# Enabling RCU Callbacks to Benefit from Expedited Grace Periods

Linux Storage, Filesystem, MM & BPF Summit 2026

Puranjay Mohan <puranjay@kernel.org>
Paul E. McKenney <paulmck@kernel.org>

# RCU in 60 Seconds

## Readers pay nothing

```
rcu_read_lock();
p = rcu_dereference(gp);
use(p);
rcu_read_unlock();
```

On non-preemptible kernels: zero atomic ops, zero cache misses.

## Writers defer freeing

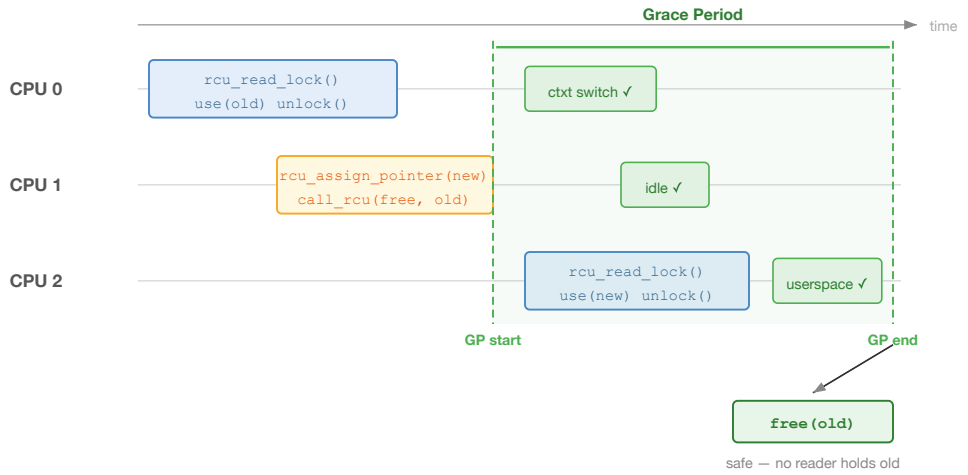
```
old = gp;
rcu_assign_pointer(gp, new);
/* old still live — readers may hold refs */
call_rcu(&old->rcu, my_free);
```

Or block until safe:

```
synchronize_rcu();
kfree(old);
```

## Grace period

A **grace period** waits until every CPU passes through a **quiescent state** (context switch, idle, userspace) — a point where the CPU is guaranteed not to be inside an RCU read-side critical section. After that, no pre-existing reader can still hold a reference.



## How Callbacks Know When to Run

RCU tracks grace periods with a counter.

When `call_rcu()` registers a callback, two things happen:

1. **Snap:** read the current counter and compute a target — "I need one full GP after this point"
2. **Check:** periodically compare the current counter against the target — when it reaches or passes, the callback is ready

```
counter:   ... 5    6    7    8  ...
                    |         |
               call_rcu    counter
               snaps 6,    reaches
               target = 8  target → done!
```

The target is always ahead of the current value — ensuring at least one full grace period elapses between the snap and the callback becoming runnable.

## The Segmented Callback List

Each CPU organizes its callbacks into 4 segments based on their progress through the GP pipeline:

```
head → [CB_A] → [CB_B] → [CB_C] → [CB_D] → [CB_E] → NULL
        ──────   ────────────────   ────────   ──────────────

          DONE       WAIT          NEXT_READY     NEXT
        GP done,   waiting for    assigned to   just arrived,
        ready to   current GP     next GP       no GP yet
        invoke
```

- `call_rcu()` appends to **NEXT**
- **Accelerate**: assigns a GP target and moves NEXT callbacks into:
  - **WAIT** — if no GP is currently being waited on
  - **NEXT_READY** — if WAIT already has callbacks waiting for an older GP
- **Advance**: when a GP completes, moves completed segments → DONE
- **Invoke**: runs DONE callbacks and frees them

NEXT_READY allows pipelining — new callbacks can be assigned to the next GP while WAIT is still waiting for the current one.

Each WAIT and NEXT_READY segment stores a GP counter target. When the counter reaches a segment's target, those callbacks move to DONE.

