How to not break PREEMPT_RT

Linux Plumbers Conference

Dublin, Ireland September 12-14, 2022

Sebastian A. Siewior

Linutronix GmbH

September 12, 2022

Userland is important

- The focus is on userland.
- The goal is schedule userland thread as soon as possible.
- Low latency between request to schedule and the actual schedule.
- The non-preemptible context is reduced. Non-preemptible as in
  - interrupts are disabled.
  - preemption is disabled.

Locking

- spinlock_t and rwlock_t sleep while blocked instead of spinning.
- Similar to mutex_t:
  - While blocking on the lock (sleeping), task state is preserved.
  - While owning the lock, the context is never migrated to another CPU.
  - CPU shutdown (CPU hotplug) is delayed.
  - RCU won’t complian if with rcu_read_lock().
- The lock owner can be preempted.
- PREEMPT_LAZY avoids preemption if possible. (SCHED_OTHER vs elevated priority).

Interrupt handlers are threaded (as with ”threadirqs”)

- Handlers which must not be threaded need IRQF_NO_THREAD.
- Implicit example is the timer interrupt.
- Perf interrupt needs to marked explicit.
- Driver are usually fine without.
- Small/ quick routines are ”okay”, disabling interrupt source.
- Driver in need are interrupt controller, multiplexer.

Softirq context is preemptible

- Usually the softirq is ”raised” within the interrupt handler and then invoked on the exit from hard IRQ.
- With threaded interrupt handlers, the softirq is invoked after the threaded handler completed within its thread.
- Softirqs which are ”raised” from hard-IRQ context are deferred to ”ksoftirqd”.
- Once ksoftirqd is active, it will process all following requests.

hrtimer callback is invoked in softirq context

- Handler needs HRTIMER_MODE_HARD if hardirq context for callback execution is needed.
- watchdog, needs to act if the CPU is locked up, not preemptible.
- Usually hardirq context is not needed.
- The needed infrastructure is often not hardirq compatible.
- clock_nanosleep() from RT tasks are handed in hardirq.

irq_work callbacks are invoked in a thread

- Handler needs IRQ_WORK_INIT_HARD() if hardirq context for callback execution is needed.
- Often used if schedule_work() can’t be used.
- nohz_full_kick_func() needs to observe idle CPU.
- wake_up_klogd_work_func() console print or wake up from any context.

RCU callback is invoked from a thread (”rcutree.use_softirq=0”)

- Less softirq work. No explicit softirq requirement.
- The invocation of callbacks can be preempted.
- Config option RCU_BOOST is enabled by default. For preempted RCU reader.
- Callback threads can be moved to other CPU (CPU isolation).

Additional synchronisation points for "spin until ready"

```
CPU0-Thread0                CPU1-Thread1
                            preempt_disable(); /* implicit, part of spin_lock() */
                            active = 1
while (active)              do1();
      cpu_relax()           do2();
                            do3();
                            active = 0;
/* no longer active */      preempt_enable(); /* implicit, part of spin_unlock() */
```

Additional synchronisation points for ”spin until ready”

```
                CPU0
Thread0                     Thread1
                            preempt_disable(); /* implicit, part of spin_lock() */
                            active = 1
                            /* preempt by high priority Thread0 */
while (active)
    cpu_relax()
 /* spin for ever */
```

Additional synchronisation points for ”spin until ready”

- One side is waiting for the other in cpu_relax().
- Timers, del_timer_sync() / hrtimer_cancel().
- seqcount, read_seqcount_begin().
- softirq, had no soft(irq) user, tasklet_kill() waits for completion.
- A generic solutions, RT and debug or RT only.

LINUTRONIX

Unchanged primitives

- local_irq_disable(), preempt_disable(), bit spinlock, raw_spinlock_t
- get_cpu_var() or _ptr().
- Scope is usually per-CPU variables, HW registers, ...
- Scheduling is not possible.
- If needed, local_lock_t might fit as a replacement.

Lock ordering is from sleeping to may spin, to spinning

- 1) Sleeping locks (mutex, semaphore, ...)
- 2) Sleeping locks on PREEMPT_RT spinlock_t, rwlock_t, local_lock
- 3) Always spinning locks raw_spinlock_t and bit spinlocks
- Preempt and interrupt disabled sections (atomic sections) count as 3).
- local_irq_disable() + spin_lock() ≠ spin_lock_irq().

Which lock to use, raw_spinlock_t vs spinlock_t

- The low level core code uses raw_spinlock_t, everything in the hardirq context.
- Examples are interrupt controller code, time keeping or the scheduler which need to be accessed from hardirq sections.
- Once the hardirq context is left, using spinlock_t is fine. Everything else is handled in thread context.
- Driver's ISR is threaded.

Only on RT, preempt_disable() ⇒ spin_lock() with CONFIG_DEBUG_ATOMIC_SLEEP:

```
BUG: sleeping function called from invalid context at kernel/locking/...
in_atomic(): 1, irqs_disabled(): 0, non_block: 0, pid: 1, name: swapper/0
preempt_count: 1, expected: 0
RCU nest depth: 0, expected: 0
1 lock held by swapper/0/1:
 #0: ffffc (&l){+.+.}-{2:2}, at: function+0xbc/0x200
Preemption disabled at:
[<94004>] function+0xb4/0x200
CPU: 0 PID: 1 Comm: swapper/0 Not tainted 6.0.0-rc4-rt6
Hardware name: QEMU Standard PC
Call Trace:
 <TASK>
 dump_stack_lvl+0x4c/0x63
 rt_spin_lock+0x44/0xd0
```

Memory allocator (SLUB, page allocator) use sleeping locks

- Memory can not be allocated from atomic context, even with GFP_ATOMIC.
- Memory allocations were moved outside, avoided or the section was removed.
- All allocations within atomic() sections are problematic.
- There are no known allocation in atomic sections.

