The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux

- D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole

Presented By:

Indradip Ghosh
RCU recap

- **Modus operandi**
  - writers perform updates by creating a new copy
  - readers read from the old copy
  - multiple versions (copies) enable readers and writers of the same data item to run concurrently

- **Safe, scalable and cheap memory reclamation**
  - rcu_read_lock() / unlock()
  - synchronize_rcu / call_rcu()
  - Minimizes the use of expensive memory barriers and read modify update instructions
Initial standings: RCU v.s. realtime OS primitives

- **2/3rds of the game is already won**
  - RCU does not suffer from priority inversion
  - RCU updaters are not dependent on reader completions

- **However, RCU has non-preemptible sections of code**
  - read critical sections are non-preemptible to support quiescent state based memory reclamation by updaters
  - !!Realtime OS primitives require a preemptible kernel
Issues in making RCU realtime

- Memory reclamation on voluntary context switch of reader threads may potentially lead to indefinite delays in memory reclamation especially if read critical sections are preempted.
- This would lead to huge memory footprints (not good for embedded or realtime scenarios) and degrade system performance (potential opportunity for DOS).
- All because rcu_read_lock/unlock calls do nothing in non-preemptible kernel.
So what’s the fix

- Finer grained quiescent state detection
  - Memory reclamation should not wait for all active readers to go through a context switch but only those who have a handle to the version being reclaimed
  - With preemption of read side critical sections grace period for memory reclamation would stretch but not so much if grace period is per version object
- Even then there is need to protect version collection from getting preempted and delayed too much (what is too much is a system specific issue)
- Writes should use asynchronous synchronization primitives: call_rcu() so they do not block.
RCU realtime adaptations

- Need for tracking threads that got preempted to precisely know when their grace periods end
  - Approach I - Single Global Counter: Very simple but not scalable
    - Atomically increment and decrement counter with calls to rcu_read_lock()/_unlock
    - Hence once this counter == 0, all threads are in a quiescent state.
    - However, as this tracks system wide critical reads, the situation worsens with increasing number of CPUs or even with few CPUs and multiple threads accessing different memory objects concurrently
    - Very high chance counter may never reach zero
RCU realtime adaptations

- **Approach II**: Having a two element array of counters:
  - Each `_lock/_unlock` atomically inc/dec matched pair of ‘current’ and ‘last’ cpu counter
  - Grace period start invokes swap of ‘current’ and ‘last’ fields, ‘last’ can only dec and ‘current’ counter can do both
  - Counter eventually reaches 0, marking end of grace period

Issues: Single pair of counter for each CPU leads to high memory contention leading to cache misses
RCU realtime adaptations

- **Approach of Choice:** 2N array of counters (N is # of threads)
  - Avoids high cache misses, atomic instructions, memory barriers as this data-structure is maintains per-thread state
  - However requires a state-machine to track when each CPU has seen change of global index
  - The global index can then be changed accordingly to indicate grace period endings
Boosting Priority when required-I

- Stopping indefinite delays for lower priority threads by boosting priority only when absolutely needed:
  - A pair of four element arrays, one labelled “Boosting | Old | Aging| New” and the other half being “Boosted for each corresponding entry as shown
Boosting Priority when required - II

- Each element of the 4 element array is a linked list of tasks in their respective states of being preempted 1, 2, 3 or 4 times.
  - Being in the Boosting state does not guarantee that it would be boosted for sure, that said it probably would be most of the time.
Multiple locking domains

- Reduced lock contention for writers by having each CPU maintains 4 locking domains by
  - Hence each task once enqueued is guaranteed to be protected by the same lock
  - Therefore \( \frac{1}{4} \) th lock contention per cpu
Increasing granularity for grace periods

- **Concept of RCU control block**
  - A single thread can block for excessive time if there is a single global set of grace periods
  - Therefore have a control implement a grace period for each related read side critical section
  - Reduces blocking time on synchronization primitives

Example:
```
idx = srcu_read_lock(&scb)
/* SRCU read-side critical section. */
srcu_read_unlock(&scb, idx)
```

The `scb` variable is the sleepable RCU control block, then a call to `synchronize_srcu(&scb)` blocks only until the end the respective `scb` variable grace period
Providing freshness

- Don’t read if it is stale
- Have a staleness indicator such as a flag
Table 1 Performance of RCU variants on 4-processor, 1.8-GHz Opteron 844 system

<table>
<thead>
<tr>
<th>RCU variant</th>
<th>Build</th>
<th>Reads (ns)</th>
<th>Updates</th>
<th>Quiescent state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 CPU</td>
<td>2 CPUs</td>
<td>3 CPUs</td>
</tr>
<tr>
<td>Rcu</td>
<td>S</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>rcu-bh</td>
<td>S</td>
<td>11.2</td>
<td>11.2</td>
<td>11.2</td>
</tr>
<tr>
<td>rcu-preempt</td>
<td>r</td>
<td>17.3</td>
<td>17.3</td>
<td>17.3</td>
</tr>
<tr>
<td>Sched-rt</td>
<td>R</td>
<td>27.4</td>
<td>27.4</td>
<td>27.4</td>
</tr>
<tr>
<td>rcu-rt</td>
<td>R</td>
<td>48.0</td>
<td>48.0</td>
<td>48.1</td>
</tr>
<tr>
<td>rcu-rt (new)</td>
<td>R</td>
<td>33.0</td>
<td>33.0</td>
<td>33.0</td>
</tr>
<tr>
<td>rcu-rt (nested)</td>
<td>R</td>
<td>8.4</td>
<td>8.4</td>
<td>8.4</td>
</tr>
<tr>
<td>srcu</td>
<td>A</td>
<td>10.0</td>
<td>10.0</td>
<td>10.1</td>
</tr>
</tbody>
</table>
Thankyou

- Questions?