SYNOPSYS®

Ticket spinlocks in Zephyr RTOS: what, where & why

Evgenii Paltsev

Zephyr Developer Summit, 2024

Evgenii Paltsev

- Sr software engineer in Synopsys
- Develop, port and maintain OSS projects for Synopsys ARC and RISC-V processors architectures
- Maintainer of ARC architecture in Zephyr RTOS
- Main focus - Zephyr RTOS, Linux kernel, U-Boot

Rationale

- Multicore platform bring up – ARCv3, 8 / 12 cores
- Necessity of running something
  - SMP capable
  - allows to load the system
  - simple (simpler than Linux)

Rationale

Zephyr is great candidate:

- Mature SMP support – in tree since v1.11
- POSIX subsystem – allows to run existing benchmarks / stress tests with minimal changes
- We plan to support Zephyr anyway

Co-development

- The HW platform was not yet ready – so tests were prepared and run in simulation
- Synopsys nSIM simulator was used

Issue – some stress tests fail

What to blame?

- SMP implementation for our architecture
- POSIX subsystem in Zephyr
- Stress tests itself
- Simulator
- Toolchain
- Universe
- ….

Issue – some stress tests fail

- We have a livelock!
- It was possible due to combination of how simulation works and how Zephyr spinlocks are implemented

Was this issue unic?

- At the same moment other team was investigating similar issue in Zephyr (but with HW platform)
- GitHub: zephyrproject-rtos/zephyr#61541

- This all ended in addition of spinlock fairness tests and alternative spinlock implementation to Zephyr

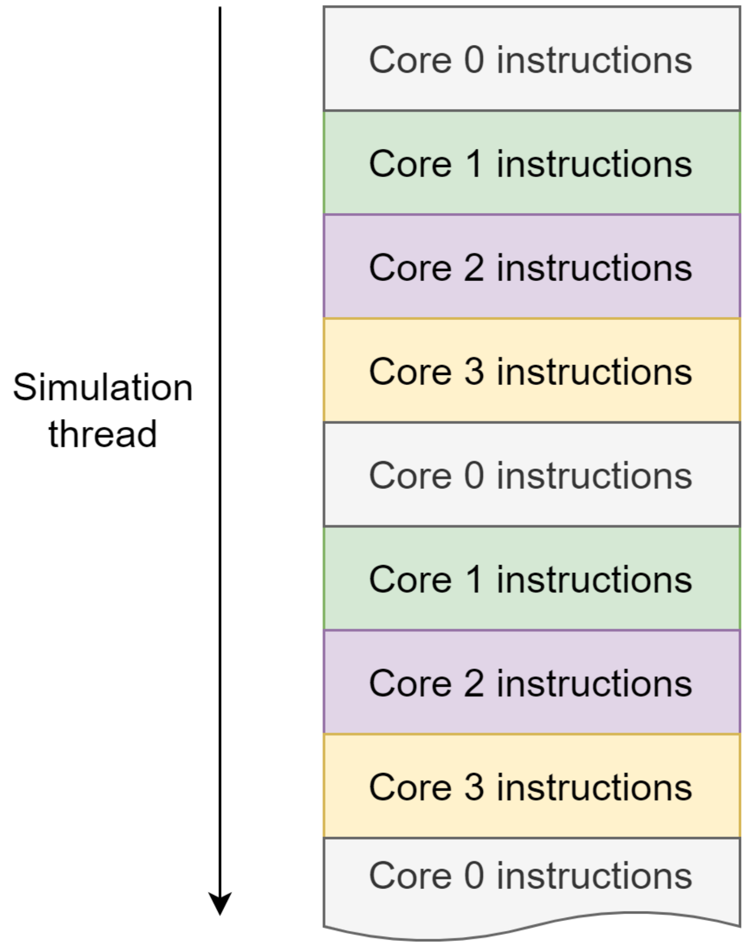
Default Zephyr spinlock implementation

- Introduced with SMP addition in v1.11
- Unchanged since then
- No fairness guarantee

```
struct k_spinlock {
    atomic_t locked;
};

k_spinlock_key_t k_spin_lock(struct k_spinlock *l) {
    k_spinlock_key_t key = arch_irq_lock();

    while (!atomic_cas(&l->locked, 0, 1)) { }
    return key;
}

void k_spin_unlock(struct k_spinlock *l, k_spinlock_key_t key) {
    atomic_clear(&l->locked);
    arch_irq_unlock(key);
}
```

Our simulator (nSIM)

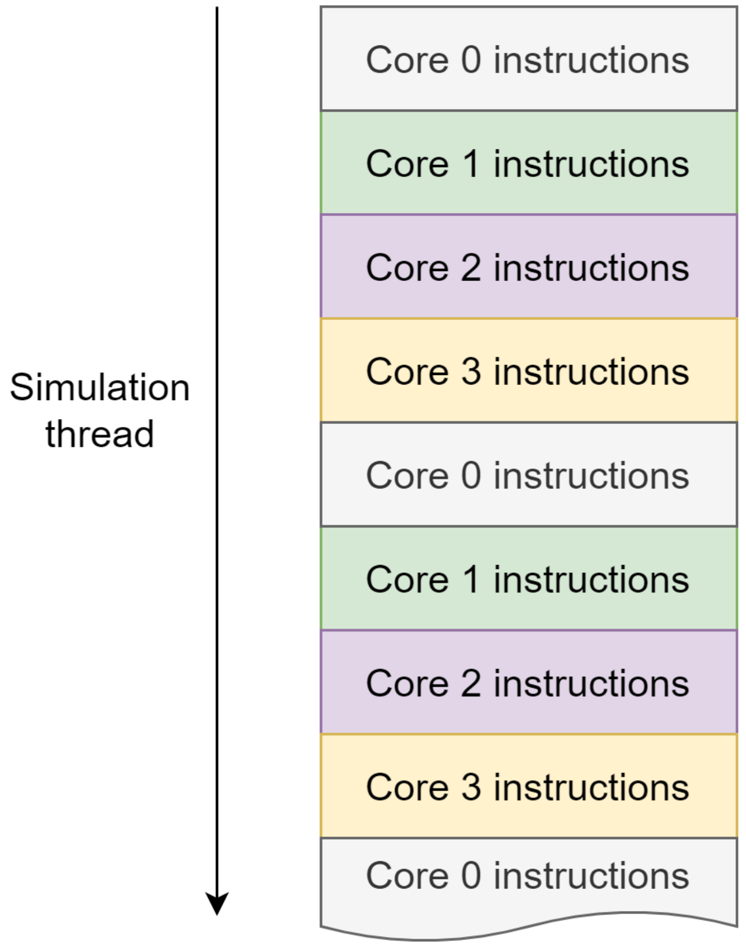
- Sequentially execute blocks of instructions from each core in single simulation thread
- Simulation is deterministic:
  - Number of instructions in a block is fixed
  - No external events / interrupts due to lack of peripherals in model used



