Real-time Scheduling Open Problems Seminar (RTSOPS 2018)

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

Daniel B. de Oliveira[1,2,3], Daniel Casini[2], Rômulo S. de Oliveira[3], Tommaso Cucinotta[2], Alessandro Biondi[2], and Giorgio Buttazzo[2]
Email: bristot@redhat.com, romulo.deoliveira@ufsc.br,
{ daniel.casini,tommaso.cucinotta, alessandro.biondi, giorgio.buttazzo } @santannapisa.it

Real-time Linux

- Linux is a GPOS with RTOS ambitions
    - Preemptive
    - FIFO (TLFP) and Deadline (JLFP) schedulers
    - User-space locks with PIP & PCP

- PREEMPT-RT improves Linux's predictability by:
    - Making system as preemptive/schedulable as possible
    - Bounding priority inversions using PIP on kernel locks
    - Max (activation delay?) latency of 150 µs

Real-time Linux howevers

Due to Linux's GPOS nature, RT Linux developers are challenged to provide the predictability required for an RTOS, while not causing regressions on the general purpose benchmarks.

In practice it means

- Developers cannot cause performance regressions
    - Throughput:
        - Two implementations: RT and non RT
            - A newer algorithm cannot cause - much - regression compared to the older one
    - Predictability:
        - Cannot increase the *latency*
            - e.g, cannot disable the preemption for a long period

Consequences...

As a consequence, the implementation of some well known algorithms, like read/write semaphores, has been done using approaches that were exhaustively explored in academic papers.

IOW: it works, but...

Read/write semaphores on Linux

Read-side

```
down_read(&rw_semapore) {
  /* enters in the read-side */
}

/*
 * Read-side critical section
 * Parallel with other readers
 * No writers
 */

up_read(&rw_semapore) {
  /* leaves the read-side */
}
```

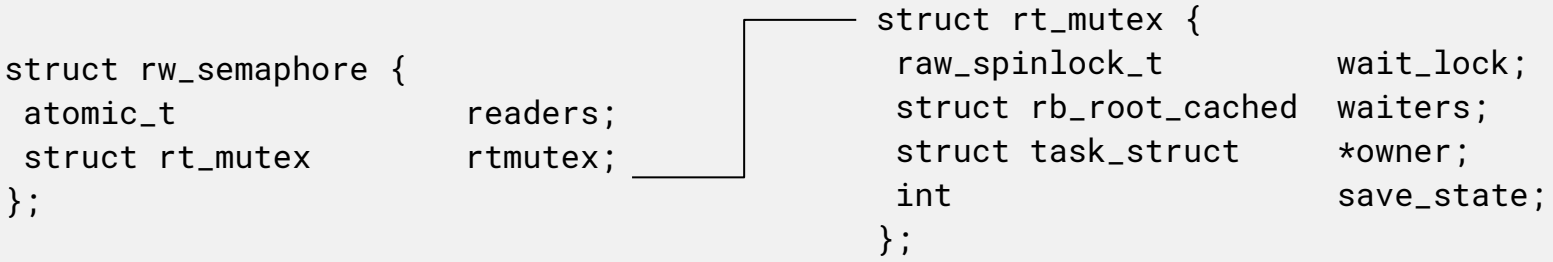
Write-side

```
down_write(&rw_semapore) {
  /* enters in the write-side */
}

/*
 * Write-side critical section
 * Exclusive access
 */


up_write(&rw_semapore) {
  /* leaves the write-side */
}
```

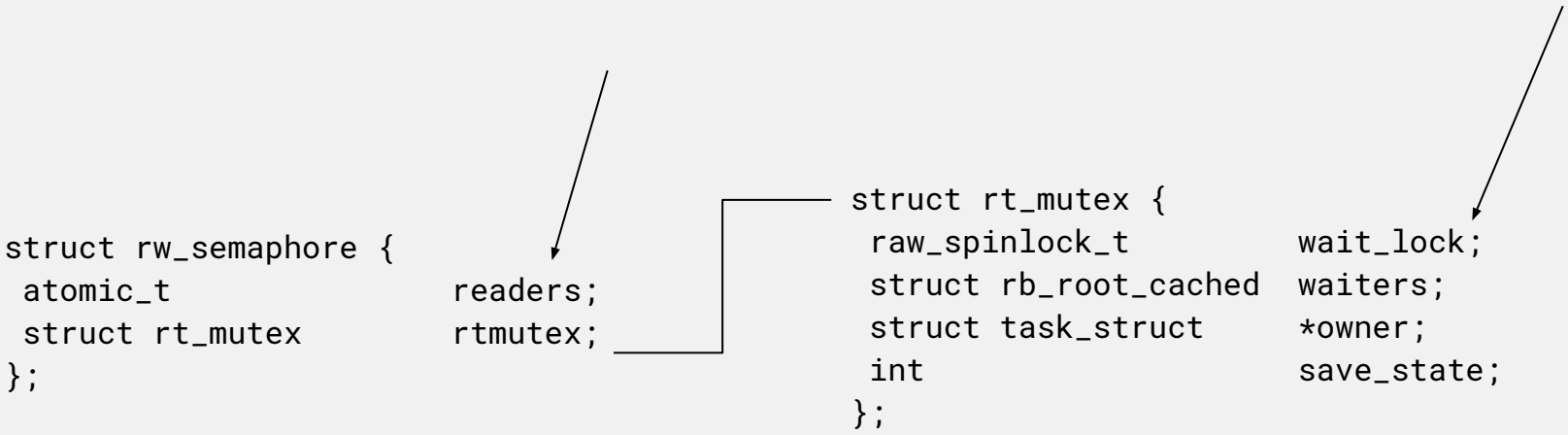
Read/write semaphores on Linux

```
struct rw_semaphore {
 atomic_t               readers;
 struct rt_mutex        rtmutex;
};
```

