Nested Locks in the Lock Implementation: The Real-Time Read-Write Semaphores on Linux

Daniel B. de Oliveira[1,2,3], Daniel Casini[3], Rômulo S. de Oliveira[2], Tommaso Cucinotta[3], Alessandro Biondi[3], and Giorgio Buttazzo[3]

[1]RHEL Platform/Real-time Team, Red Hat, Inc., Pisa, Italy.
[2]Department of Systems Automation, UFSC, Florianópolis, Brazil.
[3]RETIS Lab, Scuola Superiore Sant'Anna, Pisa, Italy.
Email: bristot@redhat.com, romulo.deoliveira@ufsc.br,
{daniel.casini,tommaso.cucinotta, alessandro.biondi, giorgio.buttazzo}@santannapisa.it

I. Introduction

Linux is a GP (General Purpose) OS (Operating System) that has gained many features of RT (Real-Time) OS over the last decade. For instance, nowadays Linux has a *fully preemptive* mode, and a deadline-oriented scheduler [1]. Although many of these features are part of the official Linux kernel, many of them are still part of an external patch set, the PREEMPT-RT [2]. The PREEMP-RT changes the locking methods of Linux to provide the control of unbounded priority inversion, by using the Priority Inheritance Protocol [3] on mutex, and bounds the activation delay for the highest priority task. Indeed, the *latency* is the main evaluation metric for the PREEMPT-RT Linux: for example, the Red Hat Enterprise Linux for Real-time [4] (based on PREEMPT-RT) shows a max latency of 150 μs on certified hardware. However, due to Linux's GPOS nature, RT Linux developers are challenged to provide the predictability required for a RTOS, while not causing regressions on the general purpose benchmarks. As a consequence, the implementation of some well known algorithms, like read/write semaphores, has to be done using approaches that were not well explored in academic papers, which is the case of read/write semaphores.

II. Read-write semaphores on Linux

On Linux, the read-write semaphores provide concurrent reader, exclusive writers for a given critical section. For example, since the memory mapping information of a process is read very often but during the process execution rarely changes, it is protected by a read-write semaphore.

The API of the read-write semaphores is composed by four main functions. Readers call `DOWN_READ()` before entering in the read-side, calling `UP_READ()` when leaving the read-side of the critical section. Writers should call `DOWN_WRITE()` before entering in the write-side of the critical section, calling `UP_WRITE()` when leaving. These functions take only one argument, which is a structure `rw_semahore`. The `rw_semaphore` structure is presented in Figure 1[1].

The `readers` variable is an atomic type that counts how many concurrent readers are inside the critical section. This variable is also used to set `READER` or `WRITER BIAS` flag, which are used to define if there are either readers or a writer in the critical section. Whenever a task should block in the semaphore, it will do by blocking in the real-time mutex `rtmutex` of the semaphore. The rt_mutex is defined as in Figure 2[1]:

```
1 struct rw_semaphore {
2   atomic_t                readers;
3   struct rt_mutex         rtmutex;
4 };
```

Fig. 1: Real-time Mutex structure

```
1 struct rt_mutex {
2   raw_spinlock_t          wait_lock;
3   struct rb_root_cached   waiters;
4   struct task_struct      *owner;
5   int                     save_state;
6 };
```

Fig. 2: Read-write Semaphore structure

In order to protect the fields of the rt_mutex struct from concurrent access, the spin lock `wait_lock` is used whenever internal fields of the mutex are modified. The `wait_lock` of the real-time mutex is also used to avoid two writers setting the `WRITE/READ BIAS` concurrently in the `rw_semaphore` structure.

The pseudo-code of each operation is the presented in Figure 3 and 4, respectively.

[1]Debug fields removed from structure's definition.

```
1: function UP_READ(rw_sem)
2:     rw_sem->readers–
3:     if rw_sem->readers == 0 then
4:         if a writer is holding the rw_sem->rtmutex then
5:             wake-up the writer
6:         end if
7:     end if
8: end function
9:
10: function DOWN_READ(rw_sem)
11:     if rw_sem->readers > 1 then
12:         rw_sem->readers++
13:         return /* enter in the critical section */
14:     end if
15:     while 1 do
16:         take rw_sem->rtmutex.wait_lock /* might block busy */
17:         if WRITER BIAS is not set then
18:             rw_sem->readers++
19:             release rw_sem->rtmutex.wait_lock
20:             return /* enter in the critical section */
21:         end if
22:         release rw_sem->rtmutex.wait_lock
23:         take rw_sem->rt_mutex /* might block suspended */
24:         release the rw_sem->rt_mutex
25:     end while
26:     return /* enter in the critical section */
27: end function
```

