Using RCU's CPU Stall Detector

This document first discusses what sorts of issues RCU's CPU stall detector can locate, and then discusses kernel parameters and Kconfig options that can be used to fine-tune the detector's operation. Finally, this document explains the stall detector's "splat" format.

What Causes RCU CPU Stall Warnings?

So your kernel printed an RCU CPU stall warning. The next question is "What caused it?" The following problems can result in RCU CPU stall warnings:

- A CPU looping in an RCU read-side critical section.
- A CPU looping with interrupts disabled.
- A CPU looping with preemption disabled.
- A CPU looping with bottom halves disabled.
- For !CONFIG_PREEMPTION kernels, a CPU looping anywhere in the kernel without invoking schedule(). If the looping in the kernel is really expected and desirable behavior, you might need to add some calls to cond_resched().
- Booting Linux using a console connection that is too slow to keep up with the boot-time console-message rate. For example, a 115Kbaud serial console can be *way* too slow to keep up with boot-time message rates, and will frequently result in RCU CPU stall warning messages. Especially if you have added debug printk()s.
- Anything that prevents RCU's grace-period kthreads from running. This can result in the "All QSes seen" console-log message. This message will include information on when the kthread last ran and how often it should be expected to run. It can also result in the `rcu_.*kthread starved for` console-log message, which will include additional debugging information.
- A CPU-bound real-time task in a CONFIG_PREEMPTION kernel, which might happen to preempt a low-priority task in the middle of an RCU read-side critical section. This is especially damaging if that low-priority task is not permitted to run on any other CPU, in which case the next RCU grace period can never complete, which will eventually cause the system to run out of memory and hang. While the system is in the process of running itself out of memory, you might see stall-warning messages.
- A CPU-bound real-time task in a CONFIG_PREEMPT_RT kernel that is running at a higher priority than the RCU softirq threads. This will prevent RCU callbacks from ever being invoked, and in a CONFIG_PREEMPT_RCU kernel will further prevent RCU grace periods from ever completing. Either way, the system will eventually run out of memory and hang. In the CONFIG_PREEMPT_RCU case, you might see stall-warning messages.

  You can use the rcutree.kthread_prio kernel boot parameter to increase the scheduling priority of RCU's kthreads, which can help avoid this problem. However, please note that doing this can increase your system's context-switch rate and thus degrade performance.
- A periodic interrupt whose handler takes longer than the time interval between successive pairs of interrupts. This can prevent RCU's kthreads and softirq handlers from running. Note that certain high-overhead debugging options, for example the function_graph tracer, can result in interrupt handler taking considerably longer than normal, which can in turn result in RCU CPU stall warnings.
- Testing a workload on a fast system, tuning the stall-warning timeout down to just barely avoid RCU CPU stall warnings, and then running the same workload with the same stall-warning timeout on a slow system. Note that thermal throttling and on-demand governors can cause a single system to be sometimes fast and sometimes slow!
- A hardware or software issue shuts off the scheduler-clock interrupt on a CPU that is not in dyntick-idle mode. This problem really has happened, and seems to be most likely to result in RCU CPU stall warnings for CONFIG_NO_HZ_COMMON=n kernels.
- A hardware or software issue that prevents time-based wakeups from occurring. These issues can range from misconfigured or buggy timer hardware through bugs in the interrupt or exception path (whether hardware, firmware, or software) through bugs in Linux's timer subsystem through bugs in the scheduler, and, yes, even including bugs in RCU itself. It can also result in the `rcu_.*timer wakeup didn't happen for` console-log message, which will include additional debugging information.
- A low-level kernel issue that either fails to invoke one of the variants of rcu_user_enter(), rcu_user_exit(), rcu_idle_enter(), rcu_idle_exit(), rcu_irq_enter(), or rcu_irq_exit() on the one hand, or that invokes one of them too many times on the other. Historically, the most frequent issue has been an omission of either irq_enter() or irq_exit(), which in turn invoke rcu_irq_enter() or rcu_irq_exit(), respectively. Building your kernel with CONFIG_RCU_EQS_DEBUG=y can help track down these types of issues, which sometimes arise in architecture-specific code.
- A bug in the RCU implementation.
- A hardware failure. This is quite unlikely, but has occurred at least once in real life. A CPU failed in a running system, becoming unresponsive, but not causing an immediate crash. This resulted in a series of RCU CPU stall warnings, eventually leading the realization that the CPU had failed.

The RCU, RCU-sched, and RCU-tasks implementations have CPU stall warning. Note that SRCU does *not* have CPU stall warnings. Please note that RCU only detects CPU stalls when there is a grace period in progress. No grace period, no CPU stall warnings.

To diagnose the cause of the stall, inspect the stack traces. The offending function will usually be near the top of the stack. If you have a series of stall warnings from a single extended stall, comparing the stack traces can often help determine where the stall is occurring, which will usually be in the function nearest the top of that portion of the stack which remains the same from trace to trace. If you can reliably trigger the stall, ftrace can be quite helpful.

