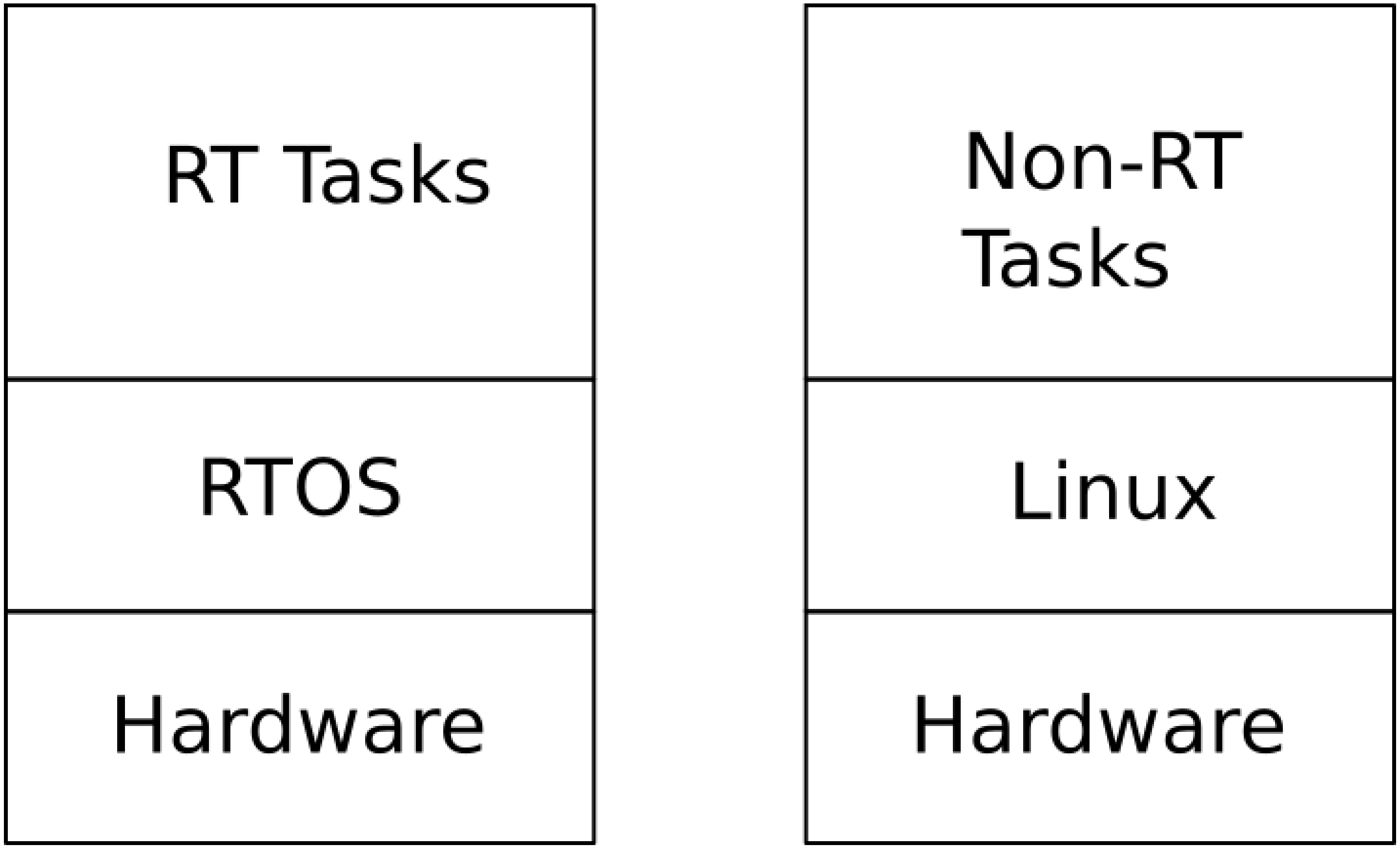
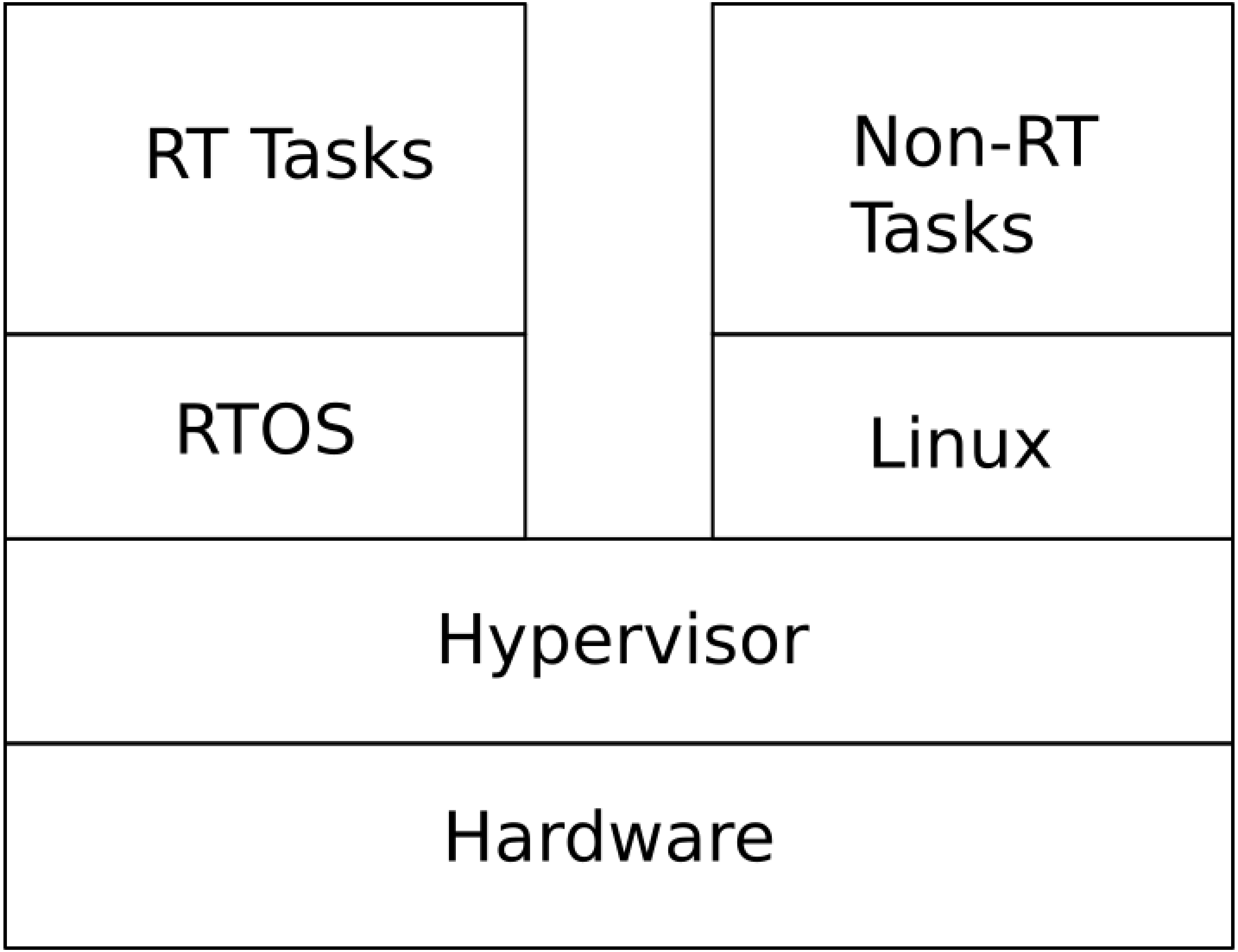
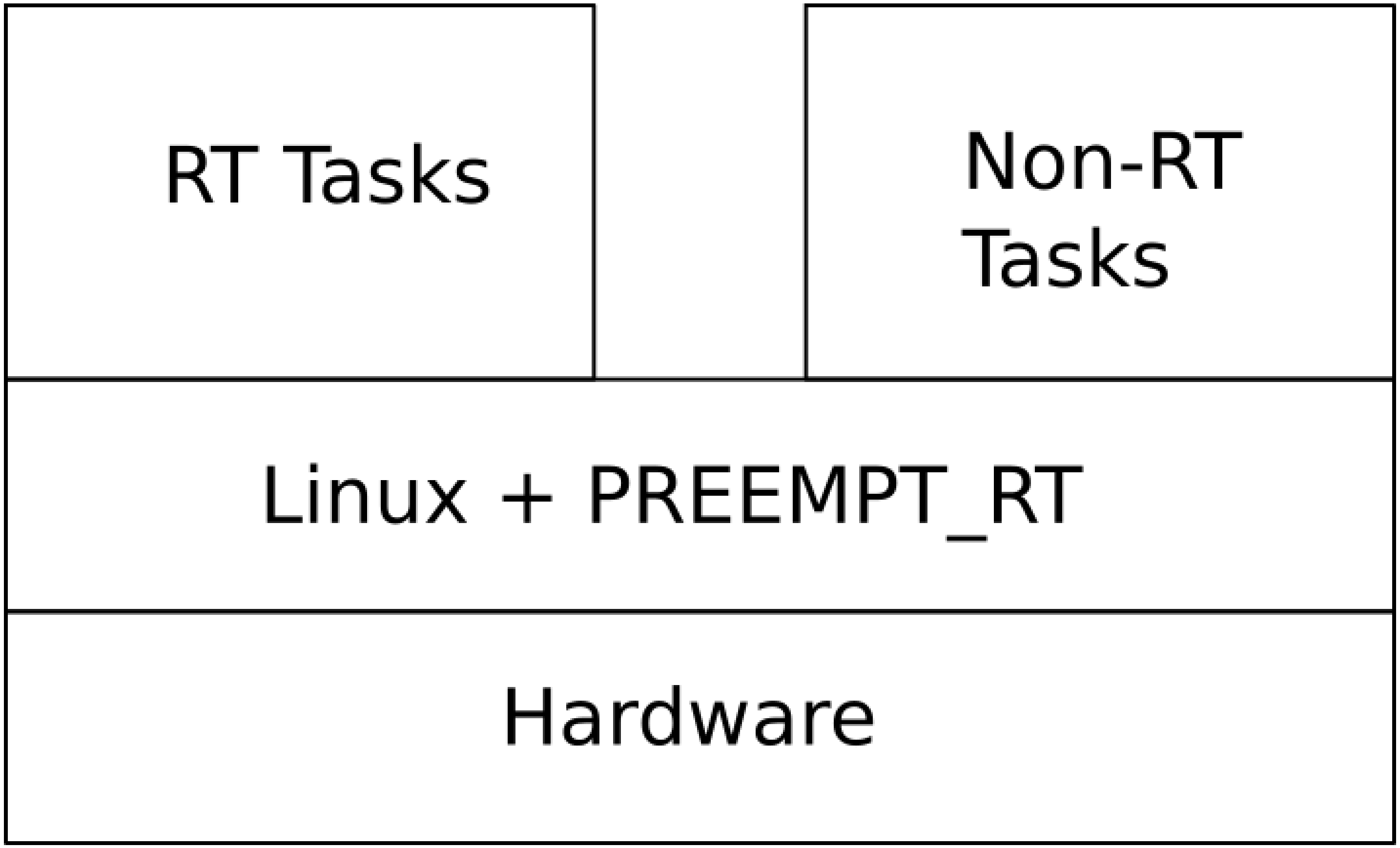
What every driver developer should know about RT

