Contribution ID: **228** Type: **not specified**

Linux kernel support for kernel thread starvation avoidance

Tuesday, 21 September 2021 08:25 (35 minutes)

ABSTRACT

Running CPU-intensive high-priority real-time applications on a real-time Linux kernel (based on the PREEMPT_RT patchset) can lead to situations where the kernel's own housekeeping tasks such as per-cpu kernel threads get starved out, resulting in system instability (hangs/unresponsive system). The Real-Time Throttling feature in the Linux kernel is ineffective in addressing this problem as it does not protect low-priority real-time kernel threads (such as ktimersoftd). The stalld userspace daemon was introduced to solve this problem, and is quite effective in principle; but it has a number of limitations that makes it hard to use in practice, especially in production deployments. We propose implementing stalld-like starvation avoidance for kernel threads directly in the Linux kernel, to address all the practical limitations of stalld. This design scales well with the number of CPUs, has minimal monitoring overhead (CPU usage), and compartmentalizes the fault-domain such that a misbehaved or misconfigured real-time application does not bring down the entire system.

INTRODUCTION

The Telco industry is undergoing a major revamp of its infrastructure at the edge (cell towers) as well as the core datacenter, in order to meet the demands of 5G networking. As part of this effort, the underlying infrastructure called the Radio Access Network (RAN), which was traditionally implemented in hardware (FPGAs) for low-latency predictable real-time response, is being replaced with software-defined RAN applications running on real-time Linux kernel (PREEMPT_RT). These soft real-time applications involve running CPU-intensive high-priority real-time tasks, to meet the stringent latency requirements as defined by the 5G/3GPP specification.

There are a number of challenges that the Linux real-time stack needs to address to support this new class of workloads. This proposal focuses on system stability issues when running CPU-intensive high-priority real-time applications on the PREEMPT_RT Linux kernel and highlights the open issues and proposes a novel design to address the limitations of existing solutions by implementing kernel-thread starvation avoidance in the Linux kernel.

PROBLEM STATEMENT

In the Telco/5G Radio Access Network (RAN) usecase, deploying the application involves running high-priority CPU hogs such as "L1 app" (based on Intel's FlexRAN and DPDK Poll-Mode-Driver). These latency-sensitive tasks are bound to isolated CPUs and they run infinite polling loops (in userspace) with high real-time priority (typically SCHED_FIFO/90+). In this scenario, even if the L1 app RT tasks don't invoke kernel services by themselves, generic (non-RT) workloads running on non-isolated CPUs (such as Kubernetes control plane tasks) can cause per-CPU kernel threads to wake up on every CPU.

However, such kernel threads on isolated CPUs running the L1 app RT tasks will get starved out, since the L1 app never yields the CPU.

One of the consequences of starving out essential kernel threads is system-wide hangs. As an example, if a container gets destroyed (from non-isolated CPUs), the corresponding network namespace teardown code in the Linux kernel queues callbacks on per-CPU kworkers, and invokes flush_work(), thus expecting the per-CPU kworker on every CPU to participate in the teardown mechanism. As a result, the container destroy will get hung indefinitely due to kthread starvation on CPUs running the L1 app RT tasks. Furthermore, since this code path holds the rtnl_lock, any other task that touches kernel networking will end up getting stuck in uninterruptible sleep ('D' state) too (eg: sshd, ifconfig, systemd-networkd etc.), thus cascading to a system-wide hang.

This pattern of kernel subsystems invoking per-CPU kernel threads for synchronization is quite pervasive throughout the Linux kernel, and the resulting kthread starvation issues go well beyond the specific networking scenario highlighted above. Furthermore, even essential real-time configuration tools and debugging utilities such as tuned and ftrace/trace-cmd themselves rely on kernel interfaces that can induce such starvation issues.

EXISTING SOLUTIONS AND LIMITATIONS

The community tried to address the problem of system instability caused by running CPU-intensive high priority real-time applications in LPC Real-Time microconference 2020 by introducing stalld. The stalld userspace daemon monitors the system for starving tasks (both userspace and kernel threads), and revives them by temporarily boosting them using the SCHED_DEADLINE policy. It achieves this revival and system stability by operating within tolerable bounds of OS-jitter as configured by the user.

We have been using stalld along with RAN applications and it has been quite effective in maintaining system stability. However, we have also come across a number of limitations in stalld, owing to its design as well as the choice to implement starvation monitoring and boosting in userspace. We would like to bring out stalld's pain-points and then discuss a prototype that we have developed to address these concerns, by implementing stalld-like kernel-thread starvation avoidance directly in the Linux kernel.

Limitations of stalld in resolving kthread starvation:

1. Stalld does not scale with the number of CPUs

Stalld spawns a pthread for every CPU to monitor and boost starved tasks on the respective CPU. However, in RAN usecases, due to the use of CPU isolation, all of stalld's threads are forced (bound) to run only on the housekeeping CPUs, which are typically a small subset of the available CPUs in the system. For example, on a 20 CPU server with CPUs 2-19 isolated to run RT tasks, potentially 20 stalld threads compete for CPU time on housekeeping CPUs 0-1, trying to monitor and boost starved tasks on all the 20 CPUs.

2. Stalld can get starved itself

Since stalld runs as a normal priority task, higher priority tasks (or even a high volume of similar priority tasks) running on the housekeeping CPUs can starve out stalld itself. Attempting to solve this problem by turning stalld into an RT application is risky, as it can make the situation worse – since all of stalld's per-CPU monitoring threads put together can potentially consume all the available CPU time on the housekeeping CPUs (depending on how

aggressive the stalld configuration is), real-time stalld can end up causing starvation itself!