Issue – our simulator (nSIM)

- We access spinlock from multiple cores
- Spinlock is locked & unlocked multiple times on Core X
- Simulator always switch to another core instructions when spinlock on Core X is locked (coincidence)
- Other cores aren't able to grab spinlock at all

- Not reproduced stably – minor code changes may mask issue



Spinlock fairness test

- Test case in tests/kernel/spinlock
- Grab – hold – release spinlock in a loop on all cores
- Collect per-core attempts statistics

```
struct k_spinlock lock;

void test_thread(...) {
    int key = arch_irq_lock();
    synchronize_cores();

    do {
        k_spinlock_key_t s_key = k_spin_lock(&lock);

        spinlock_grabbed[core_id]++;
        busy_work();

        k_spin_unlock(&lock, s_key);
    } while (!finished);

    arch_irq_unlock(key);
}
```

Fairness test - nSIM

- nSIM simulator, ARCv2 architecture, 4 cores (modified nsim_hs_smp config)

- Result:

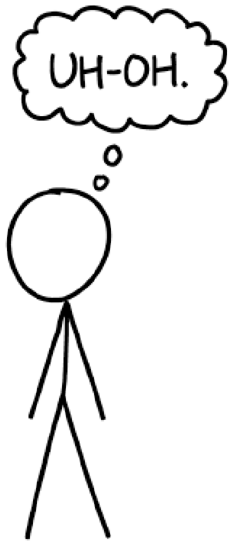
```
CPU0 acquired spinlock 981 times, expected 1000
CPU1 acquired spinlock 1019 times, expected 1000
CPU2 acquired spinlock 990 times, expected 1000
CPU3 acquired spinlock 1010 times, expected 1000
```

What about real SMP HW?

- HW board, ARCv2 architecture, 4 cores (hsdk config)

- What results would we get? Ideal:

```
CPU0 acquired spinlock 1000 times, expected 1000
CPU1 acquired spinlock 1000 times, expected 1000
CPU2 acquired spinlock 1000 times, expected 1000
CPU3 acquired spinlock 1000 times, expected 1000
```



What about real SMP HW?

- HW board, ARCv2 architecture, 4 cores (hsdk config)
- What results would we get? Probable:

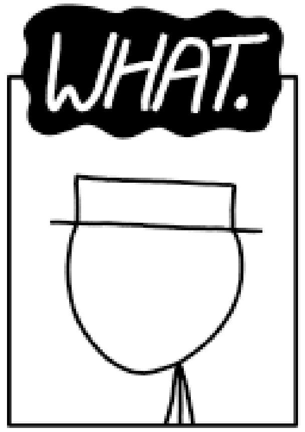
```
CPU0 acquired spinlock 999 times, expected 1000
CPU1 acquired spinlock 1001 times, expected 1000
CPU2 acquired spinlock 999 times, expected 1000
CPU3 acquired spinlock 1001 times, expected 1000
```

What about real SMP HW?

- HW board, ARCv2 architecture, 4 cores (hsdk config)

- Actual results:

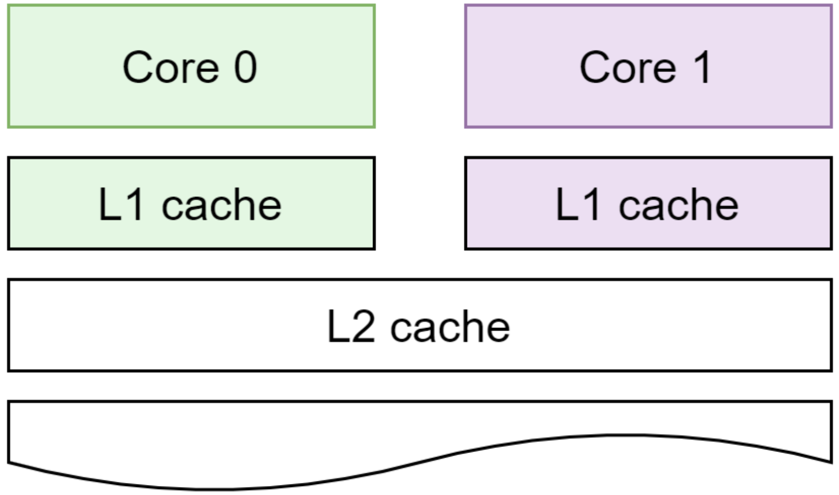
```
CPU0 acquired spinlock 0 times, expected 1000
CPU1 acquired spinlock 3997 times, expected 1000
CPU2 acquired spinlock 1 times, expected 1000
CPU3 acquired spinlock 2 times, expected 1000
```



What about real SMP HW?

How does this happen?

- CAS – Compare And Swap atomic primitive
- Access '*locked*' var in loop from multiple cores simultaneously
- Our HW has private L1 caches for each core
- Core which owns cache line (in its L1 D$) with '*locked*' var has more chances for CAS to succeed



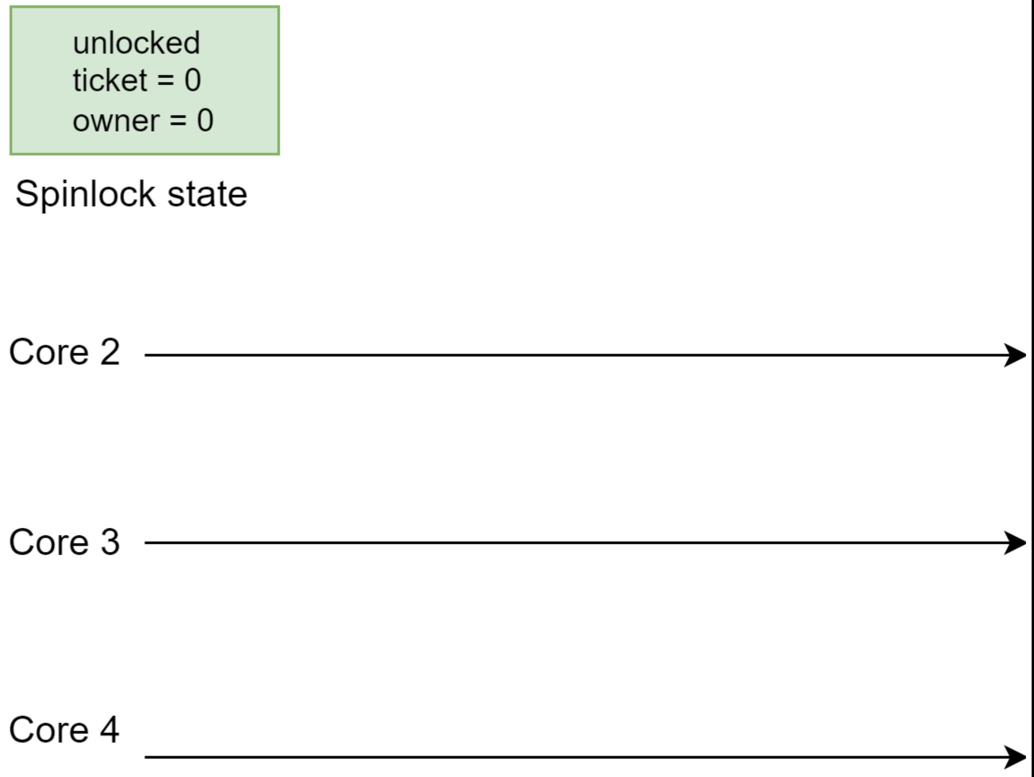
```
k_spinlock_key_t k_spin_lock(k_spinlock *l) {
    ...
    while (!atomic_cas(&l->locked, 0, 1)) {
    }
    ...
}
```