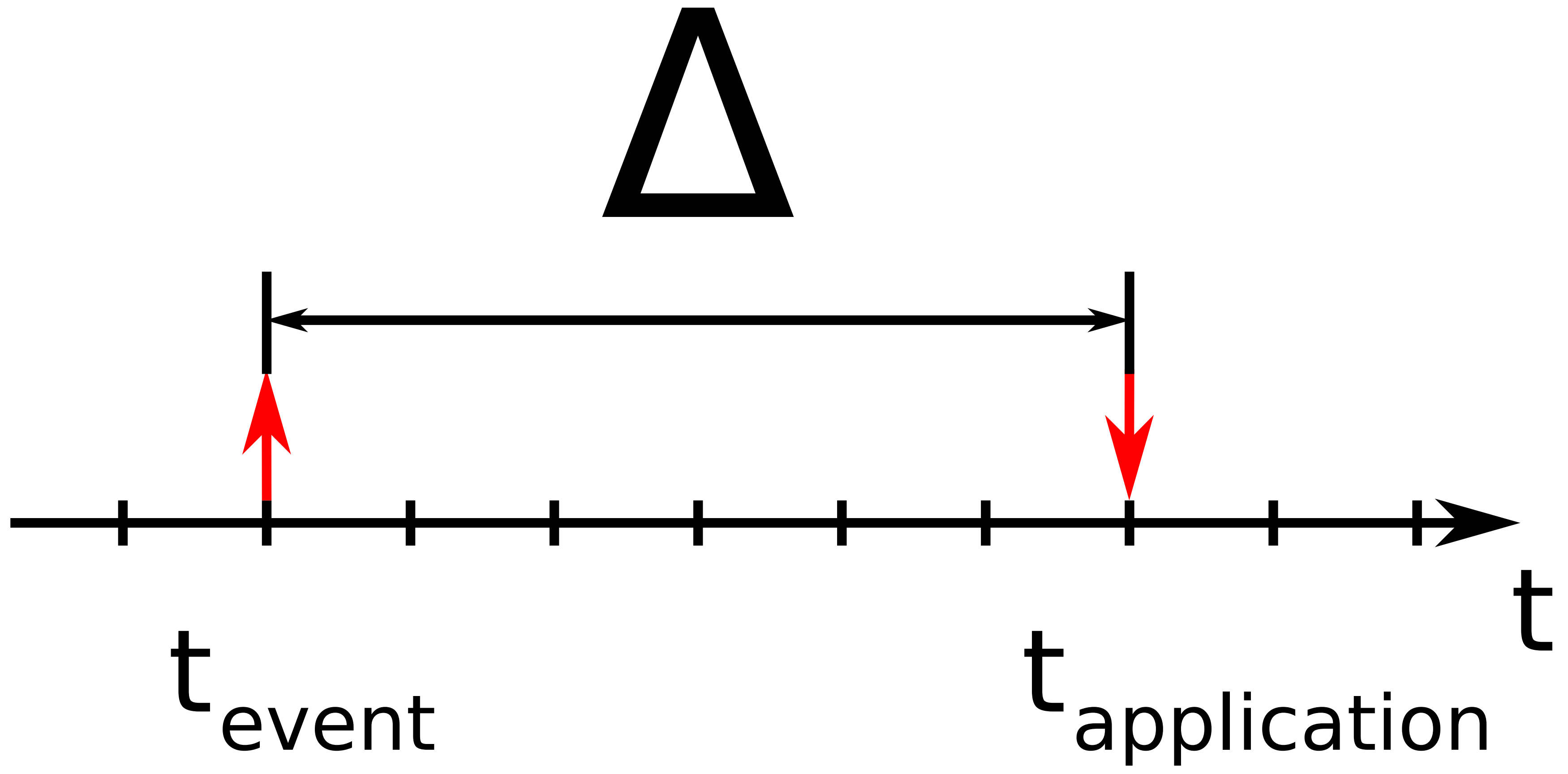
Julia Cartwright

National Instruments


RT Tasks
RTOS
Hardware
Non-RT Tasks
Linux
Hardware

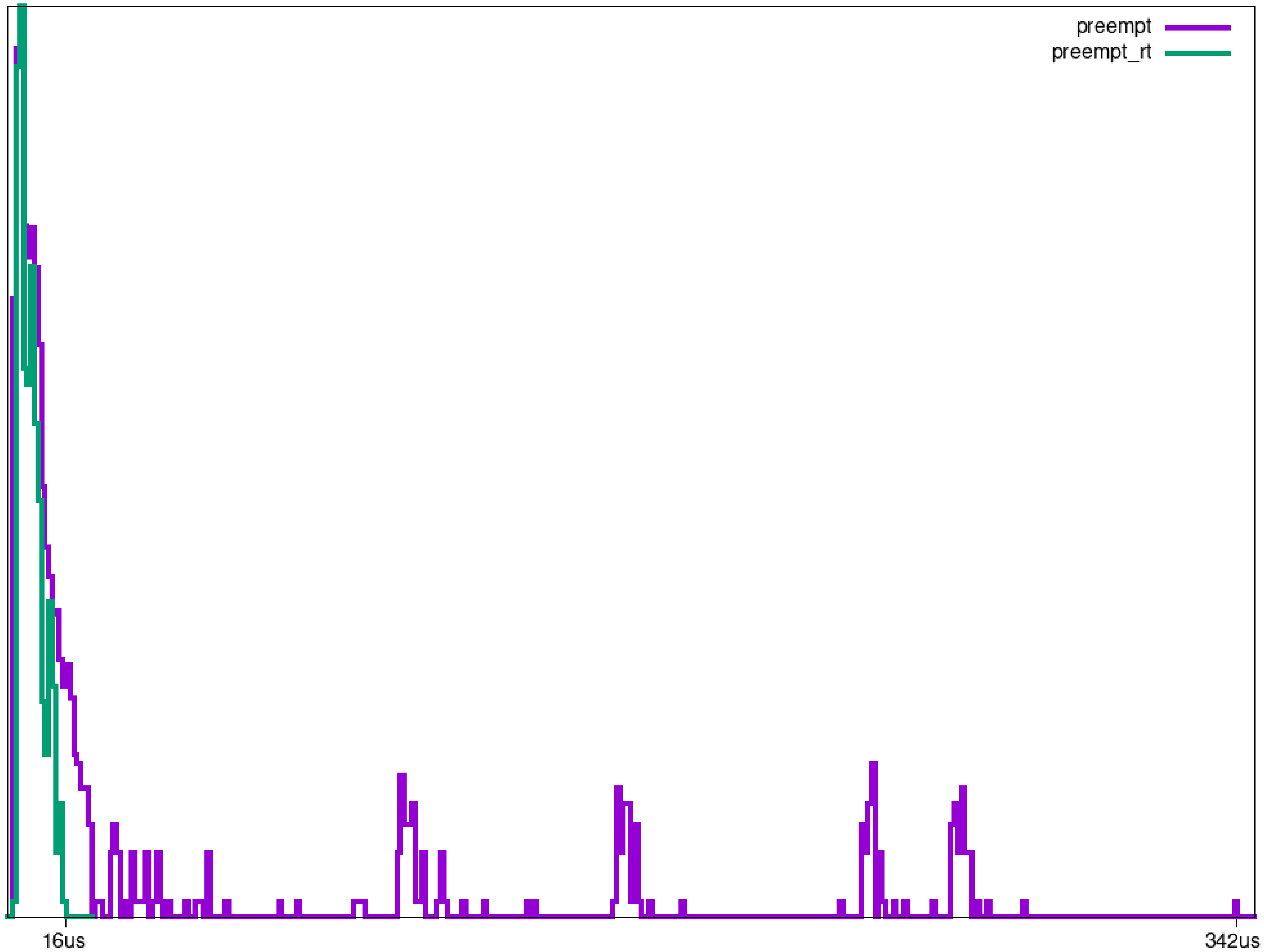

RT Tasks
Non-RT Tasks
