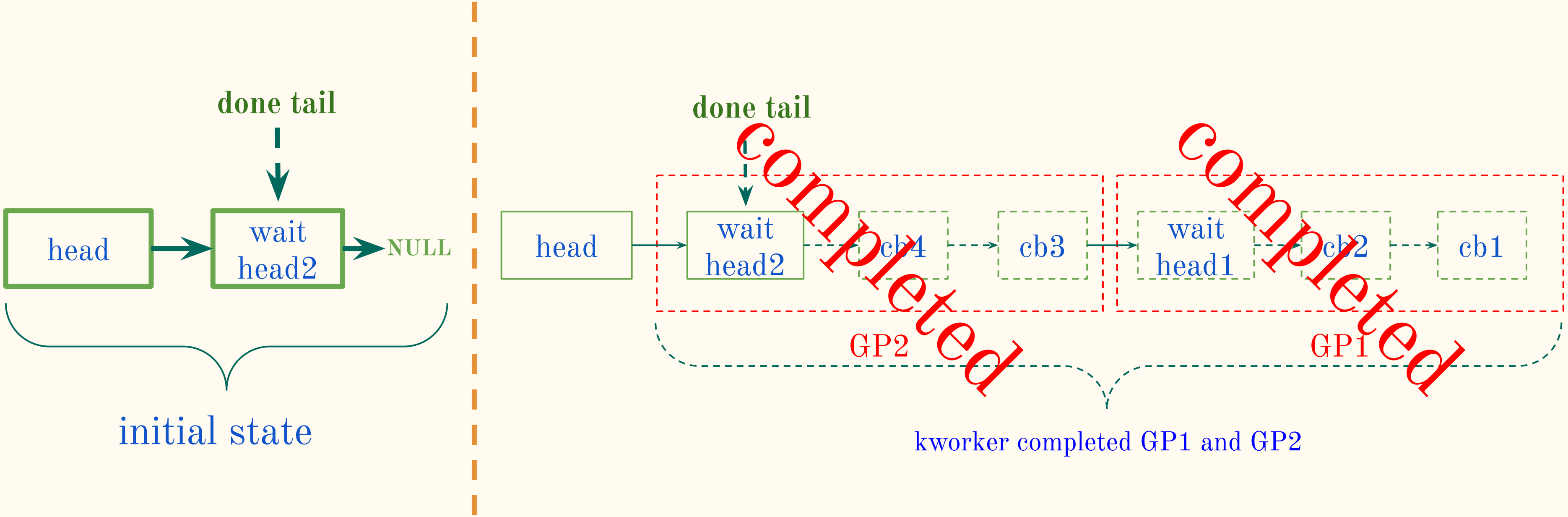
While transition from [d] to [e] state, a kworker can observe either the old done tail [d] or new done tail [e]:

1. *if it sees an old done tail*
2. *newly queued work processes the updated done tail*



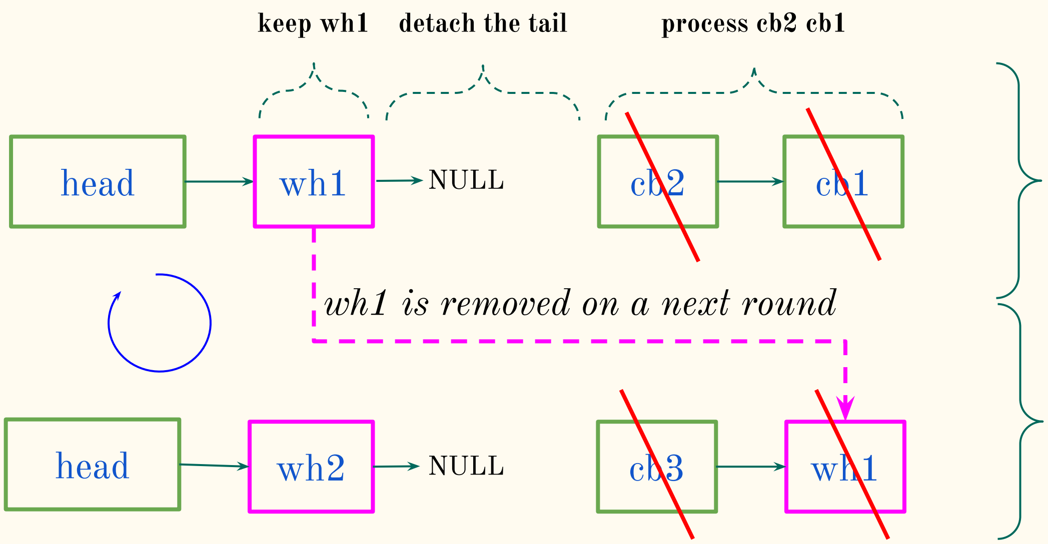
A state machine and cases(cont.)