Ticket spinlock

- Known algorithm – added to Linux kernel ~ 20 years ago

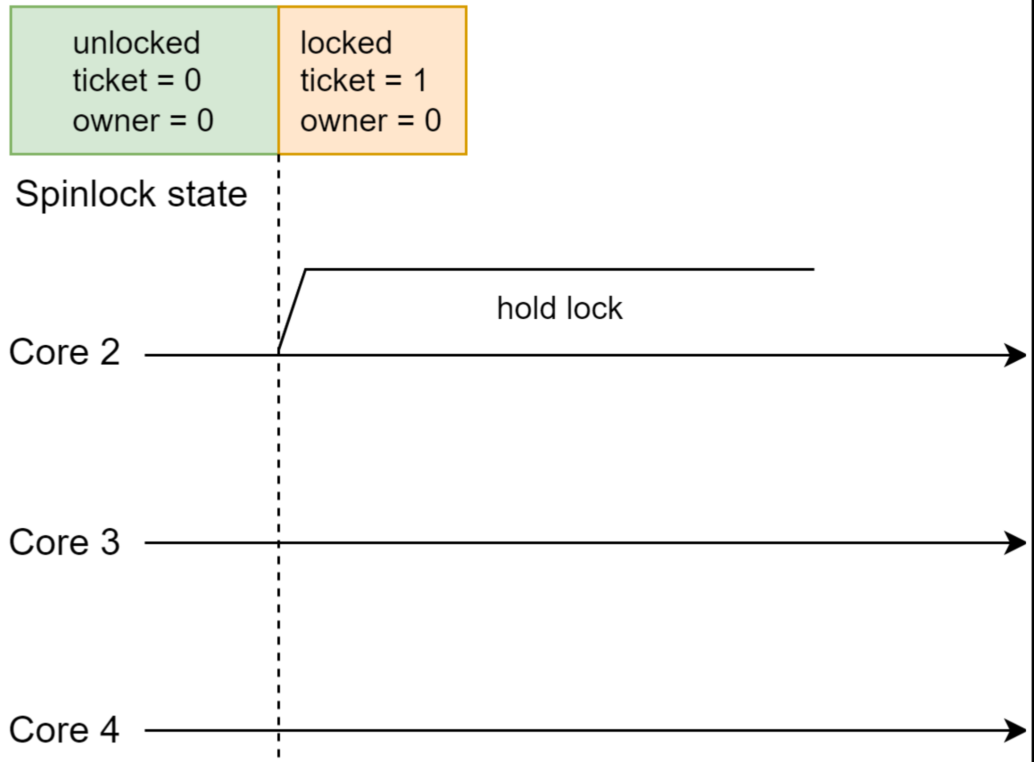
```
struct k_spinlock {
    atomic_t owner;
    atomic_t ticket;
};

k_spinlock_key_t k_spin_lock(struct k_spinlock *l) {
    k_spinlock_key_t key = arch_irq_lock();

    atomic_val_t ticket = atomic_inc(&l->ticket);
    while (atomic_get(&l->owner) != ticket) { }

    return key;
}

void k_spin_unlock(struct k_spinlock *l, k_spinlock_key_t key) {
    atomic_inc(&l->owner);
    arch_irq_unlock(key);
}
```

Ticket spinlock



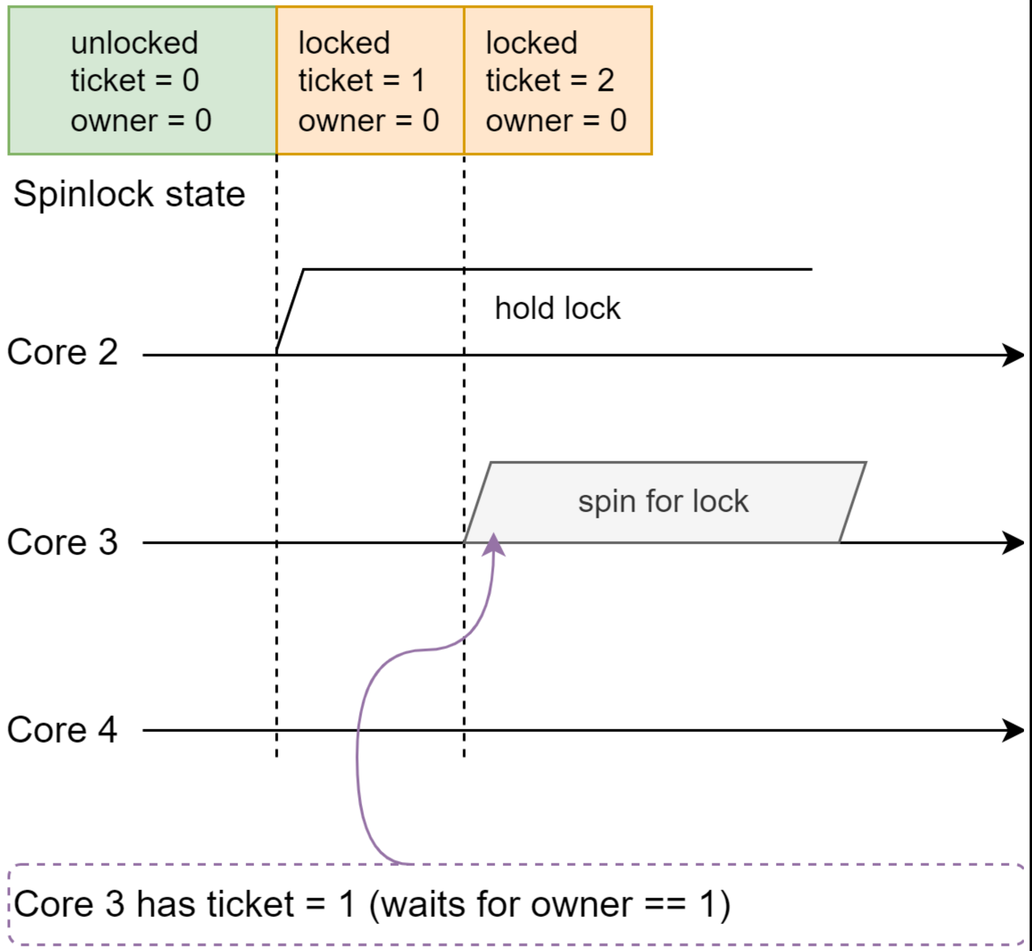
```
struct k_spinlock {
    atomic_t owner;
    atomic_t ticket;
};

k_spinlock_key_t k_spin_lock(struct k_spinlock *l) {
    k_spinlock_key_t key = arch_irq_lock();

    atomic_val_t ticket = atomic_inc(&l->ticket);
    while (atomic_get(&l->owner) != ticket) { }

    return key;
}

void k_spin_unlock(struct k_spinlock *l, k_spinlock_key_t key) {
    atomic_inc(&l->owner);
    arch_irq_unlock(key);
}
```