## Two Kinds of Grace Periods

### Normal GP — `synchronize_rcu()`

- Waits for CPUs to naturally reach quiescent states
- Context switches, idle, userspace
- Latency: typically a few milliseconds

### Expedited GP — `synchronize_rcu_expedited()`

- Sends IPIs to all CPUs to force immediate quiescent state reporting
- Much faster than normal GPs
- Used by: BPF map-in-map updates, netns teardown, mount/umount

### Separate counters, same guarantee

Both prove the same thing — all pre-existing readers have finished. But they use independent state machines:

```
rcu_state.gp_seq /* normal GP counter */

rcu_state.expedited_sequence /* expedited GP counter */
```

They run concurrently and don't know about each other.

Callbacks today only snap and check `gp_seq`. They never look at `expedited_sequence`.

# How RCU Processes Callbacks

## GP kthread (`rcu_gp_kthread`)

A dedicated kthread runs the GP state machine in a loop: init GP → wait for all CPUs to report quiescent states → cleanup → advance `gp_seq`.
When `gp_seq` advances, other parts of the system notice and process callbacks.

## Softirq path (most CPUs)

On every timer tick, `rcu_pending()` checks: did `gp_seq` change? If yes, it raises `RCU_SOFTIRQ` → `rcu_core()`:

- `__note_gp_changes()`: GP completed? advance callbacks WAIT → DONE
- `rcu_do_batch()`: invoke DONE callbacks

## NOCB path (offloaded CPUs)

On `nohz_full` CPUs, there may be no timer ticks. Callback processing is offloaded to kthreads:

- **rcuog** (1 per group of CPUs): sleeps waiting for `gp_seq` to change. Woken by the GP kthread during `rcu_gp_cleanup()`. Advances callbacks.
- **rcuoc** (1 per CPU): woken by rcuog to invoke callbacks.

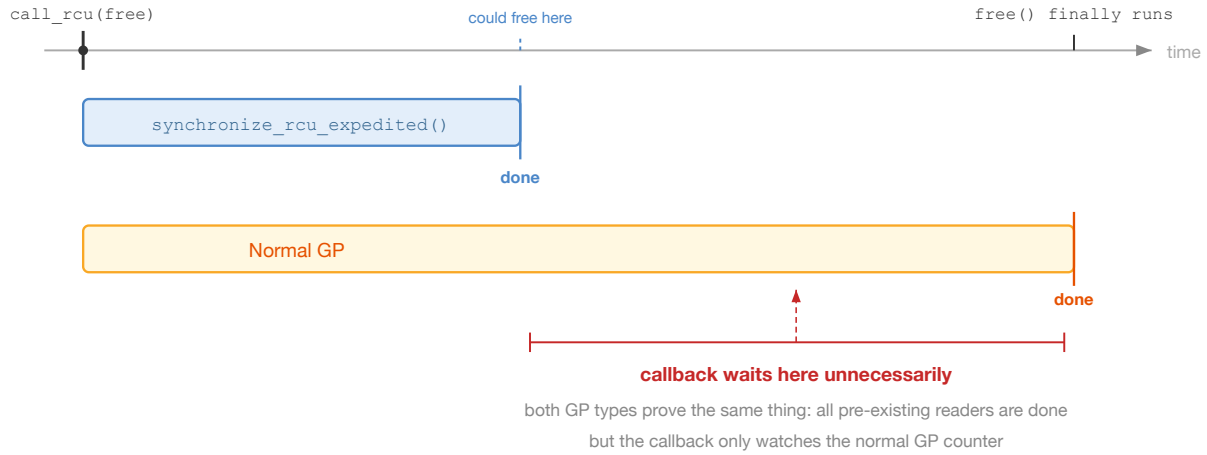
`rcu_core()` on offloaded CPUs skips callback processing — the kthreads handle it instead.

## Key point

The GP kthread drives `gp_seq`. The softirq and NOCB paths react to `gp_seq` changes to process callbacks. None of them watch `expedited_sequence`.

# The Problem

Callbacks only track `gp_seq` (normal). Even if `expedited_sequence` advances — proving all readers are done — the callback doesn't notice.