f. kworker callbacks processing complete



A wait-dummy-node

- A wait-node is inserted on every GP:
  - This allows lockless additions of new users while the cleanup work executes;
  - Dummy-nodes are removed, in a next round of cleanup work execution



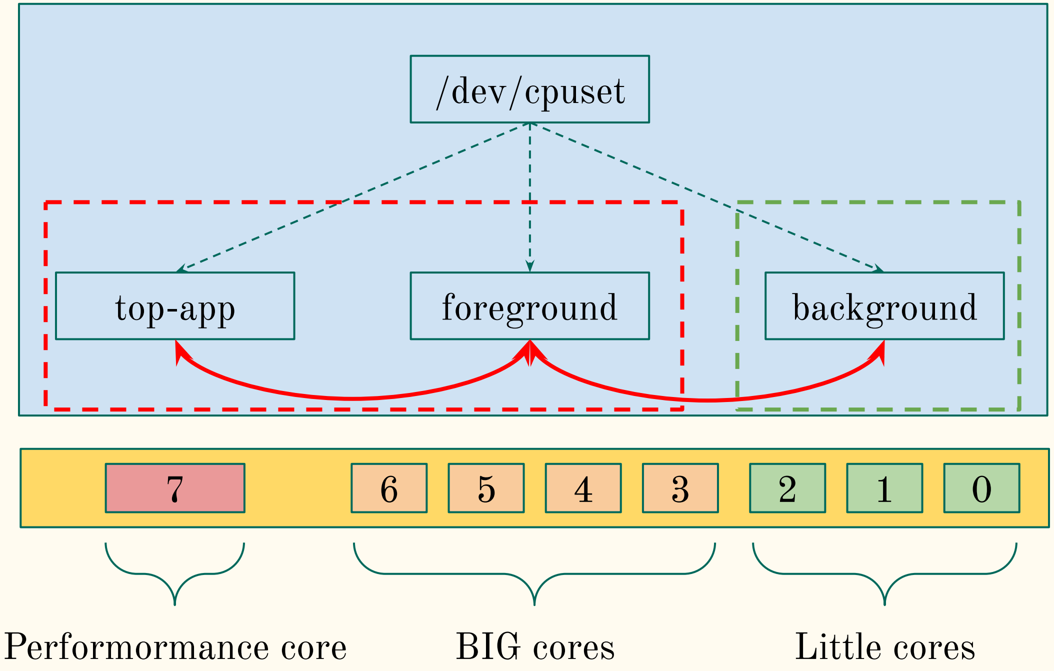
kworker sets wh1 to NULL, it is kept in the list, cb2 and cb1 are completed

kworker sets wh2 to NULL, it is kept in the list, cb3 and wh1 is released for later reuse

Practical example

- One user of synchronize_rcu() is a **percpu-read-write-semaphore**
- Locking for writing, uses synchronize_rcu(), so it is expansive
- CGROUP is a user of such per-cpu semaphore:
    - **cgroup_threadgroup_rwsem** is a per-CPU reader-writer semaphore. When **migrating** a process with all its threads to another cgroup, it needs to **WRITE lock** this semaphore and block **forks** and **exits**, which require the **READ lock**. The purpose is to make the "threadgroup" of a process stable during the migration. Otherwise, there might be new threads in the old cgroup.
- Android uses CGROUP to classify tasks to different groups:
    - top-app, foreground, system-background, etc. For performance and power saving reasons.

