Full Dynamic Ticks

Summary of the discussion led by Frederic Weisbecker during the RT workshop at LPC

Why was full dynamic ticks designed?

Difference between NOHZ_IDLE and NOHZ_FULL

NOHZ_IDLE : As long as a CPU is idle, the periodic tick is halted. This is to ensure **powersavings**.

NOHZ_FULL : As long as a CPU is running just one task, the periodic tick is halted.
Workloads which need a **high response time** can benefit when the periodic ticks are halted.

Are there no side effects of getting rid of the periodic tick when the CPU is running ?

Periodic tick verifies if task running on a CPU has to be preempted, ensures that the timekeeping is updated and updates cpu and task group load.

Periodic tick is not necessary since there is no need to preempt task. There is one CPU which takes care of time. And the load gets updated when a new task comes on the runqueue.

This framework was merged in 3.10 kernel.

The contribution from NOHZ_FULL are:

The following features were designed to support NOHZ_FULL. But they can potentially be useful by themselves.

NO_RCU callbacks : CPUs which have pending RCU callbacks are not allowed to enter tickless idle. Therefore for CPUs to enter tickless idle they must be specified in a kernel parameter. RCU callbacks are offloaded from such CPUs.

Sysidle detection : Full system idle detection. NOHZ_FULL mandates that the boot CPU has to keep time **always.** Which means it can never enter tickless idle. This was the compulsion up until recently. There is no harm in the boot CPU entering tickless idle when the entire system is idle. Sysidle detection is a state machine designed by Paul McKenney which indicates when the system is idle. This is a solution based on RCU.

Context Switch Tracking: NOHZ_FULL needs to keep track of context switch between user space and kernel space to enable some housekeeping work.

Tickless cputime accounting : This is being done at well defined boundaries such as tick_start, tick_stop, cputime report to user.

The above contributions has actually made the NOHZ_IDLE framework function much better as a side effect. But the NOHZ_FULL is overall large tricky and fragile code with few qualified reviewers and has a large overhead when enabled today.

Results on running a microbenchmark

1000Hz -> full dynticks =+ 2-3% performance
100Hz -> full dynticks =+ 0.003% performance

Future:

1. Distros want it to be available right away. However there is still work to be done to ensure that there is no compile time/run time overhead when full dynticks is not used.

2. There are certain use cases which demand extreme real time, which cannot afford any interruption. This needs more work because today the scheduling tick fires on NOHZ_FULL cpus too, except after longer periods. This is required to keep the scheduler happy. But the maintainers are for removing that residual tick if scheduler can tolerate inaccurate time keeping.

3. Another suggestion was to move all the scheduler ticks with max deferment of 300s or so to the cpu0. Instead of overloading one cpu, there could be a cpumask which could do this.

4. Unbound timers and workqueues also need to be moved to completely isolate the cpu.

5. But the overhead of full dynticks is high. There was interest in using full dynticks for isolating cores. Although full dynticks has the capacity to achieve this to some extent, it will incur a high overhead. One of the reasons is we need to track context switch from user to kernel. .

Read Write Semaphores: This discussion was led by Steven Rostedt

The philosophy of **PREEMPT-RT** (which is real time linux) is that tasks should not experience huge latencies. Not even preemption delays. This means that critical sections, interrupt handlers are preemptible. What are the consequences of this?

There are some problems that PREEMPT-RT faces in synchronization that mainline does not sffer from.

Let us revisit the problem of priority inversion to begin with.

Let us call three tasks L,M,H in the increasing order of priority. If L holds a lock R, and H attempts to take R, it does not get to run. However M being of higher priority can preempt the task L. M gets to run sooner than H. So priority inheritance is used to temporarily give L the priority of H so that H can run next.

Now let us consider the issue with spinlock on RT

Spinlocks are preempt_disabled on mainline. However on Real Time kernels they protect *preemption enabled sections*. Which means they can experience the *priority inversion issue*. Hence real time kernel specifically has to *disable preemption before taking the spinlocks.*

But what about **semaphores** ? We cannot disable preemption since it does not make sense to do so with semaphores. They are going to sleep while holding a semaphore, which can lead to priority inversion anyway. So *real time kernels do a priority inheritance solution for the semaphores.*

But there are issues with respect to real time kernels here. Let us see what happens on mainline.

Mainline design of RW semaphores

RW semaphores are preemptible even on mainline kernels. Which means the writer can experience priority inversion issues. So they too do a priority inheritance solution. But to avoid writer starvation due to multiple readers, readers serialize when the writer is blocked. But when the writer is not blocked, they can run in parallel.

On RT however, even when the writer is not blocked, readers cannot run in parallel. They are made to serialize to avoid preemption latency for the writer which can arrive anytime. This will affect workloads drastically. The users of RW semaphores which are majorly causing this damage are users of **mmap_sem, page fault handling, threaded workloads like Java**.

Peter Zijlstra is looking to get rid of as much usage of mmap_sem, particularly in the page fault handling code by looking at RCU like look up. There are plans to get rid of semaphores in the Thp-Mem compaction code too.

State of Preempt_RT : This discussion was led by Sebastian A. Siewior, who maintains it today

How does the development of Preempt_RT work?

1. The RT patches are rebased on the mainline for each vanilla release
2. Rebase is not smooth. The core kernel code gets reorganized
3. Even if the code compiles, it does not boot. So the solution is disable something temporarily and verify if it boots. There could be RCU stalls. Sometimes there is absolutely no output.

Hence it is not a fun job to maintain RT kernel. The difference between the fundamental design points between RT and mainline will only get worse as the development pace quickens on the mainline. It will get harder to keep up to integrate the RT patches.

-RT changes the core kernel code which also undergoes some optimizations. Very few people care about -RT kernel. There is very little funding.

What is fundamentally different about PREEMPT-RT which makes it difficult to easily merge it with mainline ?

1. spin_lock_irq() does not disable interrupts

2. interrupts are threaded. [Softirq is one way of reducing interrupt handling latency. Another is threaded interrupt handlers].

3. The portion where interrupts are disabled should be really short

4. The preempt disabled sections should be broken into smaller regions.

There have been very good ideas that have come out of RT like lockdep. There can be better scope for the betterment of the mainline if -RT development is supported and funded.