Ticket spinlock



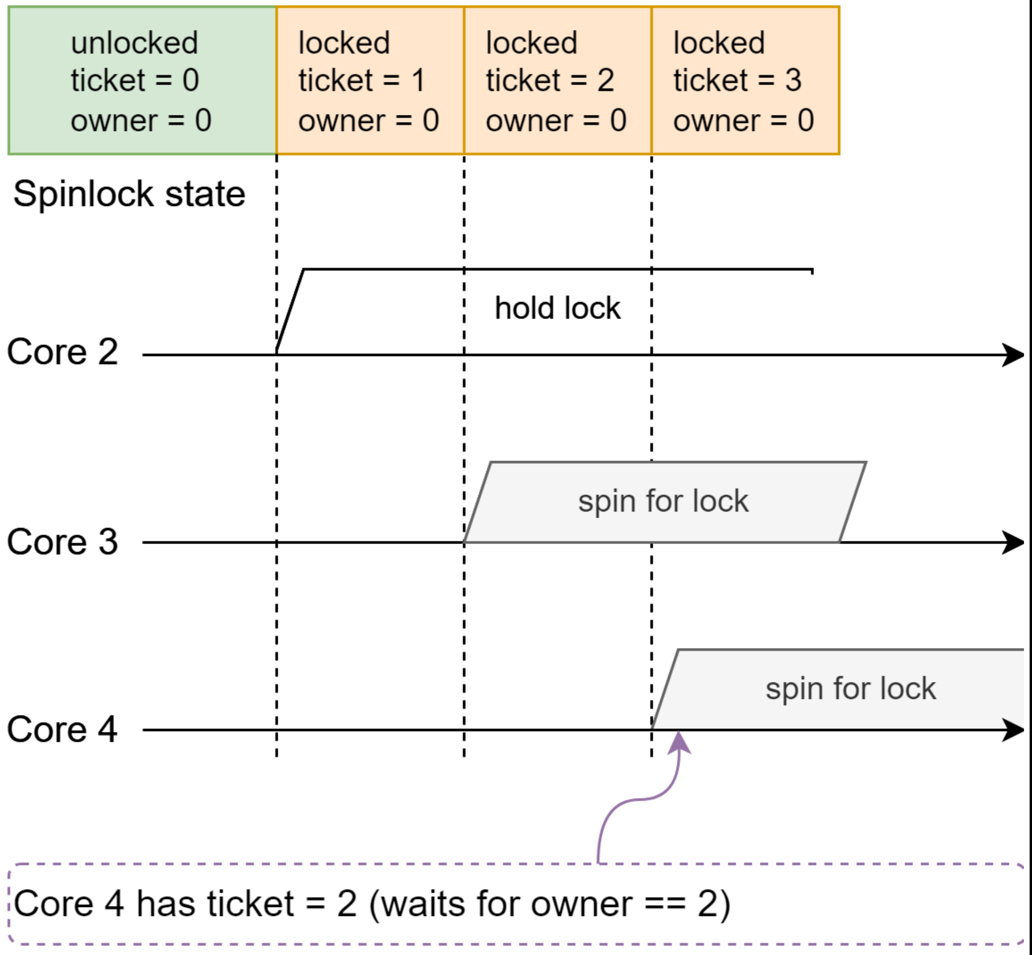
```
struct k_spinlock {
    atomic_t owner;
    atomic_t ticket;
};

k_spinlock_key_t k_spin_lock(struct k_spinlock *l) {
    k_spinlock_key_t key = arch_irq_lock();

    atomic_val_t ticket = atomic_inc(&l->ticket);
    while (atomic_get(&l->owner) != ticket) { }

    return key;
}

void k_spin_unlock(struct k_spinlock *l, k_spinlock_key_t key) {
    atomic_inc(&l->owner);
    arch_irq_unlock(key);
}
```

Ticket spinlock



```c
struct k_spinlock {
    atomic_t owner;
    atomic_t ticket;
};

k_spinlock_key_t k_spin_lock(struct k_spinlock *l) {
    k_spinlock_key_t key = arch_irq_lock();

    atomic_val_t ticket = atomic_inc(&l->ticket);
    while (atomic_get(&l->owner) != ticket) { }

    return key;
}

void k_spin_unlock(struct k_spinlock *l, k_spinlock_key_t key) {
    atomic_inc(&l->owner);
    arch_irq_unlock(key);
}
```