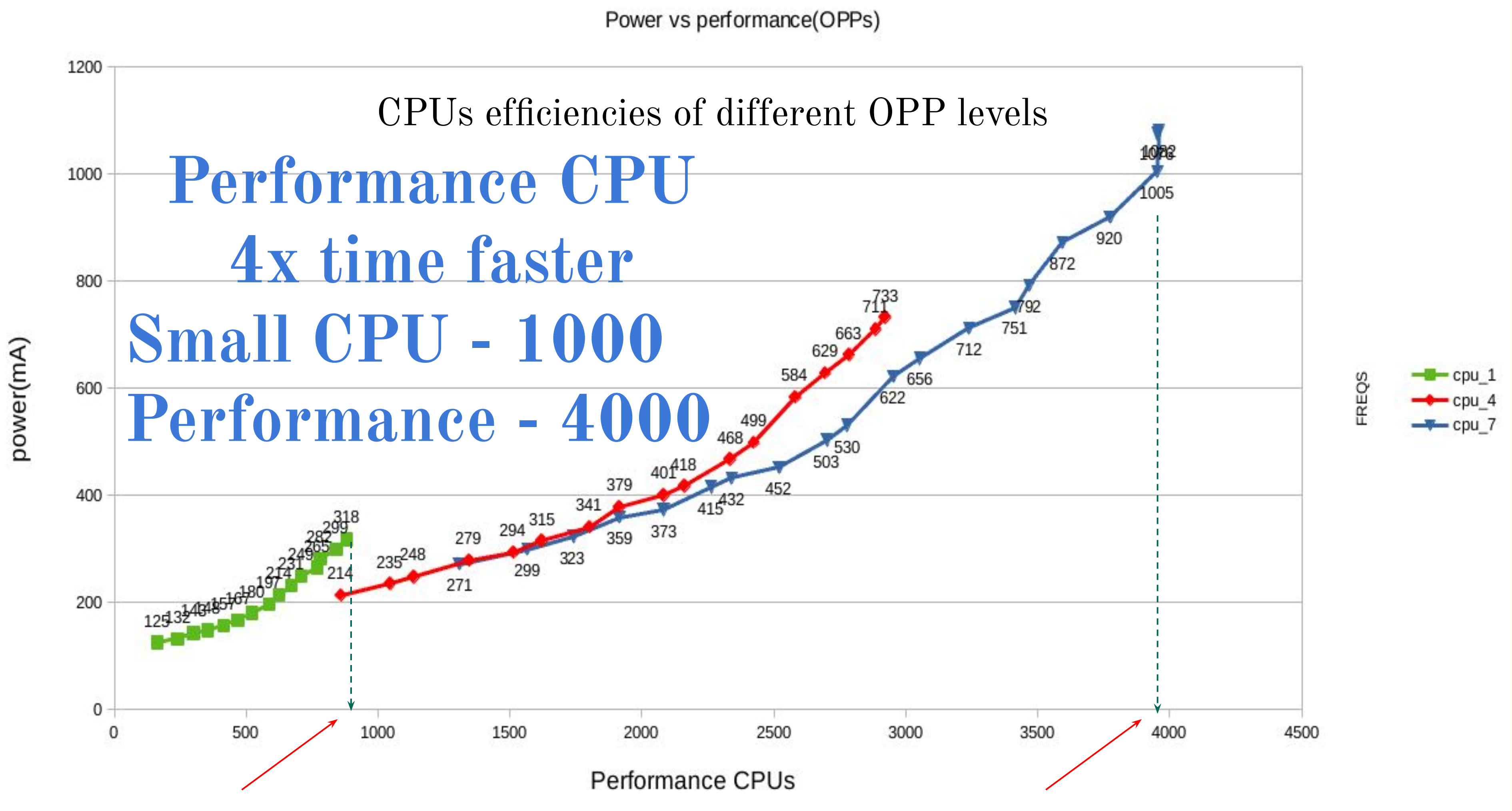
Practical example(cont.)