RTOS
Linux
Hypervisor
Hardware

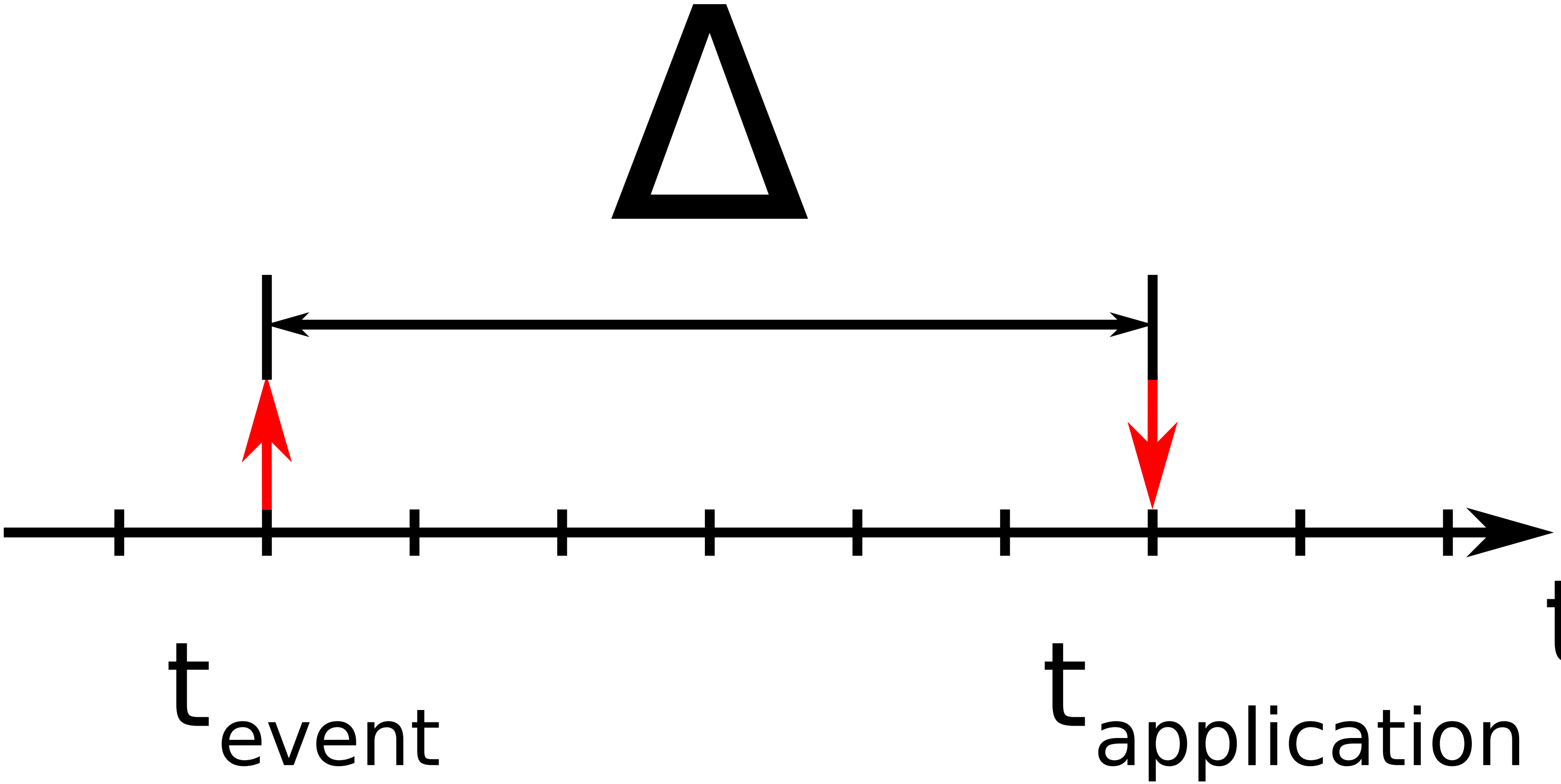

RT Tasks
Non-RT Tasks
Linux + PREEMPT_RT
Hardware

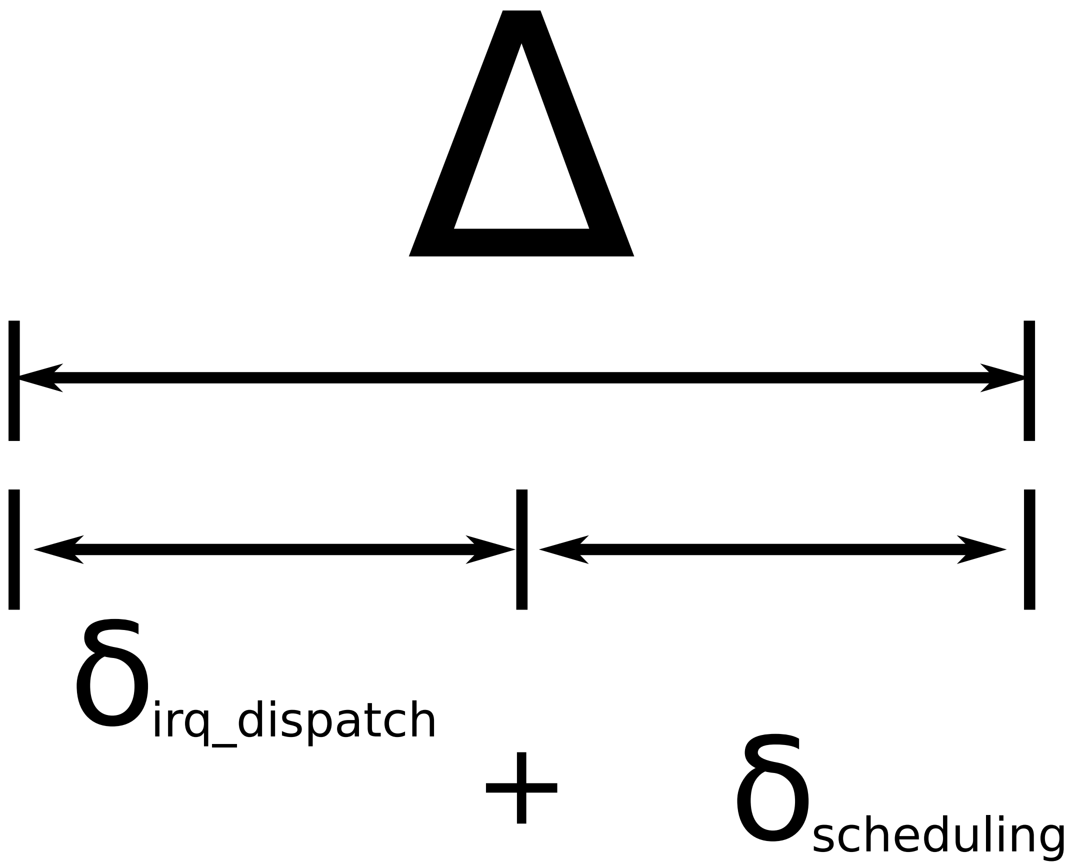


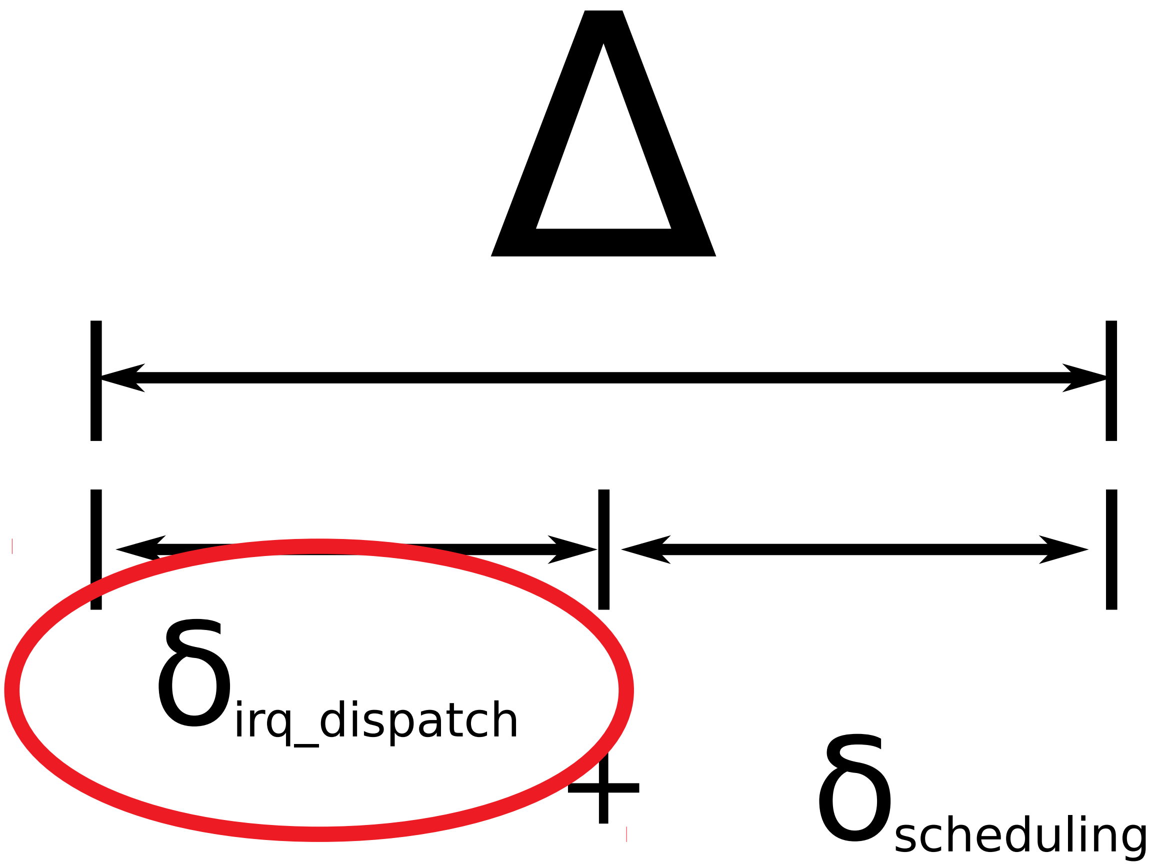