Also on !RT, raw_spin_lock() ⇒ spin_lock() with CONFIG_PROVE_RAW_LOCK_NESTING:

```
[ BUG: Invalid wait context ]
swapper/0/1 is trying to lock:
3bd8 (&port_lock_key){....}-{3:3}, at: serial8250_console_write+0x47a/0x4f0
other info that might help us debug this:
context-{5:5}
3 locks held by swapper/0/1:
 #0: 060 (rcu_tasks.cbs_gbl_lock){....}-{2:2}, at: cblist_init_generic+0x21/0x250
 #1: 080 (console_lock){+.+.}-{0:0}, at: _printk+0x63/0x7e
 #2: e60 (console_owner){....}-{0:0}, at: console_emit_next_record+0x111/0x300
stack backtrace:
CPU: 0 PID: 1 Comm: swapper/0 Not tainted 6.0.0-rc4
Hardware name: QEMU Standard PC
Call Trace:
```

The printk example

- Upon invocation the format string is evaluated.
- The result is saved in the ring buffer and printed on the console.
- The console driver is using spinlock_t locking.
- printk can be invoked from atomic context. What about the console driver? Can the lock become raw_spinlock_t instead of spinlock_t?
- The serial UART needs for one byte at 115200 baud 86.8us.
- A message with 20 characters takes over 1.5ms.
- The console would need remain spinlock_t.
- printk() needs to add content at calling time and the console driver needs to printed the message from somewhere else.

Introducing long latencies

- Inability to preempt the current running task/ code.
- Atomic context.
- Iterating over list with an unbounded amount if items.
- Long computations, cache flushing.
- Heavy contended locks, deep lock nesting.

LINUTRONIX

Example of wake up of 100 tasks via clock_nanosleep()

```
T: 1 (27230) P:95 I:250 C:  66128 Min:      2 Act:    3 Avg:    3 Max:      26
```

500:

```
T: 1 (27230) P:95 I:250 C: 324110 Min:      2 Act:   75 Avg:    3 Max:      75
```

1000:

```
T: 1 (27230) P:95 I:250 C: 488167 Min:      2 Act:  374 Avg:    3 Max:     374
```

5000:

```
T: 1 (27230) P:95 I:250 C: 568108 Min:      2 Act: 4113 Avg:    3 Max:    4113
```

Priority inheritance / PI boost

- Task A owns lock L. Task A is preempted and not on CPU.
- Task B wants lock L. Task B hands over priority to A.
- Task A can make progress and unlocks L.
- Elevated priority is reverted, B becomes L. releases L.

```
Task        Function
lockingA    tracing_mark_write: locked
lockingA    sched_wakeup: comm=lockingB prio=79 target_cpu=001
lockingA    sched_switch: prev_comm=lockingA prev_prio=89 prev_state=R+ ==>
   next_comm=lockingB next_prio=79

lockingB    tracing_mark_write: B needs that lock
lockingB    sched_pi_setprio: comm=lockingA oldprio=89 newprio=79
lockingB    sched_switch: prev_comm=lockingB prev_prio=79 prev_state=S ==> next_comm
   =lockingA next_prio=79

lockingA    tracing_mark_write: A about to unlock
lockingA    sched_pi_setprio: comm=lockingA oldprio=79 newprio=89
lockingA    sched_wakeup: comm=lockingB prio=79 target_cpu=001
lockingA    sched_switch: prev_comm=lockingA prev_prio=89 prev_state=R+ ==>
   next_comm=lockingB next_prio=79
```

Who can use PI boost

- Works for sleeping locks, which are based on rtmutex (spinlock_t, mutex_t,).
- To some degree with rwsem and rwlock_t.
- Doesn’t work with semaphore or percpu_rw_semaphore.
- User space pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);

Using softirq context

- softirq callbacks need to be fully synchronised against each other.
- NET_TX_SOFTIRQ with TASKLET_SOFTIRQ with TIMER_SOFTIRQ.
- Can be unrelated but may share resources. A per-CPU BKL.
- local_lock_t is used for synchronisation.
- A high prio needs to wait until all softirq is done with a PI-boost.

Dispatching work for later

- Using tasklet to complete the work.
  - Contributes to the softirq problem
  - No way of steering / preferring work except CPU pinning.
- Queuing a work_struct to complete the "work"
  - Ends up on a random kworker context.
  - No way of steering / preferring work.
  - Starting the worker on a remote CPU hurts "isolated" CPUs. Also affects NO_HZ_FULL users.

Summary

- Think about locking, is raw_spinlock_t needed?
- Be careful with local_irq_ / preempt_disable().
- High contended (global) spinning locks.
- New primitives, polling to complete, boosting.
- Carefull about adding more softirq callbacks.

Thank you for your attention

Special thanks to the Linux Foundation for supporting our efforts to bring PREEMPT_RT mainline.

`<bigeasy@linutronix.de>`