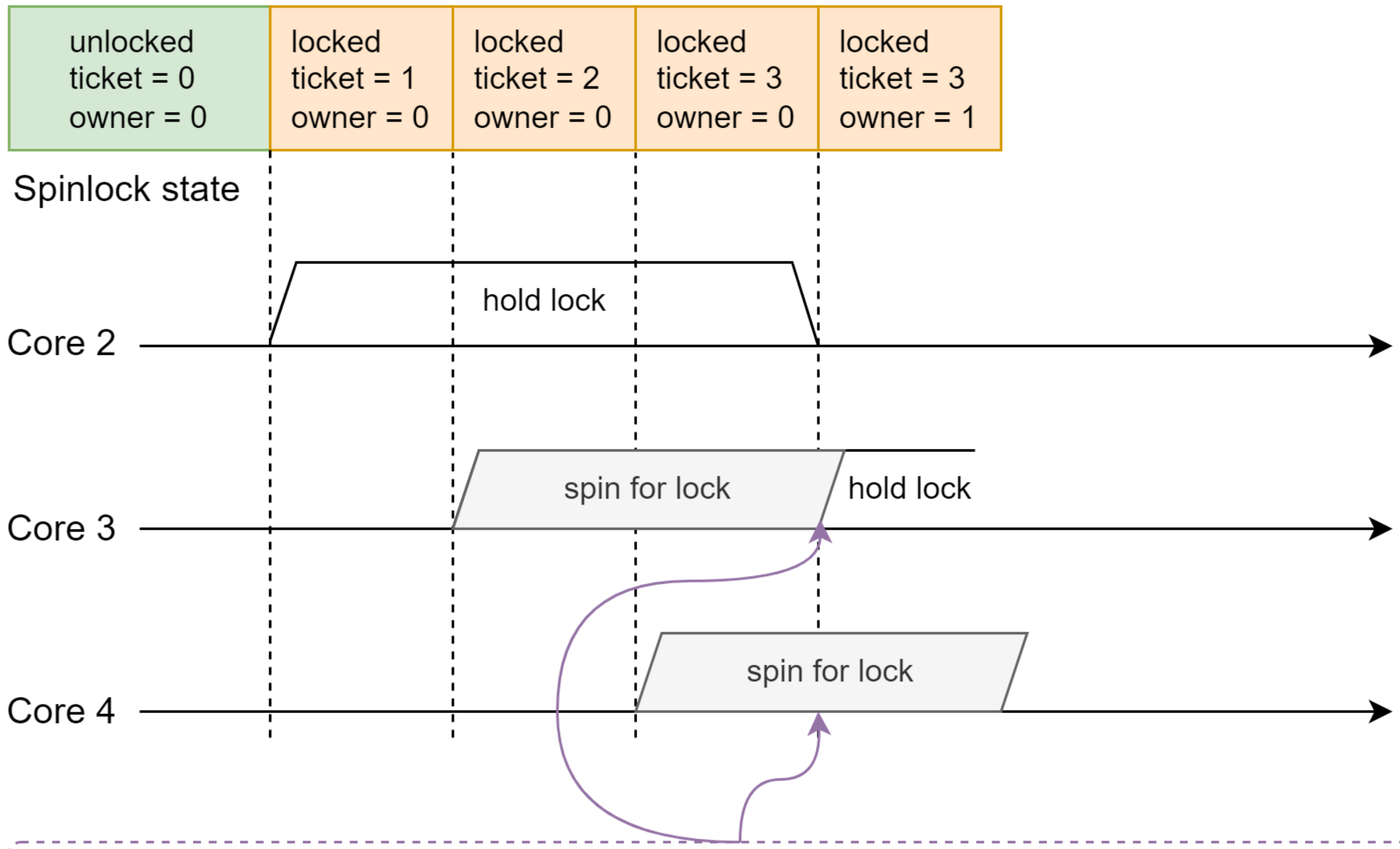
Ticket spinlock



```
struct k_spinlock {
    atomic_t owner;
    atomic_t ticket;
};

k_spinlock_key_t k_spin_lock(struct k_spinlock *l) {
    k_spinlock_key_t key = arch_irq_lock();

    atomic_val_t ticket = atomic_inc(&l->ticket);
    while (atomic_get(&l->owner) != ticket) { }

    return key;
}

void k_spin_unlock(struct k_spinlock *l, k_spinlock_key_t key) {
    atomic_inc(&l->owner);
    arch_irq_unlock(key);
}
```

© 2024 Synopsys, Inc.

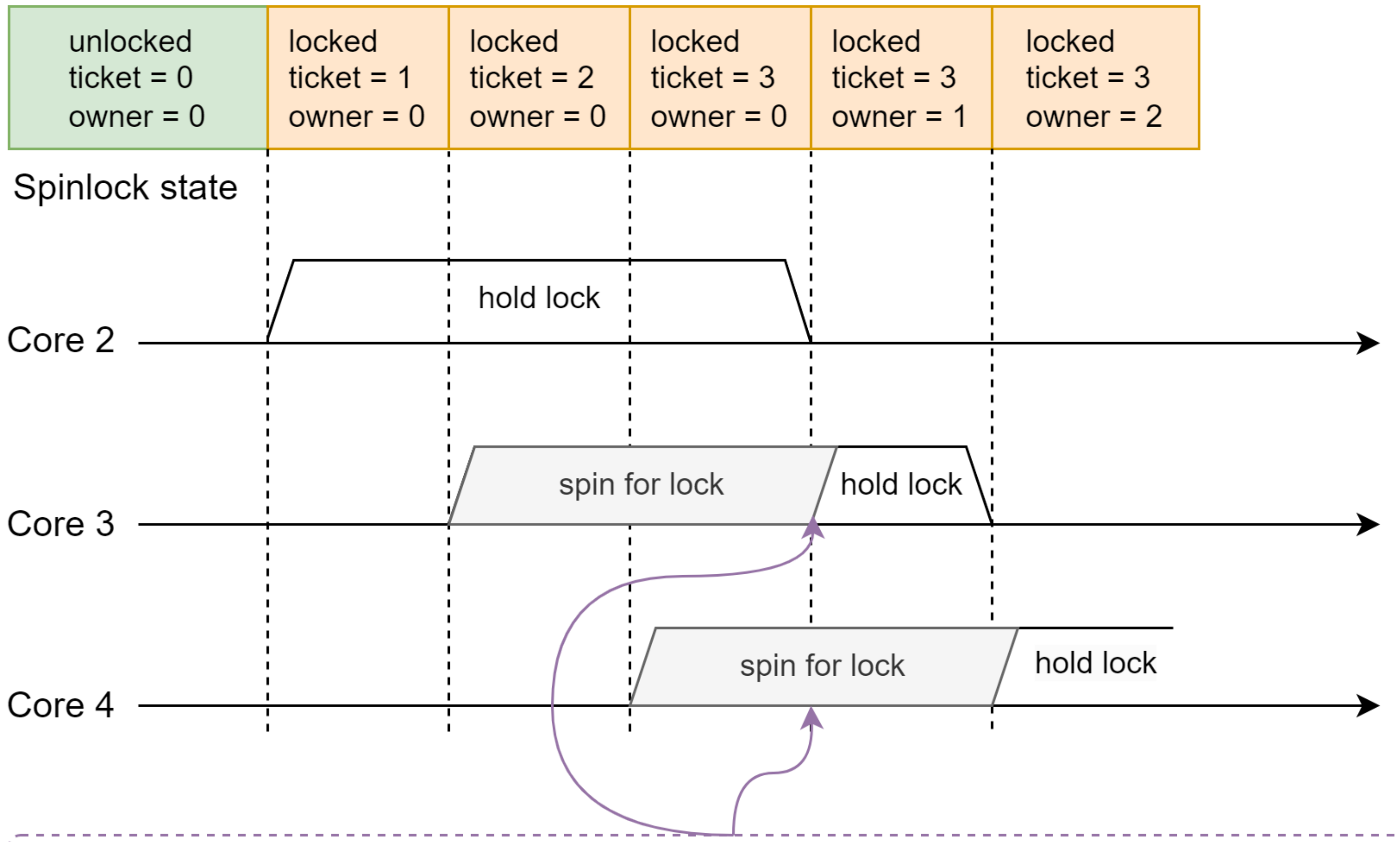
Ticket spinlock



Core 3 has ticket = 1 (waits for owner == 1), Core 4 has ticket = 2 (waits for owner == 2)

```
struct k_spinlock {
    atomic_t owner;
    atomic_t ticket;
};

k_spin_lock(k_spinlock_t *l) {
    key = arch_irq_lock();

    ticket = atomic_inc(&l->ticket);
    while (atomic_get(&l->owner) != ticket)
    { }

    return key;
}

k_spin_unlock(k_spinlock_t *l, key_t key) {
    atomic_inc(&l->owner);
    arch_irq_unlock(key);
}
```