Δ
t
t_{event}
$t_{application}$

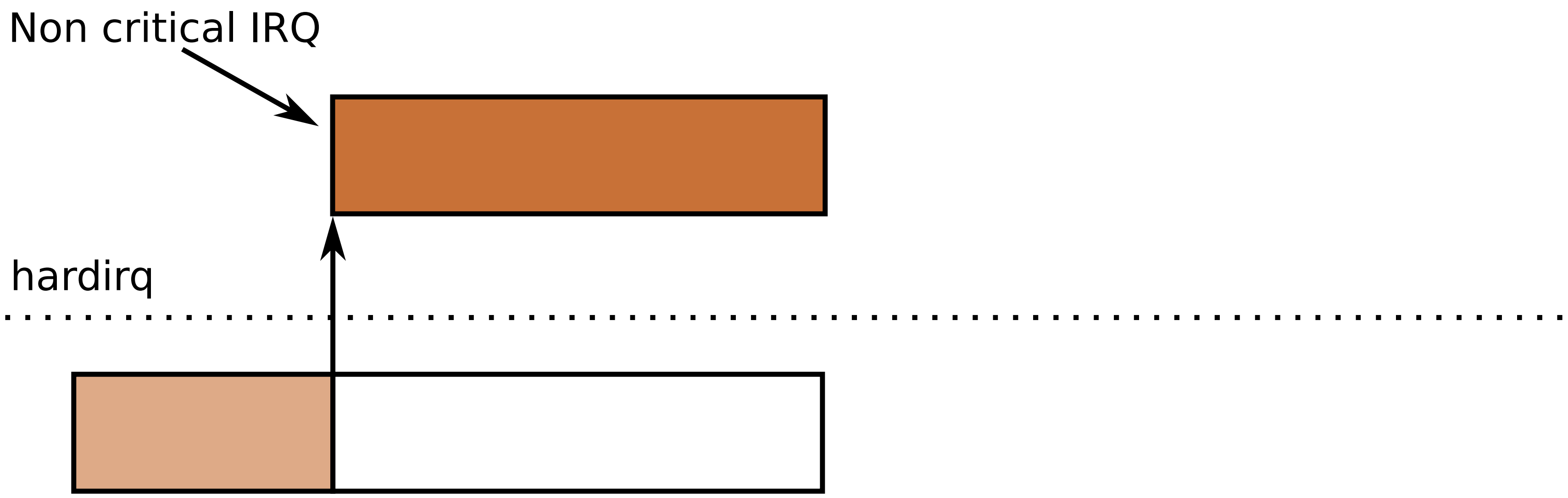
```
for (;;) {
    t0 = now();

    sleep(duration);

    t1 = now();

    plot(t1 - t0 - duration);
}
```