3. Stalld's logging is unreliable

On systemd-based Linux installations, stalld logs its output related to starvation conditions and boosting events to journalctl logs via systemd-journald. However, in most situations involving system-wide hangs, systemd-journald gets stuck in uninterruptible state too, leaving no trace of stalld's execution flow and boosting decisions.

4. Trade-off between time-to-respond vs CPU consumption

One of the other concerns with stalld's design is the use of per-CPU threads for starvation monitoring and boosting, which can be CPU intensive. To address this problem, stalld supports a single-threaded mode of operation to monitor the entire system, but trades-off the time-to-respond to starvation conditions in exchange for lesser CPU consumption. However, this is a tricky trade-off for the system administrator in practice, since typical starvation issues arise from per-CPU kthreads woken on every CPU and demand quick boosting/revival on every CPU for system stability.

Considering these limitations of stalld for practical deployments, we have developed a prototype design to address these concerns by implementing stalld-like kernel-thread starvation avoidance directly in the Linux kernel.

DESIGN OF PROPOSED SOLUTION (IN-KERNEL KTHREAD STARVATION AVOIDANCE)

Our design to address the limitations of stalld builds on the following key insights:

1. Compartmentalize the fault-domains of the RT application & the OS

System-wide hangs (as described above) are almost always caused by starving kernel threads, which may be the result of a misconfigured real-time application. However, ensuring that kernel threads never starve (using an in-built starvation-avoidance algorithm in the kernel) will keep the OS stable, while limiting the hangs or starvation issues to the misbehaving application itself. A misconfigured RT application can no longer bring down the entire OS.

2. Starvation avoidance via (per-CPU) scheduler-hooks scales well

In a typical real-time RAN application deployment, CPU isolation is used to move all movable tasks to housekeeping CPUs, so as to run the real-time application on the isolated CPUs. In such a configuration, the only remaining kernel threads on the isolated CPUs are non-migratable per-CPU kthreads such as ktimersoftd, per-CPU kworkers etc., and those are the ones that are likely to get starved out. Therefore, the problem of identifying starved kernel-threads and reviving them via priority boosting is naturally CPU-local, and it can be implemented without the need for system-wide monitoring or cross-CPU coordination.

The Linux kernel scheduler uses a per-CPU design for scalability. Hence, implementing per-CPU kernel thread starvation avoidance by directly hooking onto the scheduler should automatically scale well.

3. Kernel-based design lends itself to an elegant implementation

Implementing starvation monitoring and revival for kernel-threads in the Linux kernel itself offers a number of surprising benefits, including the ability to elegantly side-step entire problem classes altogther, as compared to a userspace solution, as noted below.

`3A. Efficiency`

-

The in-kernel implementation allows hooking the starvation avoidance algorithm to specific events of interest within the scheduler (such as task wakeups) which helps minimize unnecessary periodic monitoring activity, thus saving CPU time.

```
3B. No risk of starving the starvation avoidance mechanism
```

-

In NOHZ_FULL mode, a single task can effectively monopolize the CPU without ever entering the kernel; but luckily this also means that there is no chance of starvation since there is only one task eligible to run on that CPU. Waking up any other task targeted for that CPU will invariably invoke the scheduler, which gives the opportunity to run starvation avoidance as needed.

This design also side-steps problems that arise with userspace solutions such as deciding the scheduling policy and priority at which stalld runs so as to not get starved itself.

IMPLEMENTATION OUTLINE

We have developed a prototype that implements the design envisioned above by using scheduler hooks in the Linux kernel as well as hrtimer callbacks. A brief outline is presented below.

When a task gets enqueued into a CPU's runqueue, the "stall monitor" code arms a starvation-detection hrtimer (if not already armed) to fire after a (user-configurable) starvation-threshold, iff the task that was enqueued was a kernel thread.

Once the starvation-detection timer fires, the stall monitor code checks if the set of runnable kernel threads on that CPU have been starving for the threshold duration. If it detects starvation, it arranges to boost the kernel threads (one-by-one) using the SCHED_DEADLINE policy in the irq_exit() path of the hrtimer interrupt, and arms a deboost hrtimer to fire after the (user-configurable) boost duration.

The deboost timer's callback restores the scheduling policy and priority of the boosted kernel thread to its original settings.

We are still working on revising this basic design and implementation, and we are looking forward to share more details at the conference and seek the Linux real-time community's invaluable feedback for further improvements or better alternatives.

CONCLUSION

The Telco Radio Access Network (RAN) for 5G is an exciting avenue that brings a new class of real-time workloads to Linux. While the Linux real-time stack based on the PREEMPT_RT patchset has been used with great success for decades with tightly controlled real-time applications and system configuration, the Telco/RAN usecase challenges the status quo by demanding lower real-time latency than ever before, while co-existing with non-real-time workloads as generic (i.e., not tightly controlled) as Kubernetes.

One of the major pain-points faced by the industry in running these workloads on Linux is instability of the underlying OS itself, often times triggered by the very tools that are used for Linux real-time system configuration, tracing and debugging! In this proposal, we discussed the most promising current solution in this problem space, namely stalld, and highlighted its limitations as observed in practical deployment scenarios. We proposed an alternative design that addresses these limitations by implementing stalld-like kernel thread starvation avoidance in the Linux kernel itself.

We are looking forward to the Linux community's insightful feedback on our design, as well as invaluable suggestions more broadly on solving OS stability issues for RAN-like usecases that involve running CPU-intensive high priority real-time tasks.

I agree to abide by the anti-harassment policy

I agree

Primary authors: TURLAPATI, Sharan; BHAT, Srivatsa (VMware)

Presenters: TURLAPATI, Sharan; BHAT, Srivatsa (VMware)

Session Classification: Real-time MC

Track Classification: Real-time MC