Fig. 3: Read-side operations

```
1: function UP_WRITE(rw_sem)
2:     clear WRITER BIAS
3:     set READER BIAS
4:     release sem->rtmutex
5: end function
6:
7: function DOWN_WRITE(rw_sem)
8:     take rw_sem->rtmutex /* might block suspended */
9:     clear READER BIAS
10:     if rw_sem->readers != 0 then
11:         suspend waiting for the last reader
12:     end if
13:     while 1 do
14:         take sem->rtmutex->wait_lock /* might block busy */
15:         if sem->readers == 0 then
16:             set WRITER BIAS
17:             release rw_sem->rtmutex->wait_lock
18:             return /* enter in the critical section */
19:         end if
20:         release rw_sem->rtmutex->wait_lock.
21:         suspend waiting for the last reader
22:     end while
23:     return
24: end function
```

Fig. 4: Write-side operations

III. Open Problems

Considering our example, when `DOWN_WRITE()` is called the task that is trying to acquire the read/write semaphore for writing has to lock two nested resources, a regular *mutex* (acquired at line 8, Figure 4) and a *spin lock* (acquired at line 14, Figure 4), thus creating a **heterogeneous nested lock** (e.g., a suspension-based lock with a nested spin-based lock or vice-versa). This case study taken from the Linux kernel highlights two open issues. The first one concerns the need for developing analysis techniques for (possibly heterogeneous) nested locks. To the best of our knowledge, only few works on shared-memory multiprocessor synchronization targeted nested critical sections. Two notable examples are the work by Biondi et al. [5], in which a graph abstraction is introduced to derive a fine-grained analysis (i.e., not based on asymptotic bounds) for FIFO *non-preemptive* spin locks, and the one by Ward and Anderson [6], in which the *real-time nested locking protocol* (RNLP) is proposed, with the related *asymptotic* analysis. Later, Nemitz et al. [7] proposed an optimization for the average-case of RNLP. However, none of these works are explicitly designed to deal with nested heterogeneous locks. For instance, although RNLP provides different resource satisfaction mechanisms (RSMs) for suspension-based mutexes and spin locks, the behavior when multiple RSMs are used in conjunction is not discussed, and most of the asymptotic bounds are provided considering RSMs separately. Deriving a novel RSM explicitly designed for working in presence of heterogeneous types of nested resources may be an interesting starting point for future work. Also, future research could evaluate the possibility of extending the graph abstraction proposed by Biondi et al. [5] to allow fine-grained analysis for nested heterogeneous locks. The second open problem concerns the design of specialized analysis techniques accounting for specific implementations of complex types of locks (e.g., the aforementioned read/write lock in Linux). Considering the problem previously presented for the `DOWN_WRITE` function, an implementation-aware analysis would account for the contention on the heterogeneous nested critical section, considering it when a blocking-bound for the reader/writer semaphore is derived. The analyses for reader/writer semaphores that have already been proposed (e.g., the protocol proposed by Brandenburg and Anderson [8], or R/W RNLP [9], a variant of RNLP conceived to deal with nested, spin-based, read/write locks) could be integrated with implementation-specific aspects. The availability of blocking-bounds conceived considering the specific implementation adopted in the Linux kernel may help it to be more suitable for real-time contexts. Finally, a third open research area consists in finding more efficient locking protocols (with the related implementation), accounting both for general purpose benchmark performances (i.e., average-case behavior, needed by the GPOS nature of Linux) and predictability.

References

[1] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the Linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016.

[2] D. B. de Oliveira and R. S. de Oliveira, "Timing analysis of the PREEMPT RT linux kernel," *Softw., Pract. Exper.*, vol. 46, no. 6, pp. 789–819, 2016.

[3] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, Sep. 1990.

[4] Red Hat, Inc., "Red Hat Enterprise Linux for Real Time," Available at: https://www.redhat.com/it/resources/red-hat-enterprise-linux-real-time [last accessed 28 March 2017].

[5] A. Biondi, A. Weider, and B. Brandenburg, "A blocking bound for nested fifo spin locks," in *Real-Time Systems Symposium (RTSS)*, 2016, pp. 291–302.

[6] B. C. Ward and J. H. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, 2012, pp. 223–232.

[7] C. E. Nemitz, T. Amert, and J. H. Anderson, "Real-time multiprocessor locks with nesting: Optimizing the common case," in *Proceedings of the 25th International Conference on Real-Time and Network Systems (RTNS 2017)*, 2017.

[8] B. B. Brandenburg and J. H. Anderson, "Reader-writer synchronization for shared-memory multiprocessor real-time systems," in *2009 21st Euromicro Conference on Real-Time Systems*, July 2009, pp. 184–193.

[9] B. C. Ward and J. H. Anderson, "Multi-resource real-time reader/writer locks for multiprocessors," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 177–186.