## Who Cares?

Expedited GPs happen at multiple parts of the kernel including BPF:

- **BPF map-in-map updates** — `maybe_wait_bpf_programs()` calls `synchronize_rcu_expedited()` on every `HASH_OF_MAPS` / `ARRAY_OF_MAPS` update
- **Network namespace teardown** — `net/core/net_namespace.c`
- **Mount/umount** — `fs/namespace.c`
- **Memory management** — `mm/swap.c`, `mm/backing-dev.c`

Any `call_rcu()` callback in the kernel could benefit — socket freeing, inode reclaim, credential release, page table teardown.

The more frequently expedited GPs happen, the more callback latency is wasted.

# The Fix: Track Both GP Types

## Before: one counter per segment

```
struct rcu_segcblist {
    ...
    unsigned long gp_seq[NSEGS];
    ...
};
```

## After: both counters per segment

```
struct rcu_segcblist {
    ...
    struct rcu_gp_oldstate gp_seq_full[NSEGS];
    ...
};

struct rcu_gp_oldstate {
    unsigned long rgos_norm;
    unsigned long rgos_exp;
};
```

## Registration: capture both

```
/* After: both GP counters */
get_state_synchronize_rcu_full(&rgos);
/* rgos_norm = snap(gp_seq)
   rgos_exp  = snap(expedited_sequence) */
rcu_segcblist_accelerate(&rdp->cblist, &rgos);
```

## Advancement: OR logic

```
for (i = RCU_WAIT_TAIL; ...; i++) {
    gs = &rsclp->gp_seq_full[i];
    if (!poll_state_synchronize_rcu_full(gs))
        break; /* neither GP done */
    /* move to DONE */
}
```

Either GP completing is sufficient:

- Normal GP completed? → done
- Expedited GP completed? → done
- Neither? → keep waiting

## Three Notification Gaps

The data structure change was the straightforward part. Three code paths also needed to detect expedited GP completion:

**1. `rcu_exp_wait_wake()` didn't wake NOCB kthreads**

rcuog kthreads monitor GPs for offloaded CPUs. When an expedited GP completed, nobody woke them — they kept sleeping waiting for a normal GP.

Fix: call `wake_nocb_gp()` from `rcu_exp_wait_wake()`.

**2. `rcu_pending()` blind to expedited GPs**

`rcu_pending()` runs on every timer tick and decides whether `rcu_core()` should run. It only checked for normal GP changes.

Fix: add `poll_state_synchronize_rcu_full()` check for pending callbacks.

**3. `rcu_core()` couldn't advance callbacks**

Callback advancement in `__note_gp_changes()` bails out when `rdp->gp_seq == rnp->gp_seq` — which is always true when only an expedited GP changed.

Fix: add a separate advancement block in `rcu_core()` that checks expedited GP completion directly.

# The Benchmark

### What the threads do

### 15 socket threads:

```
while (1) {
     fd = socket(AF_INET, SOCK_DGRAM, 0);
     close(fd);
     /* close → call_rcu() for sock free */
}
```