Read/write semaphores on Linux



Read/write semaphores on Linux



Concurrent down operations

Atomic operation

```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

Mutex: ...
Spin lock: ...

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations

Atomic operation

Mutex: held
Spin lock: ...

```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations

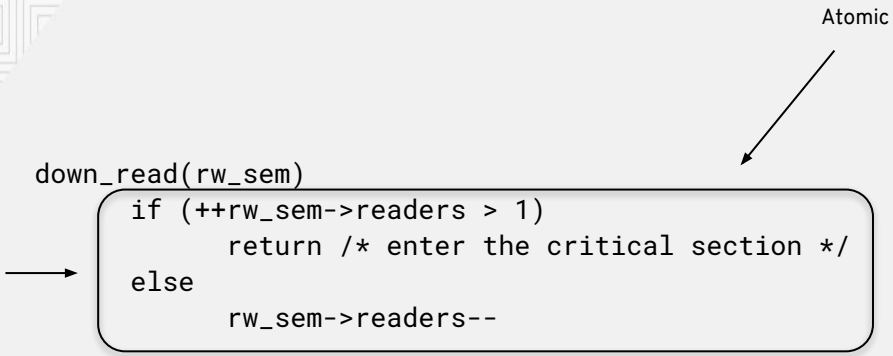
Atomic operation

```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

Mutex: held
Spin lock: ...

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations




```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

Mutex: held
Spin lock: ...

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations

Mutex: held
Spin lock: ...

```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations

Mutex: ...
Spin lock: held

```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

Mutex: held
Spin lock: block

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations

Mutex: ...
Spin lock: held

```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

Mutex: held
Spin lock: block

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations

Mutex: ...
Spin lock: held

```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

Mutex: held
Spin lock: block

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations

Mutex: ...
Spin lock: held

```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

Mutex: held
Spin lock: block

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations

Mutex: ...
Spin lock: ...

```
down_read(rw_sem)
      if (++rw_sem->readers > 1)
            return /* enter the critical section */
      else
            rw_sem->readers--

      take rw_sem->rtmutex.wait_lock

      if (WRITER BIAS is not set) {
            rw_sem->readers++
            release rw_sem->rtmutex.wait_lock
            return /* enter in the critical section */
      }
      release rw_sem->rtmutex.wait_lock
      take rw_sem->rt_mutex
      rw_sem->readers++
      release the rw_sem->rt_mutex
      return /* enter in the critical section */
}
```

Mutex: held
Spin lock: held

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
            if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Concurrent down operations

A task taking a write lock,
With a nested mutex
With a nested spin-lock.

Mutex: held
Spin lock: held

```
down_write(rw_sem) {
      take rw_sem->rtmutex
      clear READER BIAS
      if (rw_sem->readers != 0)
            suspend waiting for the last reader
      while(1) {
            take sem->rtmutex->wait_lock
      ——→   if (sem->readers == 0) {
                  set WRITER BIAS
                  release rw_sem->rtmutex->wait_lock
                  return
            }
            release rw_sem->rtmutex->wait_lock.
            suspend waiting for the last reader
      }
      return
}
```

Open Issues

1) Implementing in Linux state-of-the-art protocols for heterogeneous nested locks and developing novel analysis techniques

Shared memory nested critical sections

- 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.
    - Proposed real-time nested locking protocol (RNLP), with the related *asymptotic* analysis.

- B. C. Ward and J. H. Anderson, "**Fine-grained multiprocessor real-time locking with improved blocking,**" in Proceedings of the 21st International Conference on Real-Time Networks and Systems, ser. RTNS '13, 2013.
    - Conceived to deal with heterogeneous nested critical sections: Block + Spinning (short-on-long)

Shared memory nested critical sections

- 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
    - nested read/write spin lock with **fast path**!

- 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.
    - Graph abstraction is introduced to derive a fine-grained analysis, not based on asymptotic bounds for FIFO *non-preemptive* spin locks.

Linux's locking needs

- Sleeping:
    - Nested blocking (rt mutexes)
    - Nested read/write (rw semaphores)
- Busy-wait:
    - Nested read/write spin (rw lock)
    - Nested spinlock (raw spin lock)

- Fast path is important
- Schedulers: TLFP, JLFP & IRQ/NMI
- Arbitrary affinities

2) The design of specialized analysis techniques accounting for specific implementations of complex types of locks (e.g., the aforementioned read/write lock in Linux).

3) finding more efficient locking protocols, accounting for both general purpose benchmark performance (i.e., average-case behavior, needed by the GPOS nature of Linux) and predictability.

Questions?

Thanks!