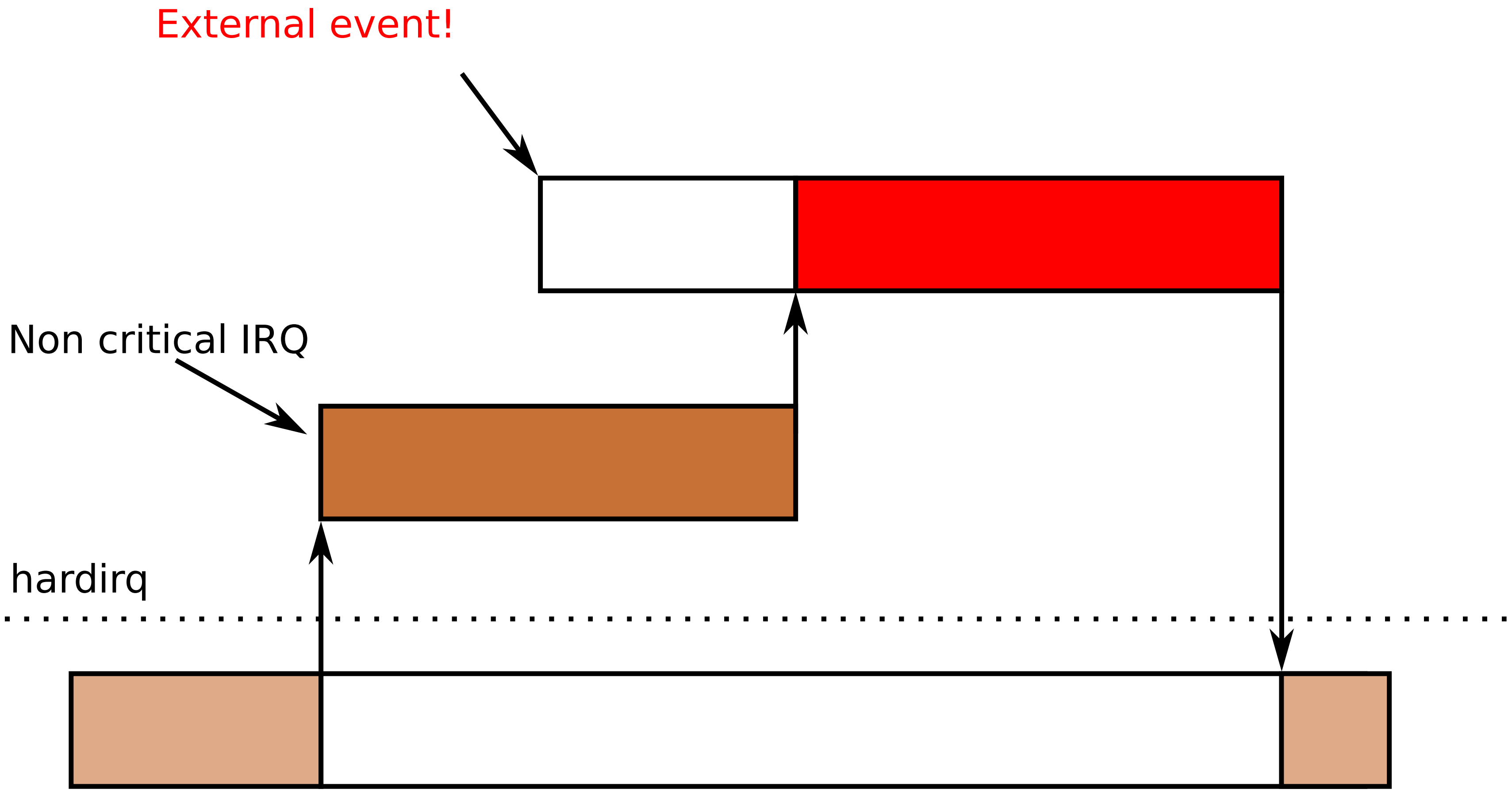

preempt
preempt_rt
16us
342us

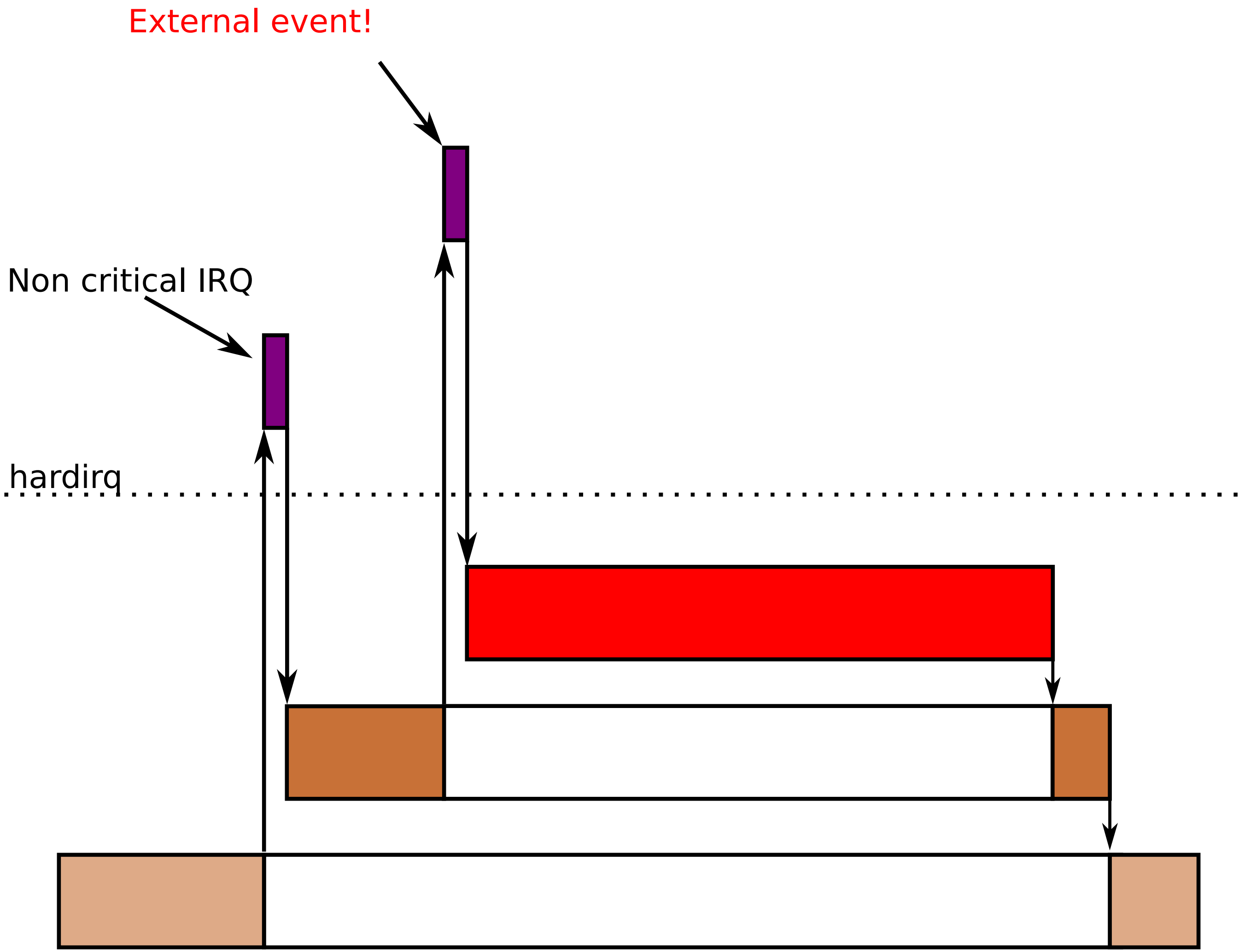

Δ
t
t_event
t_application


Δ
δirq_dispatch
+
δscheduling


Δ
δirq_dispatch
+
δscheduling


Non critical IRQ
hardirq


External event!
Non critical IRQ
hardirq


External event!
Non critical IRQ
hardirq

Good news for driver developers:

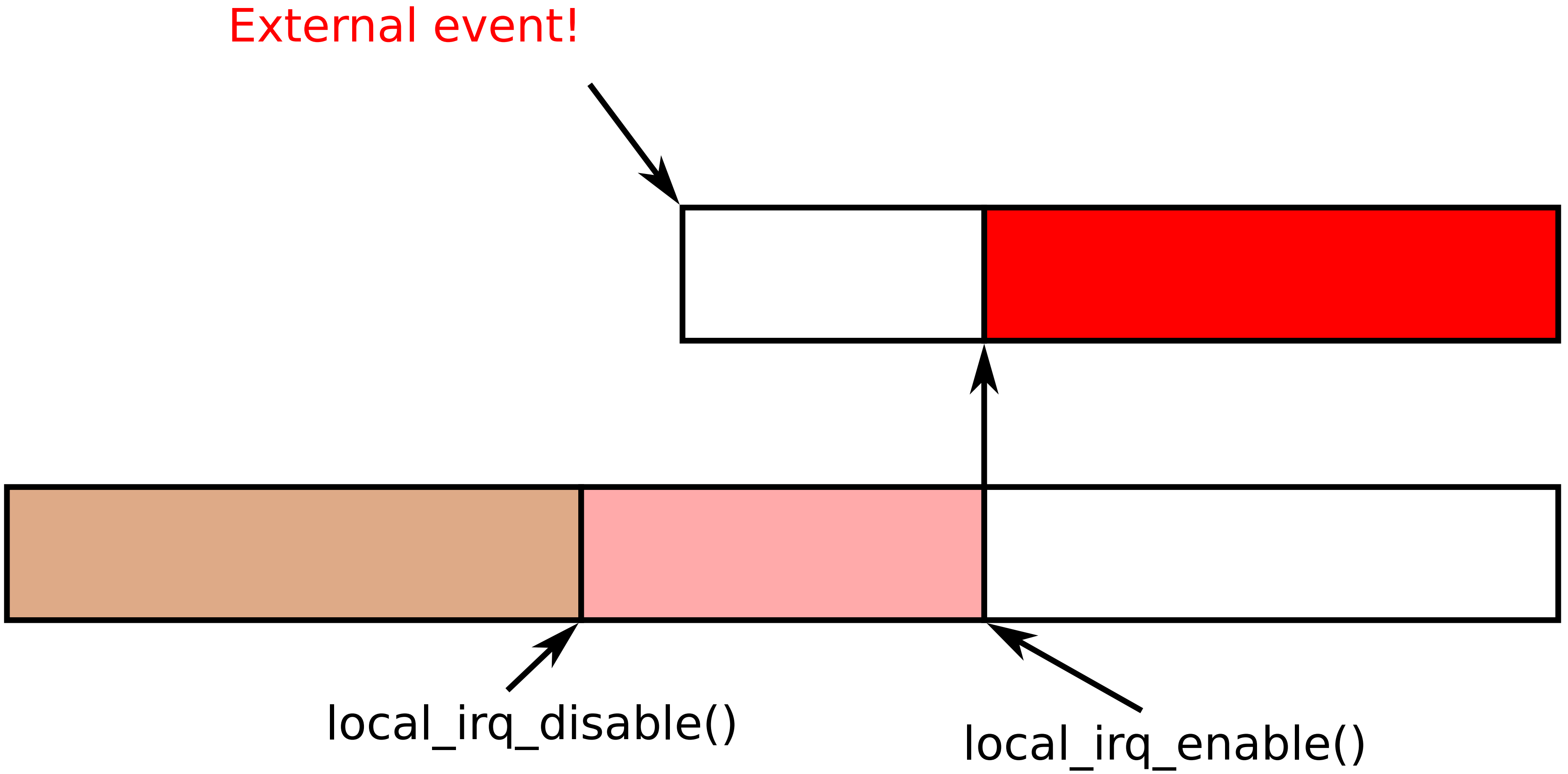
Most drivers need no modification to participate in forced irq threading.

Force threaded IRQs work great, but not for:

1. Code which is involved in hardirq dispatching: irqchips, gpio-irq, etc.

2. Code which can be invoked by the scheduler directly: cpufreq, cpuidle, etc.

For these usecases, register a handler with IRQF_NO_THREAD.


External event!
local_irq_disable()
local_irq_enable()

Why are local_irq_{disable,enable}() being used?

1. Legitimate usecase: synchronizing with Hardirq context on local CPU (cpufreq, cpuidle, etc.)
   - audit to be minimal and bounded

2. Usage is SMP bug.
   - fix to use proper spinlock

3. Heavyhanded way to prevent migration during per-CPU variable accesses.
   - use local locks

```c
#include <linux/locallock.h>

DEFINE_LOCAL_IRQ_LOCK(lock);

void foo(void)
{
        local_lock_irq(lock);

        /* stuff */

        local_unlock_irq(lock);
}
```

LOCAL_IRQ_LOCK semantics:

- Critical sections may execute concurrently on different CPUs.