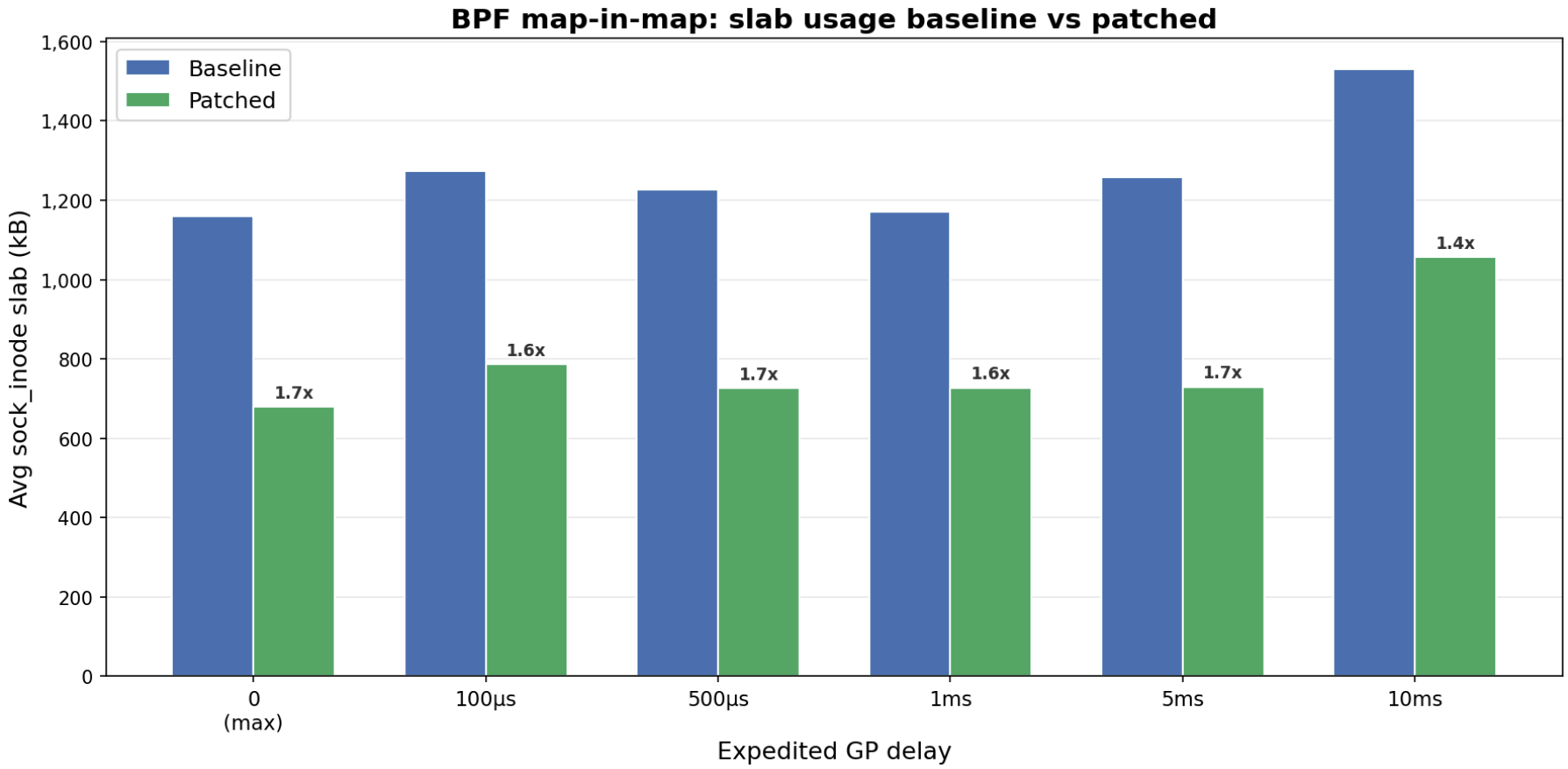
### Expedited GP threads:

```
while (1) {
    bpf(BPF_MAP_UPDATE_ELEM, map_in_map, ...);
    /* kernel: maybe_wait_bpf_programs() → synchronize_rcu_expedited() */
    usleep(delay); /* 0 to 10ms */
}
```