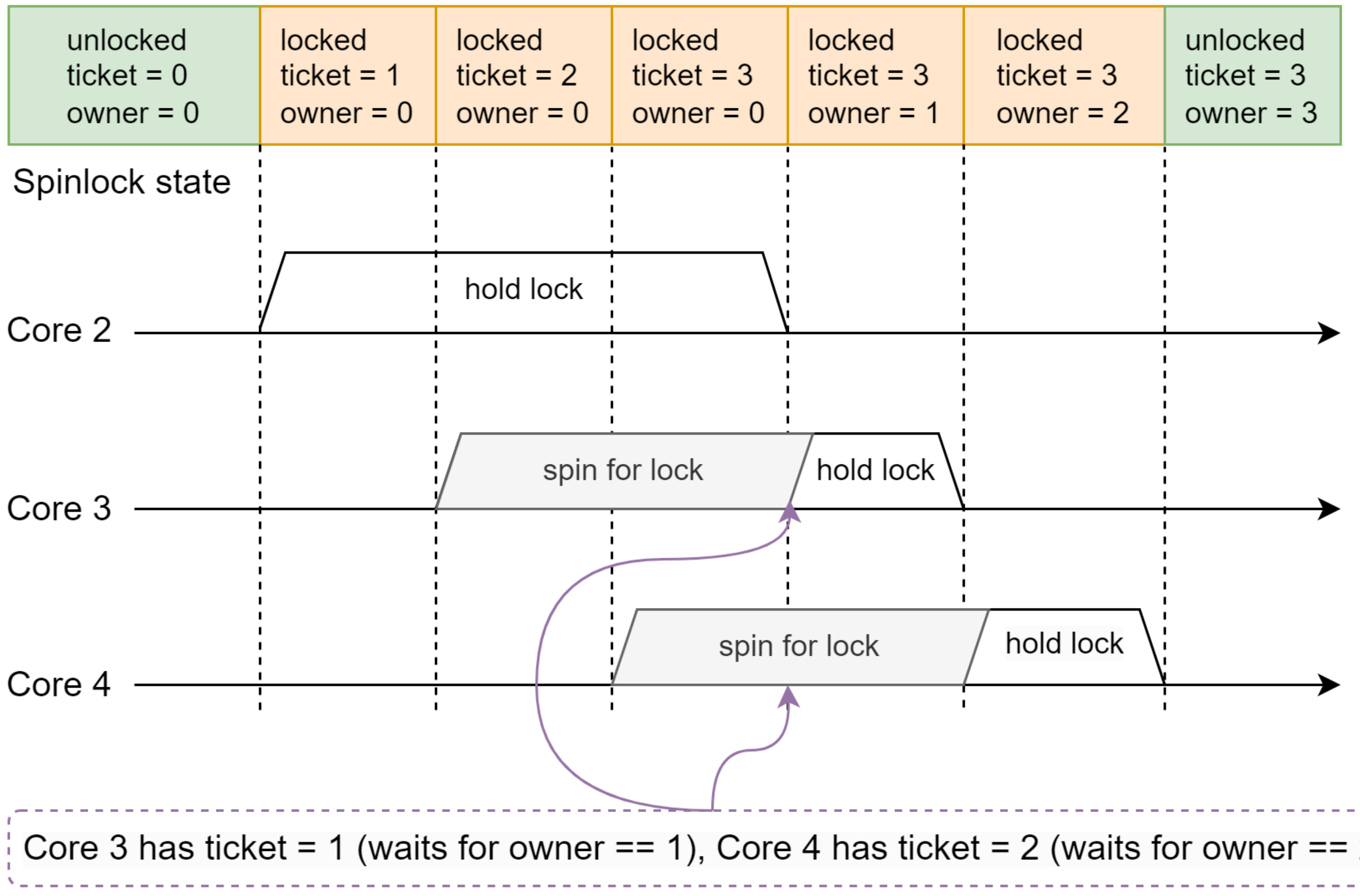
Ticket spinlock


unlocked ticket = 0 owner = 0
locked ticket = 1 owner = 0
locked ticket = 2 owner = 0
locked ticket = 3 owner = 0
locked ticket = 3 owner = 1
locked ticket = 3 owner = 2
Spinlock state
hold lock
Core 2
spin for lock
hold lock
Core 3
spin for lock
hold lock
Core 4

Core 3 has ticket = 1 (waits for owner == 1), Core 4 has ticket = 2 (waits for owner == 2)

```
struct k_spinlock {
    atomic_t owner;
    atomic_t ticket;
};

k_spin_lock(k_spinlock_t *l) {
    key = arch_irq_lock();

    ticket = atomic_inc(&l->ticket);
    while (atomic_get(&l->owner) != ticket)
    { }

    return key;
}

k_spin_unlock(k_spinlock_t *l, key_t key) {
    atomic_inc(&l->owner);
    arch_irq_unlock(key);
}
```

Ticket spinlock



Core 3 has ticket = 1 (waits for owner == 1), Core 4 has ticket = 2 (waits for owner == 2)

```
struct k_spinlock {
    atomic_t owner;
    atomic_t ticket;
};

k_spin_lock(k_spinlock_t *l) {
    key = arch_irq_lock();

    ticket = atomic_inc(&l->ticket);
    while (atomic_get(&l->owner) != ticket)
    { }

    return key;
}

k_spin_unlock(k_spinlock_t *l, key_t key) {
    atomic_inc(&l->owner);
    arch_irq_unlock(key);
}
```

Fairness test with ticket spinlock - nSIM

- nSIM simulator, ARCv2 architecture, 4 cores (modified nsim_hs_smp config)
- Works nicely:

```
CPU0 acquired spinlock 1000 times, expected 1000
CPU1 acquired spinlock 1000 times, expected 1000
CPU2 acquired spinlock 1000 times, expected 1000
CPU3 acquired spinlock 1000 times, expected 1000
```

© 2024 Synopsys, Inc.