RCU bugs can often be debugged with the help of CONFIG_RCU_TRACE and with RCU's event tracing. For information on RCU's event tracing, see include/trace/events/rcu.h.

Fine-Tuning the RCU CPU Stall Detector

The rcuupdate.rcu_cpu_stall_suppress module parameter disables RCU's CPU stall detector, which detects conditions that unduly delay RCU grace periods. This module parameter enables CPU stall detection by default, but may be overridden via boot-time parameter or at runtime via sysfs. The stall detector's idea of what constitutes "unduly delayed" is controlled by a set of kernel configuration variables and cpp macros:

CONFIG_RCU_CPU_STALL_TIMEOUT

This kernel configuration parameter defines the period of time that RCU will wait from the beginning of a grace period until it issues an RCU CPU stall warning. This time period is normally 21 seconds.

This configuration parameter may be changed at runtime via the /sys/module/rcupdate/parameters/rcu_cpu_stall_timeout, however this parameter is checked only at the beginning of a cycle. So if you are 10 seconds into a 40-second stall, setting this sysfs parameter to (say) five will shorten the timeout for the *next* stall, or the following warning for the current stall (assuming the stall lasts long enough). It will not affect the timing of the next warning for the current stall.

Stall-warning messages may be enabled and disabled completely via /sys/module/rcupdate/parameters/rcu_cpu_stall_suppress.

RCU_STALL_DELAY_DELTA

Although the lockdep facility is extremely useful, it does add some overhead. Therefore, under CONFIG_PROVE_RCU, the RCU_STALL_DELAY_DELTA macro allows five extra seconds before giving an RCU CPU stall warning message. (This is a cpp macro, not a kernel configuration parameter.)

RCU_STALL_RAT_DELAY

The CPU stall detector tries to make the offending CPU print its own warnings, as this often gives better-quality stack traces. However, if the offending CPU does not detect its own stall in the number of jiffies specified by RCU_STALL_RAT_DELAY, then some other CPU will complain. This delay is normally set to two jiffies. (This is a cpp macro, not a kernel configuration parameter.)

rcupdate.rcu_task_stall_timeout

This boot/sysfs parameter controls the RCU-tasks stall warning interval. A value of zero or less suppresses RCU-tasks stall warnings. A positive value sets the stall-warning interval in seconds. An RCU-tasks stall warning starts with the line:

> INFO: rcu_tasks detected stalls on tasks:

And continues with the output of sched_show_task() for each task stalling the current RCU-tasks grace period.

Interpreting RCU's CPU Stall-Detector "Splats"

For non-RCU-tasks flavors of RCU, when a CPU detects that some other CPU is stalling, it will print a message similar to the following:

```
INFO: rcu_sched detected stalls on CPUs/tasks:
2-...: (3 GPs behind) idle=06c/0/0 softirq=1453/1455 fqs=0
16-...: (0 ticks this GP) idle=81c/0/0 softirq=764/764 fqs=0
(detected by 32, t=2603 jiffies, g=7075, q=625)
```

This message indicates that CPU 32 detected that CPUs 2 and 16 were both causing stalls, and that the stall was affecting RCU-sched. This message will normally be followed by stack dumps for each CPU. Please note that PREEMPT_RCU builds can be stalled by tasks as well as by CPUs, and that the tasks will be indicated by PID, for example, "P3421". It is even possible for an rcu_state stall to be caused by both CPUs *and* tasks, in which case the offending CPUs and tasks will all be called out in the list. In some cases, CPUs will detect themselves stalling, which will result in a self-detected stall.

CPU 2's "(3 GPs behind)" indicates that this CPU has not interacted with the RCU core for the past three grace periods. In contrast, CPU 16's "(0 ticks this GP)" indicates that this CPU has not taken any scheduling-clock interrupts during the current stalled grace period.

The "idle=" portion of the message prints the dyntick-idle state. The hex number before the first "/" is the low-order 12 bits of the dynticks counter, which will have an even-numbered value if the CPU is in dyntick-idle mode and an odd-numbered value otherwise. The hex number between the two "/"s is the value of the nesting, which will be a small non-negative number if in the idle loop (as shown above) and a very large positive number otherwise.

The "softirq=" portion of the message tracks the number of RCU softirq handlers that the stalled CPU has executed. The number before the "/" is the number that had executed since boot at the time that this CPU last noted the beginning of a grace period, which might be the current (stalled) grace period, or it might be some earlier grace period (for example, if the CPU might have been in dyntick-idle mode for an extended time period). The number after the "/" is the number that have executed since boot until the current time. If this latter number stays constant across repeated stall-warning messages, it is possible that RCU's softirq handlers are no longer able to execute on this CPU. This can happen if the stalled CPU is spinning with interrupts are disabled, or, in -rt kernels, if a high-priority process is starving RCU's softirq handler.