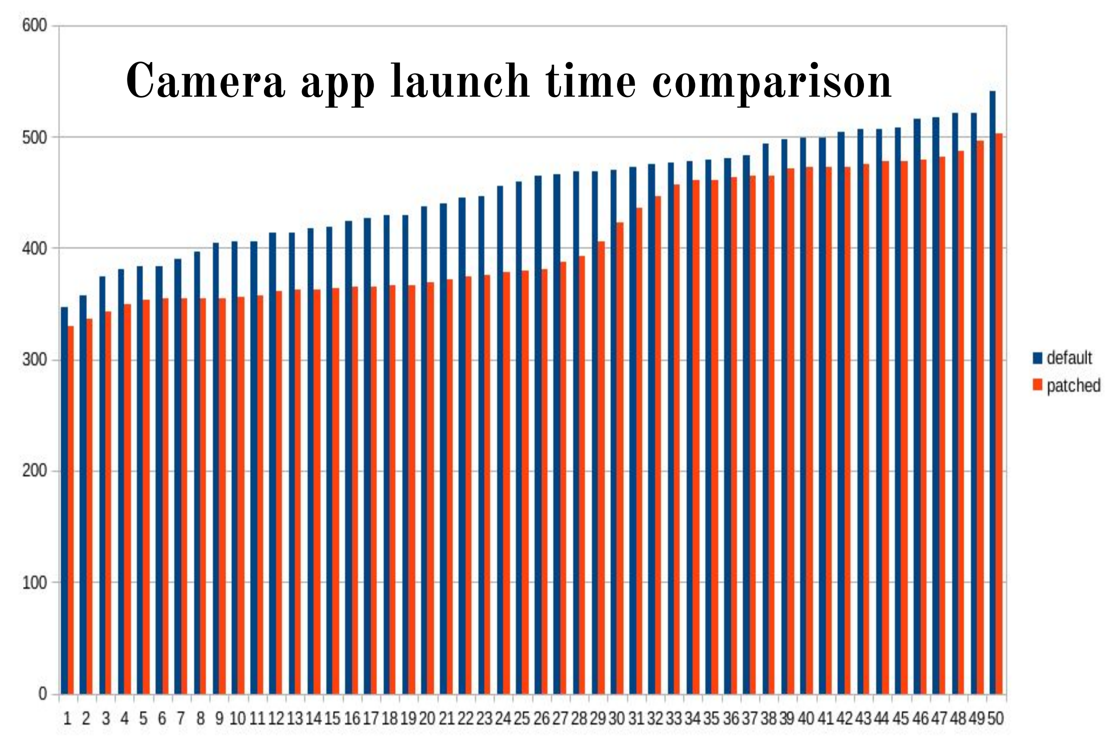
Tasks are moved between groups:

- to reduce app launch latency(especially under heavy background scenario);
- to save power.

Visible apps go to top-app or foreground groups. Non visible go back to background. As a user changes between apps.


Power vs performance(OPPs)
CPUs efficiencies of different OPP levels
Performance CPU
4x time faster
Small CPU - 1000
Performance - 4000
power(mA)
Performance CPUs
FREQS
cpu_1
cpu_4
cpu_7
0
200
400
600
800
1000
1200
0
500
1000
1500
2000
2500
3000
3500
4000
4500

Practical example(cont.)



- 50 iterations;
- time in milliseconds;
- blue is a default;
- red is a patched;
- sorted in ascending order;
- launch time: min/max/median

  3% / 22% / 17%

Practical example(cont.)

Latency distribution histogram(camera app. launch case 50 run)
Patch series on the left the **default** on the right

- 8 - 17 milliseconds
- 15 - 24 milliseconds

```
4
5     #
6     #
7     ####
8     ############
9     ###############
10    ###############
11    ######################
12    ############
13    #################
14    #####################
15    ########################################################
16    #############
17    ##########
18    ####
19    #######
20    ####
21    ##
22    ####
23    ###
24    #
26
27
28
30
```

```
6
7
8     #
9     ##
10    ###
11    #####
12    ##
13    ####
14    ######
15    ####################
16    ########
17    #############
18    ############
19    ##################
20    ########
21    ##########
22    ############
23    ############################
24    ###########
25    #######
26    #########
27    #######
28    ####
29    ####
30    ##
31    ###
32    #
```

Next steps

This work has been merged into **6.10 merge window** as a pull request but we still have some open items to solve.

- Future works:
  - RCU callbacks need a tiny scheduler?
    - One of the possibilities is always putting synchronize_rcu() callback at the head of list: https://lore.kernel.org/rcu/ZTlNogQ_nWUzVJ9M@boqun-archlinux/
  - potential sources of contentions like fixed wait-head-count
  - add per-cpu support
  - processing in a FIFO order

Use it!

Thank you for attention!

Questions?