Fairness test with ticket spinlock - HW

- HW board, ARCv2 architecture, 4 cores (hsdk config)

- Everything is OK, results:

```
CPU0 acquired spinlock 1000 times, expected 1000
CPU1 acquired spinlock 1000 times, expected 1000
CPU2 acquired spinlock 1000 times, expected 1000
CPU3 acquired spinlock 1000 times, expected 1000
```

Fairness test with ticket spinlock - QEMU

- QEMU, ARM cortex A53 architecture, 4 cores (modified qemu_cortex_a53_smp config)

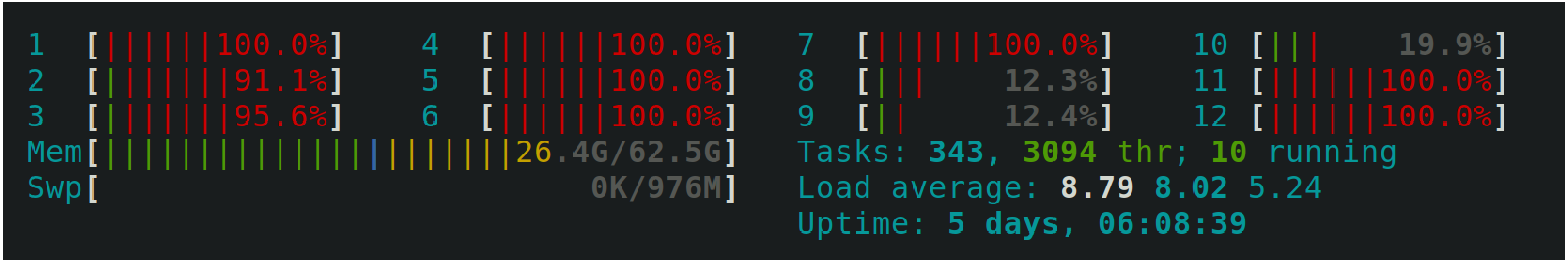
- Everything is OK, results:

```
CPU0 acquired spinlock 1000 times, expected 1000
CPU1 acquired spinlock 1000 times, expected 1000
CPU2 acquired spinlock 1000 times, expected 1000
CPU3 acquired spinlock 1000 times, expected 1000
```

Fairness test with ticket spinlock - QEMU

- But what if our host is a bit busy?

```
for i in $(seq $(($(getconf _NPROCESSORS_ONLN) - 2))); do
    cat /dev/random > /dev/null &
done
```



```
1   [||||||100.0%]     4   [||||||100.0%]    7   [||||||100.0%]     10 [|||      19.9%]
2   [|||||||91.1%]     5   [||||||100.0%]    8   [|||     12.3%]     11 [||||||100.0%]
3   [|||||||95.6%]     6   [||||||100.0%]    9   [||      12.4%]     12 [||||||100.0%]
Mem[||||||||||||||||||||||26.4G/62.5G]    Tasks: 343, 3094 thr; 10 running
Swp[                          0K/976M]    Load average: 8.79 8.02 5.24
                                          Uptime: 5 days, 06:08:39
```

Fairness test with ticket spinlock - QEMU

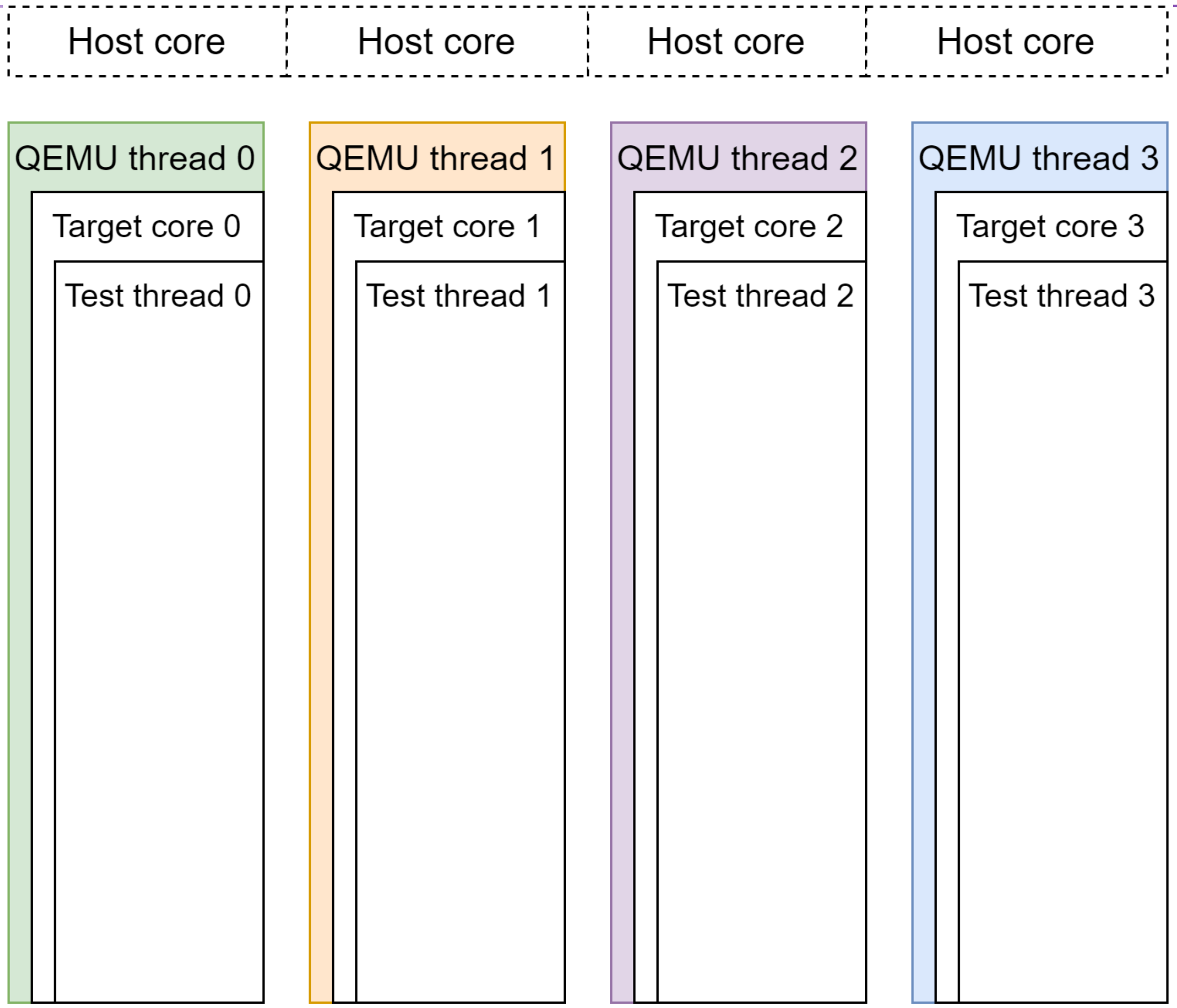
- QEMU, ARM cortex A53 architecture, 4 cores (modified qemu_cortex_a53_smp config)

