# Concurrent CPU Hotplug

## Assumptions

- Initially just concurrently onlining CPUs to speed up the boot process.
- Might later wish to concurrently offline CPUs.
- Would it be necessary to concurrently online and offline CPUs?  Doing this might be more complex, for example, see the smp_mb() in rcu_cpu_starting().

## David Woodhouse Dec 8 2021 Patches

(Review based on a meeting between Neeraj Upadhyay, Boqun Feng, and Paul E. McKenney, cut short by the Google Meet one-hour grace period.)

The email thread is [here](here).

Patch 1: David acquires a new rcu_startup_lock across the whole of rcu_cpu_starting().

Patch 2: David looks at an alternative approach of expanding the scope of the existing rcu_node ->lock to cover more of rcu_cpu_starting(), but needs to handle the fact that rcu_report_qs_rnp() drops that lock.

The problem is that lockdep needs the ->ofl_seq field to have an odd value when acquiring the rcu_node structure's ->lock.  This could be addressed by placing the ->ofl_seq field into the rcu_data structure.

However, grace-period initialization waits for ->ofl_seq to get an even value before propagating CPU-hotplug bitmap changes up the rcu_node combining tree.  It could wait on each CPU in turn, but that would bloat the rcu_gp_init() function's cache footprint.  So exactly why is that wait required?  There is a claim of the potential for a too-short grace period.  If this wait is unnecessary, then there would be no penalty for placing the ->ofl_seq field into the rcu_data structure.

But Neeraj points out the following sequence of events:

1. rcu_cpu_starting() increments ->ofl_seq so that the value is now odd.
2. rcu_cpu_starting() begins acquiring the rcu_node ->lock, and its vCPU is preempted (or whatever delay) while lockdep is in an RCU reader.

3. rcu_gp_init() beats rcu_cpu_starting to acquire the rcu_node ->lock, and thus sets up the next grace period to ignore the incoming CPU.
4. rcu_cpu_starting() gets a nasty surprise when it resumes to find that the RCU-protected data it is referencing has been freed.

Except does lock acquisition cause lockdep to execute any RCU readers?

Maybe rcutorture has an opinion?

Alternatively, how about arch_spin_lock()? [ David took the approach of repurposing the existing ->ofl_seq, but using arch_spin_lock() and arch_spin_unlock() so as to avoid lockdep complaints. This seems like an eminently reasonable approach. ]

# Inspection of RCU Onlining

## rcu_cpu_starting()

Discussed on IRC and over email. The upshot is to also acquire rcu_state.ofl_lock in rcu_cpu_starting(), but to use arch_spin_lock() and arch_spin_unlock() in order to avoid the lockdep issues. David is also looking into replacing ->ofl_seq with arch_spin_is_locked(rcu_state.ofl_lock).

## rcutree_prepare_cpu()

This function is already concurrent-online-ready, except that it invokes rcu_spawn_one_boost_kthread(). As currently written, this could get multiple RCU priority-boosting kthreads ("rcub") per rcu_node structure. I suggest making rcu_spawn_one_boost_kthread() acquire a new rcu_node-structure mutex for serialization.

Do not (repeat, not) attempt to re-use the rcu_node structure's ->boost_mtx or you will be subject to weird race conditions that mess up the priority boosting.

And the rcu_spawn_cpu_nocb_kthread() has the same issue, but with the spawning of the no-CB grace-period kthread ("rcuog"). The new rcu_node structure mutex used to serialize rcu_spawn_one_boost_kthread() can also be used to serialize rcu_spawn_cpu_nocb_kthread().

## rcutree_online_cpu()

The first part of this function looks to be concurrent-online-ready.

The call to the sync_sched_exp_online_cleanup() function is more interesting, with calls to get_cpu() and put_cpu(). But this should still work, as these functions will continue to manipulate preemption state. So no change appears to be needed here.

The call to rcutree_affinity_setting(), which calls rcu_boost_kthread_setaffinity(), manipulates affinity masks.  The new rcu_node-structure mutex should be used to serialize rcu_boost_kthread_setaffinity().  (Not the callers, but the function itself.)

The call to tick_dep_clear() is OK for concurrent onlining, but would require protection by a counter or some such if concurrent onlining and offlining is to be supported.

# Validation

In kernel/torture.c, both torture_online() and torture_online_all() need to exercise concurrent CPU onlining.  Note that the torture tests already avoid concurrency in CPU hotplug operations.  The hope is that the current serialization will continue to allow only one CPU offline or one group of CPU onlines to be in flight at any given time.

The concurrent CPU onlining appears to be x86-only, which means that the old non-concurrent validation will need to be retained.

# Inspection of RCU Offlining

Pingfan Liu proposes concurrent CPU offlining in order to speed up kexec.

## `rcutree_dead_cpu()`

Pingfan Liu's patch takes care of the non-atomic `WRITE_ONCE()` update of `rcu_state.n_online_cpus`.  However, rcu_boost_kthread_setaffinity() needs fixing so that concurrent invocations do not undo each others' work, as was noted in a reply to Pingfan here.

The `tick_dep_clear()` is SMP-safe because it uses atomic operations, but the problem is that if there are multiple `nohz_full` CPUs going offline concurrently, the first CPU to invoke `rcutree_dead_cpu()` will turn the tick off.  This might require an atomically manipulated counter to mediate the calls to `rcutree_dead_cpu()`.

There are several reasons why this call to `tick_dep_clear()` was added:

1. Timekeeping can be blocked while entering a stop-machine event, and the occasional scheduling-clock interrupt can kick things back into action.  However, it is quite possible that the more recent checks for `jiffies` not advancing has made this unnecessary.
2. There might be some dependencies on the tick from timers, and in any case, other unannounced dependencies might have appeared in the past year or so.

But unless and until it can be shown that the calls to `tick_dep_set()` and `tick_dep_clear()` are unnecessary, it will be necessary to provide an `atomic_t`, presumably in the `rcu_state` structure, that is incremented in `rcutree_offline_cpu()` and decremented in `rcutree_dead_cpu()`. If `atomic_inc_return()` is used to do the increment, then the call to `tick_dep_set()` would be conditioned on a return value of one. If `atomic_dec_and_test()` is used to do the decrement, then the call to `tick_dep_clear()` would be conditioned on a return value of zero.