Make Tasks RCU Less Lazy and Intrusive

TL;DR: There is not yet a good way of doing this

Paul E. McKenney, but with help from LKML
February 28, 2024
Updated March 20, 2024 based on Mark Rutland's investigation

The likely advent of lazy preemption (LWN coverage here and here) will mean that even kernels built with CONFIG_PREEMPT_NONE and CONFIG_PREEMPT_VOLUNTARY can sometimes be preempted. This will in turn prevent vanilla RCU from being substituted for Tasks RCU on such kernels, a turn of events that this patch series partially anticipates.

Avoid Tasks-RCU-Specific Annotations

However, it is still proving necessary to add annotations to long-running code, as evidenced by this patch for networking NAPI. The key point is that all current users (as of v6.7) of Tasks RCU are waiting not specifically on all tasks to be voluntarily blocked, but rather for a newly removed trampoline to no longer be in use. Perhaps a more targeted implementation of Tasks RCU could avoid all of these annotations, which was the subject of this email thread and which is summarized by the following sections.

Permanent Trampolines

If the jump-to address could be computed and placed into a known register inline, then a permanent trampoline could disable preemption (or similar) and then transfer to the non-permanent trampoline. The non-permanent trampoline would need to arrange for the code to which it transferred control to come back to the permanent trampoline, which could then enable preemption.

Although this might be simple in concept, it would be quite surprising if it should prove to be feasible, especially on architectures in which trampolines are used as a spectre mitigation.

Enlist the Aid of `cond_resched()` and `might_resched()`

Although `cond_resched()` is a preemption point that calls preempt_schedule_common(), as noted above, all that the current update-side uses of Tasks RCU are doing is to wait for execution to transfer out of any previously removed trampolines. It would therefore be completely legitimate for Tasks RCU to treat `cond_resched()`, `might_sleep()`, and friends as quiescent states.

This is straightforward, except that:

1. This would mean that `cond_resched()` would no longer be a no-op in kernels built with CONFIG_PREEMPT=y.
2. This would go counter to the desire to remove `cond_resched()` completely.
3. This would go counter to the desire for `might_sleep()` to become purely diagnostic.

On the other hand, if there is some other need for `cond_resched()` to remain, this could be an attractive option.

Check for Preemption Within a Trampoline

Given a precise and completely reliable way to determine whether the current preemption occurred within a trampoline, such tasks could simply be marked so that the Tasks RCU grace-period kthread could wait for them to resume. This same determination is required within IPI handlers due to the fact that Tasks RCU must wait for currently running tasks as well as preempted tasks to be free of the old trampoline.

This preferably identifies exactly which trampoline the task was preempted or IPIed within in order to avoid starvation due to an unlucky task always being preempted within a trampoline.

However, this likely adds some sort of data-structure lookup to the scheduler and entry/exit fastpaths.

The need for precision and reliability can be relaxed somewhat. It is OK for false positives that occasionally incorrectly report that the current preemption was from a trampoline. On the other hand, false negatives are completely and utterly fatal.

Mark Rutland investigated this and noted that some trampolines invoke functions, which further complicates the task of determining when there is a trampoline that a given task depends upon.

Make Tasks RCU Check for Trampoline Preemption

If enough information is associated with each preempted task to enable the RCU Tasks grace-period kthread to determine whether or not that task was preempted within a trampoline, then there is no need for any fastpath lookup of a data structure referencing all trampolines. For example, perhaps this information can be extracted from the registers. There is still a need to make this determination for a running task from an IPI handler.

Again, this preferably identifies exactly which trampoline the task was preempted or IPIed within in order to avoid starvation due to an unlucky task always being preempted within a trampoline.

This might be practical, but perhaps only on some architectures. Even on architectures where this is practical, it likely requires addition of architecture-specific code.

Drive-By Fixes

There are some use cases that assume that kernels built with `CONFIG_PREEMPTION=y` need not use Tasks RCU, which in the short term could be adjusted to instead check `CONFIG_TASKS_RCU`:

- `__bpf_tramp_image_release()` in `kernel/bpf/trampoline.c`, which invokes `call_rcu_tasks()` in preemptible kernels and `percpu_ref_kill()` otherwise. Patch sent.
- `ftrace_shutdown()` in `kernel/trace/ftrace.c`, which conditionally invokes `synchronize_rcu_tasks()`. Patch sent.

Other adjustments are likely in order once `CONFIG_PREEMPT_AUTO=y` becomes unconditional. Things to check:

1. Use of the following `Kconfig` options, whether in source code or in either `Kconfig` files or `Makefiles`:
    a. `PREEMPT`
    b. `PREEMPT_NONE`
    c. `PREEMPT_VOLUNTARY`
    d. `PREEMPTION`
    e. `PREEMPT_RT`
2. Architecture-specific code for any architectures that might still not support preemption.

The `kernel/rcu/srcutiny.c` file is an example of an indirect `Kconfig` dependency on `PREEMPTION=n`, which results in `PREEMPT_RCU=n`, which if `SMP=n` in turn results in `TINY_RCU=y` and `TINY_SRCU=y`. This last causes the `make` variable `CONFIG_TINY_SRCU` to have the value "`y`", which in turn causes `kernel/rcu/srcutiny.c` to be included in the kernel build. This file assumed that preemption was completely disabled, and so lazy preemption requires explicit `preempt_disable()` and `preempt_enable()` calls to be added to this file.