- On any given CPU, the owner task may recurse into a critical section (locks are recursive)

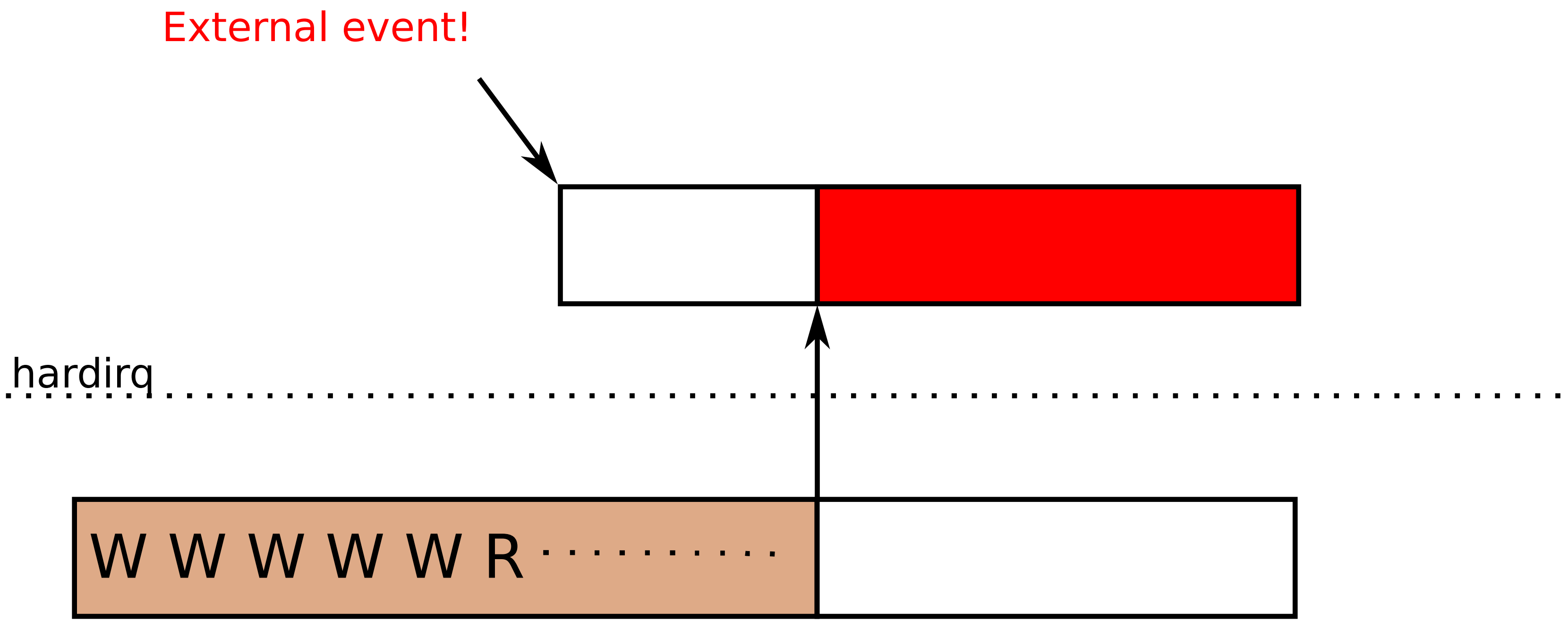
- When contended, the blocking task sleeps on RT

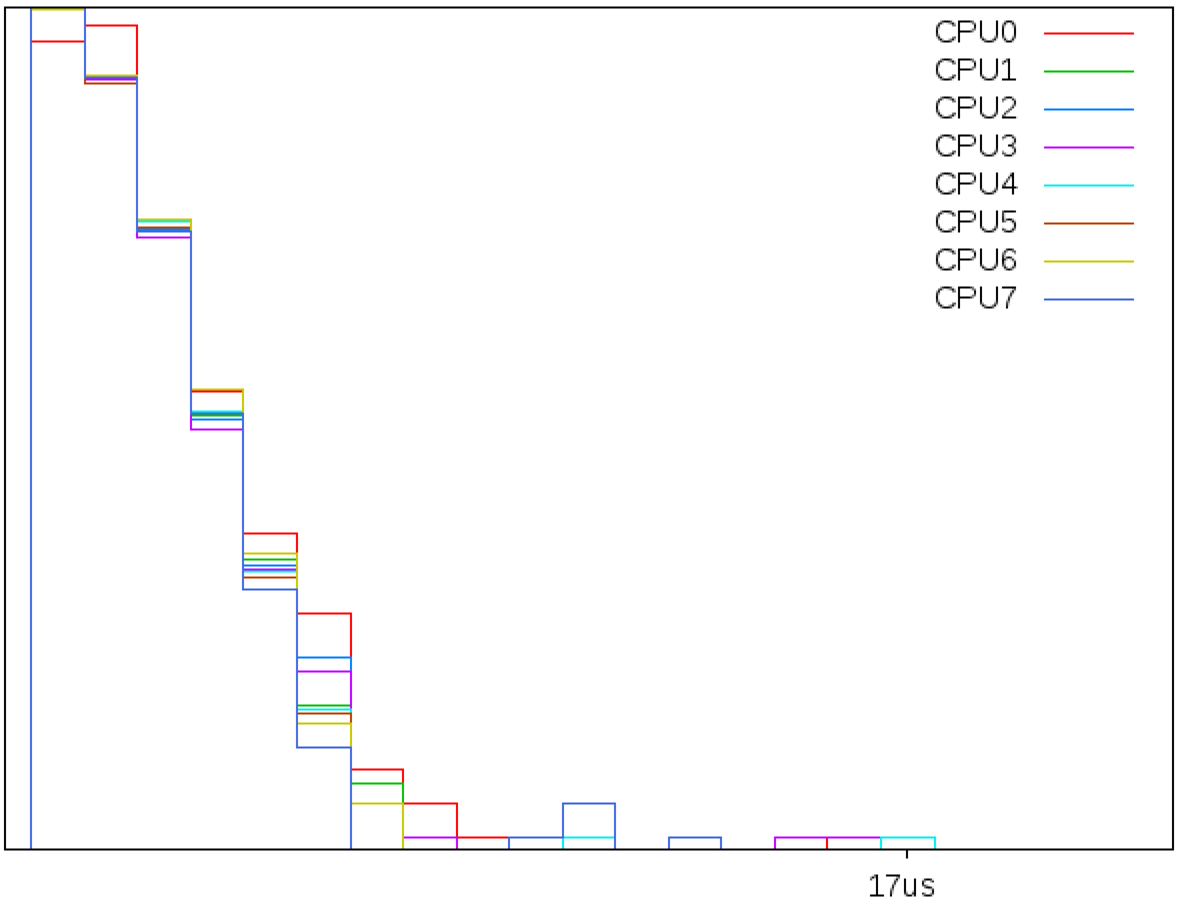
- Critical sections are otherwise fully preemptible on RT