### What we measure

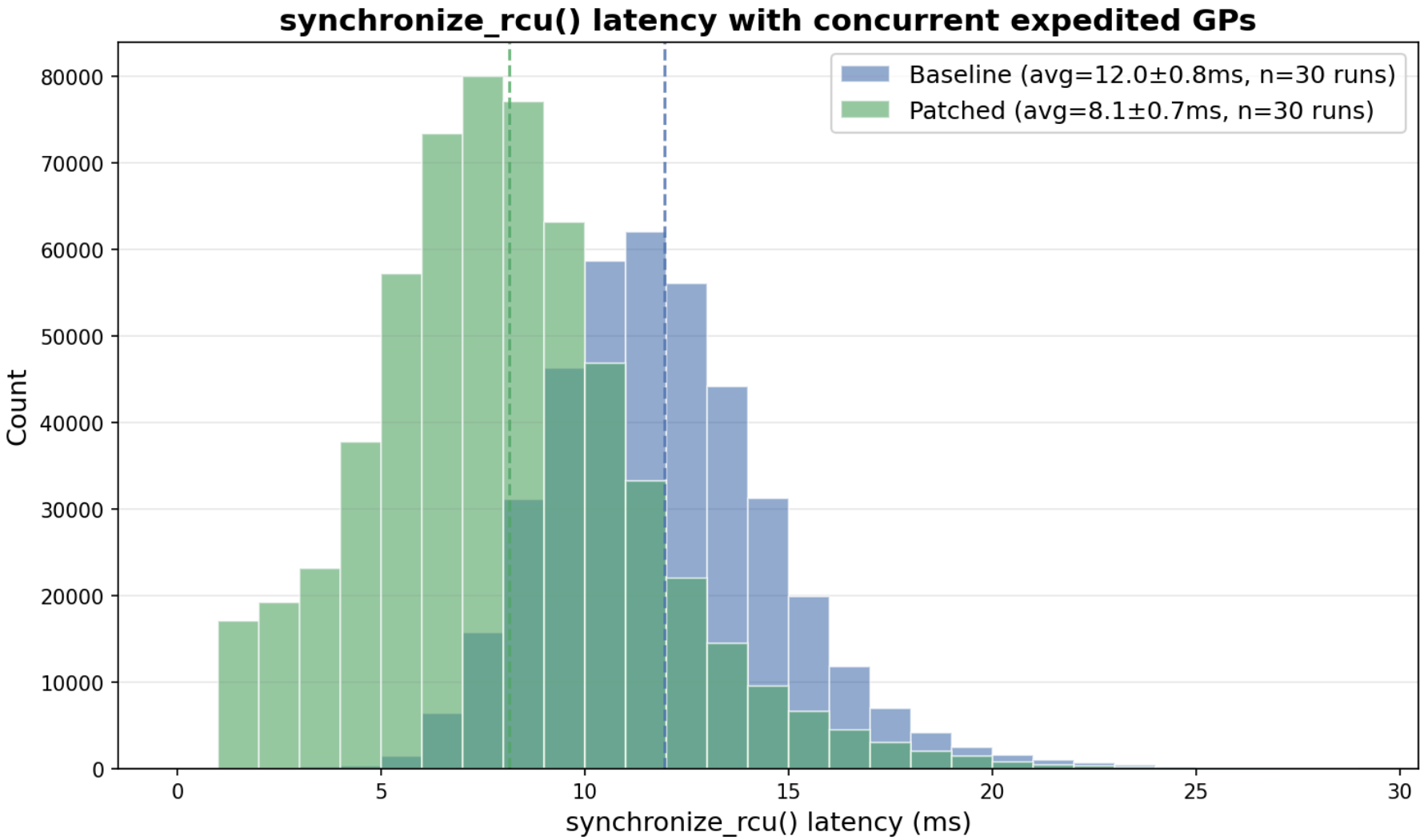
- `sock_inode` slab memory usage (kB)
- Lower => callbacks draining faster
- Sweep expedited GP rate from maximum (delay=0) to one every 10ms
- Compare baseline kernel vs patched kernel on same hardware

# Results: BPF map-in-map + UDP Sockets




Patched kernel uses **1.5-1.7x less slab** across all expedited GP rates — callbacks drain faster because they can complete as soon as the expedited GP finishes.

# Results: synchronize_rcu() Latency Distribution




rcuscale: 16 writers calling `synchronize_rcu()`, 2 concurrent threads calling `synchronize_rcu_expedited()`, 30 runs each. The entire distribution shifts left — baseline averages 12.0ms, patched averages 8.1ms.

## Summary

**The problem:** `call_rcu()` callbacks only track normal grace periods. When expedited GPs complete — proving the same thing — callbacks don't benefit.

**The fix:** Stamp callbacks with both normal and expedited GP counters. `poll_state_synchronize_rcu_full()` checks both — either one completing is sufficient.

**Results:**

- `synchronize_rcu()` latency: 12ms → 8ms with concurrent expedited GPs
- Slab memory: 1.5-1.7x reduction (BPF map-in-map + sockets benchmark)

**Who benefits:** Every `call_rcu()` callback in the kernel — and `synchronize_rcu()` on systems with more than 16 CPUs (which uses `call_rcu_hurry` internally) — whenever expedited GPs happen concurrently.

10-patch series, all rcutorture configurations pass. Posted for review.

# Thanks!



Puranjay Mohan <puranjay@kernel.org>
Paul E. McKenney <paulmck@kernel.org>