- Ooops:

```
CPU0 acquired spinlock 99670 times, expected 100000
CPU1 acquired spinlock 100344 times, expected 100000
CPU2 acquired spinlock 100264 times, expected 100000
CPU3 acquired spinlock 99722 times, expected 100000
```

Fairness test with ticket spinlock - QEMU

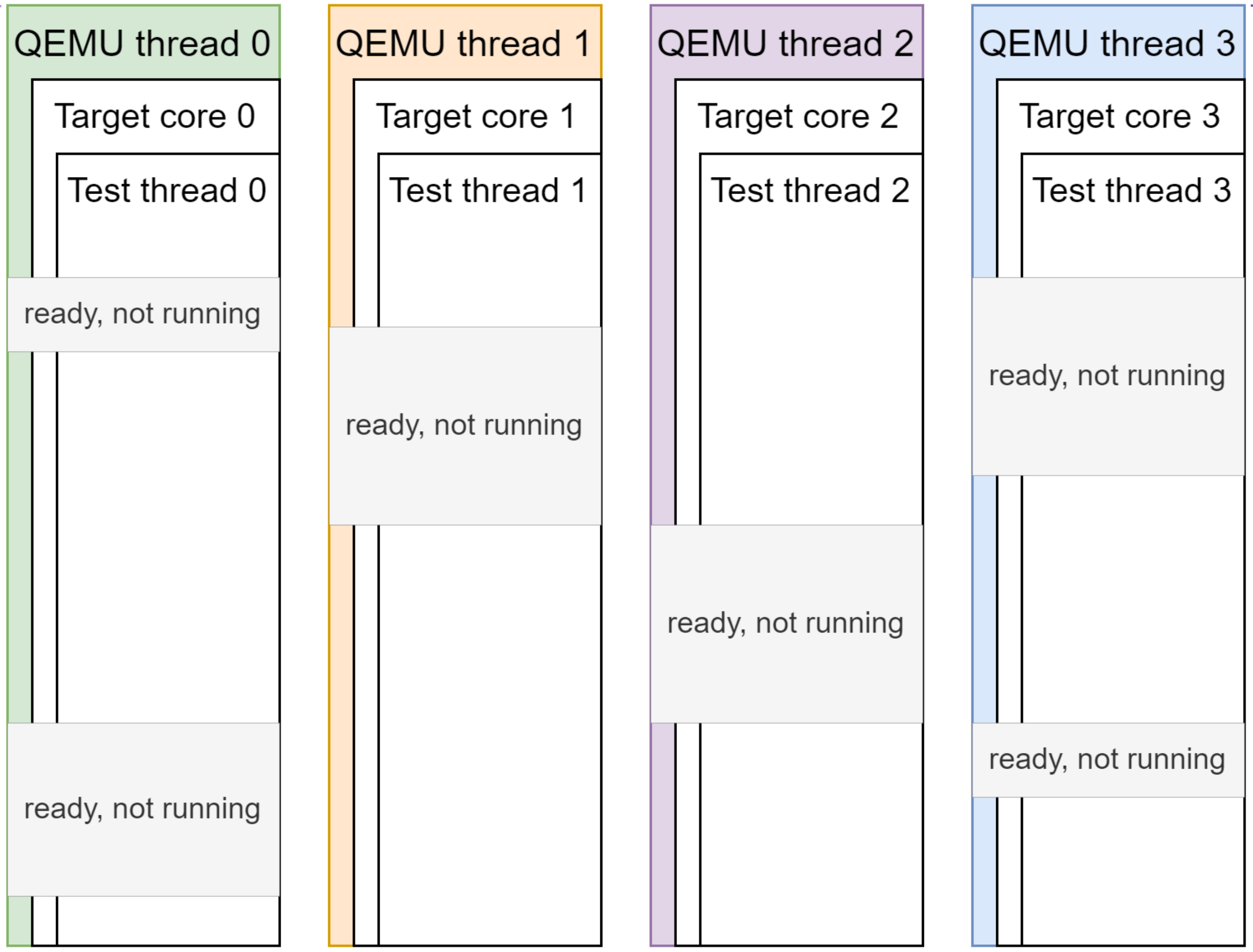
- qemu -smp cpus=4 …
- QEMU spawns several threads



Fairness test with ticket spinlock - QEMU

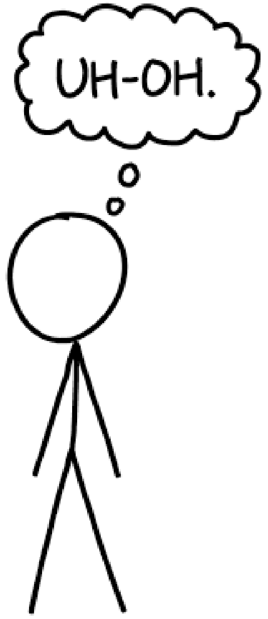
- High host load - QEMU threads more likely to be scheduled out

- If Zephyr thread request spinlock it'll get it in proper order
- But Zephyr thread may not request it as QEMU thread isn't running



Ticket spinlock

- We’ve improved situation with spinlock fairness
- Any drawbacks?



Ticket spinlock – drawbacks - slowdown

k_spin_lock()

+ atomic_get (memory read, 2 memory barriers)

k_spin_unlock()

– memory write (in atomic_clear)

+ atomic RMW (in atomic_inc)

Ticket spinlock – drawbacks - wrap around

```c
int k_spin_trylock(struct k_spinlock *l, k_spinlock_key_t *k) {
    int key = arch_irq_lock();
    atomic_val_t ticket_val = atomic_get(&l->owner);

    if (!atomic_cas(&l->tail, ticket_val, ticket_val + 1)) {
        arch_irq_unlock(key);
        return -EBUSY;
    }

    *k = key;
    return 0;
}
```

- Spinlock taken 0xffffffff + 1 times
  - requires 0xffffffff + 2 number of CPUs
- spinlock taken and released 0xffffffff times and taken again
  - Needs to be done while we execute several instructions in interrupts locked context

Ticket spinlock - wrap around

- Why it may happen

Ticket spinlock - wrap around

- Potential solution – use double atomic primitives
- Potential solution – store ticket and owner in halves of one atomic variable
- Instead of existing atomic_inc() we need to implement our own increments based on CAS

- Not (yet) implemented in Zephyr

Thanks!
Questions?

Contacts

Evgenii Paltsev

Eugeniy.Paltsev@synopsys.com
PaltsevEvgeniy@gmail.com

https://www.linkedin.com/in/evgeniy-paltsev

Images licensing notes

- The slides [4-6, 14, 16, 33, 36, 37] include images which are based on or includes content from xkcd.com.
- Content from xkcd.com is licensed under the Creative Commons Attribution-NonCommercial 2.5 license