The "fqs=" shows the number of force-quiescent-state idle/offline detection passes that the grace-period kthread has made across this CPU since the last time that this CPU noted the beginning of a grace period.

The "detected by" line indicates which CPU detected the stall (in this case, CPU 32), how many jiffies have elapsed since the start of the grace period (in this case 2603), the grace-period sequence number (7075), and an estimate of the total number of RCU callbacks queued across all CPUs (625 in this case).

If the grace period ends just as the stall warning starts printing, there will be a spurious stall-warning message, which will include the

following:

```
INFO: Stall ended before state dump start
```

This is rare, but does happen from time to time in real life. It is also possible for a zero-jiffy stall to be flagged in this case, depending on how the stall warning and the grace-period initialization happen to interact. Please note that it is not possible to entirely eliminate this sort of false positive without resorting to things like stop_machine(), which is overkill for this sort of problem.

If all CPUs and tasks have passed through quiescent states, but the grace period has nevertheless failed to end, the stall-warning splat will include something like the following:

```
All QSes seen, last rcu_preempt kthread activity 23807 (4297905177-4297881370), jiffies_till_next_fqs=3, root ->
```

The "23807" indicates that it has been more than 23 thousand jiffies since the grace-period kthread ran. The "jiffies_till_next_fqs" indicates how frequently that kthread should run, giving the number of jiffies between force-quiescent-state scans, in this case three, which is way less than 23807. Finally, the root rcu_node structure's ->qsmask field is printed, which will normally be zero.

If the relevant grace-period kthread has been unable to run prior to the stall warning, as was the case in the "All QSes seen" line above, the following additional line is printed:

```
rcu_sched kthread starved for 23807 jiffies! g7075 f0x0 RCU_GP_WAIT_FQS(3) ->state=0x1 ->cpu=5
Unless rcu_sched kthread gets sufficient CPU time, OOM is now expected behavior.
```

Starving the grace-period kthreads of CPU time can of course result in RCU CPU stall warnings even when all CPUs and tasks have passed through the required quiescent states. The "g" number shows the current grace-period sequence number, the "f" precedes the ->gp_flags command to the grace-period kthread, the "RCU_GP_WAIT_FQS" indicates that the kthread is waiting for a short timeout, the "state" precedes value of the task_struct ->state field, and the "cpu" indicates that the grace-period kthread last ran on CPU 5.

If the relevant grace-period kthread does not wake from FQS wait in a reasonable time, then the following additional line is printed:

```
kthread timer wakeup didn't happen for 23804 jiffies! g7076 f0x0 RCU_GP_WAIT_FQS(5) ->state=0x402
```

The "23804" indicates that kthread's timer expired more than 23 thousand jiffies ago. The rest of the line has meaning similar to the kthread starvation case.

Additionally, the following line is printed:

```
Possible timer handling issue on cpu=4 timer-softirq=11142
```

Here "cpu" indicates that the grace-period kthread last ran on CPU 4, where it queued the fqs timer. The number following the "timer-softirq" is the current `TIMER_SOFTIRQ` count on cpu 4. If this value does not change on successive RCU CPU stall warnings, there is further reason to suspect a timer problem.

These messages are usually followed by stack dumps of the CPUs and tasks involved in the stall. These stack traces can help you locate the cause of the stall, keeping in mind that the CPU detecting the stall will have an interrupt frame that is mainly devoted to detecting the stall.

Multiple Warnings From One Stall

If a stall lasts long enough, multiple stall-warning messages will be printed for it. The second and subsequent messages are printed at longer intervals, so that the time between (say) the first and second message will be about three times the interval between the beginning of the stall and the first message. It can be helpful to compare the stack dumps for the different messages for the same stalled grace period.

Stall Warnings for Expedited Grace Periods

If an expedited grace period detects a stall, it will place a message like the following in dmesg:

```
INFO: rcu_sched detected expedited stalls on CPUs/tasks: { 7-... } 21119 jiffies s: 73 root: 0x2/.
```

This indicates that CPU 7 has failed to respond to a reschedule IPI. The three periods (".") following the CPU number indicate that the CPU is online (otherwise the first period would instead have been "O"), that the CPU was online at the beginning of the expedited grace period (otherwise the second period would have instead been "o"), and that the CPU has been online at least once since boot (otherwise, the third period would instead have been "N"). The number before the "jiffies" indicates that the expedited grace period has been going on for 21,119 jiffies. The number following the "s:" indicates that the expedited grace-period sequence counter is 73. The fact that this last value is odd indicates that an expedited grace period is in flight. The number following "root:" is a bitmask that indicates which children of the root rcu_node structure correspond to CPUs and/or tasks that are blocking the current expedited grace period. If the tree had more than one level, additional hex numbers would be printed for the states of the other rcu_node structures in the tree.

As with normal grace periods, PREEMPT_RCU builds can be stalled by tasks as well as by CPUs, and that the tasks will be indicated by PID, for example, "P3421".

It is entirely possible to see stall warnings from normal and from expedited grace periods at about the same time during the same run.