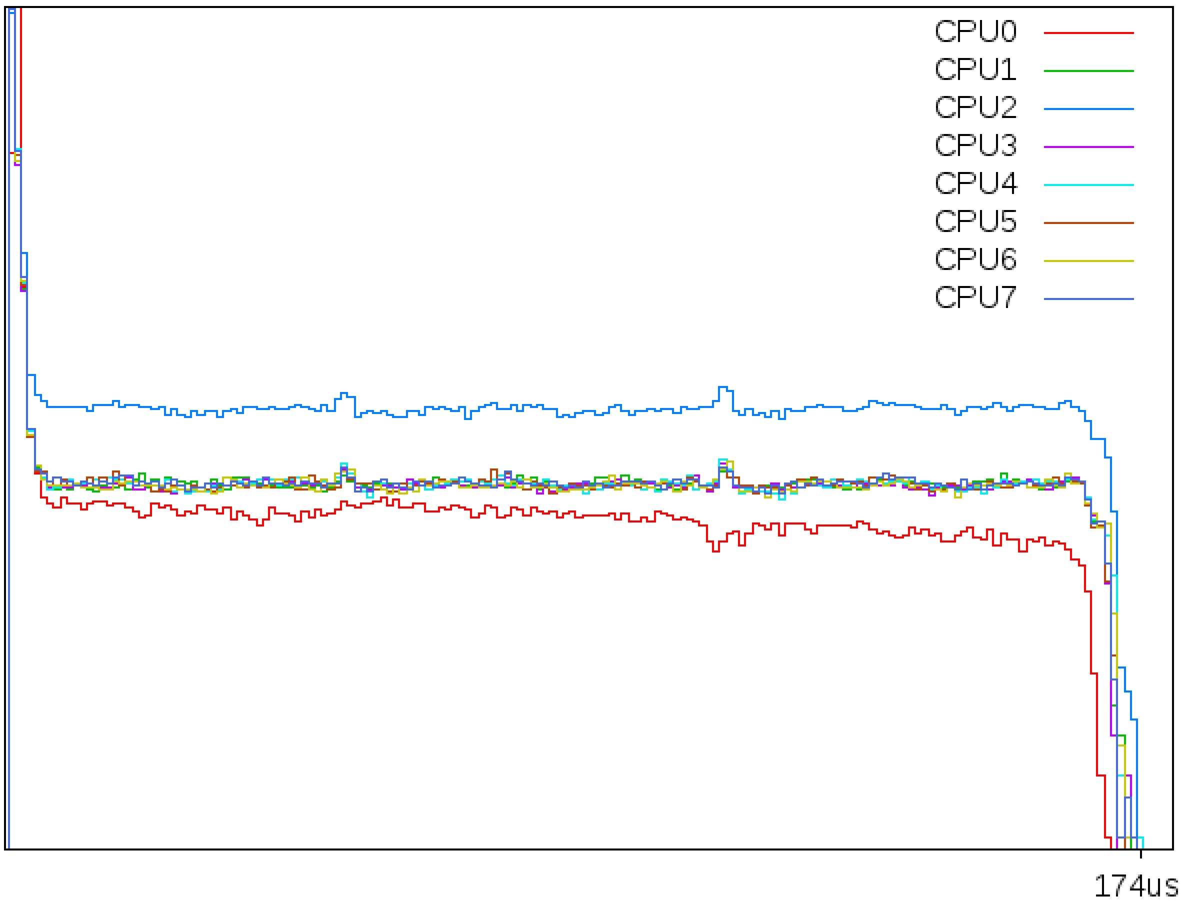
What about:

```
spin_lock_irq()

spin_lock_irqsave()

                  ?
```

```c
void initialize_my_device(struct my_device *md)
{
	writel(0xDEADBEEF, md->regs + REG1);
	writel(0x00000001, md->regs + REG2);
	/* ... */
	writel(0xEEEEEEEE, md->regs + REGN);

	while (!(readl(md->regs + REGSTATUS) & DONE_BIT))
		cpu_relax();
}
```


External event!
hardirq
W W W W W R . . . .

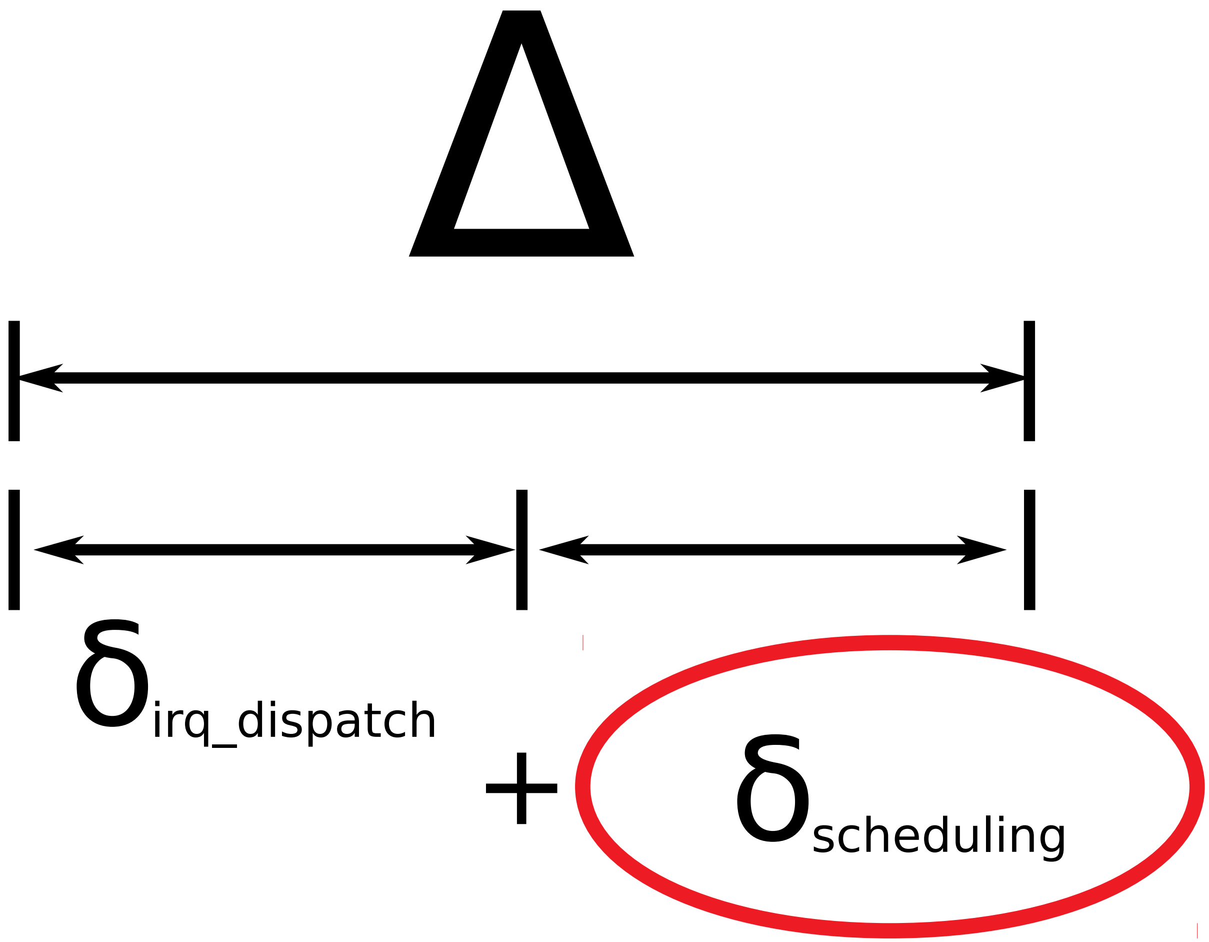

CPU0
CPU1
CPU2
CPU3
CPU4
CPU5
CPU6
CPU7
17us

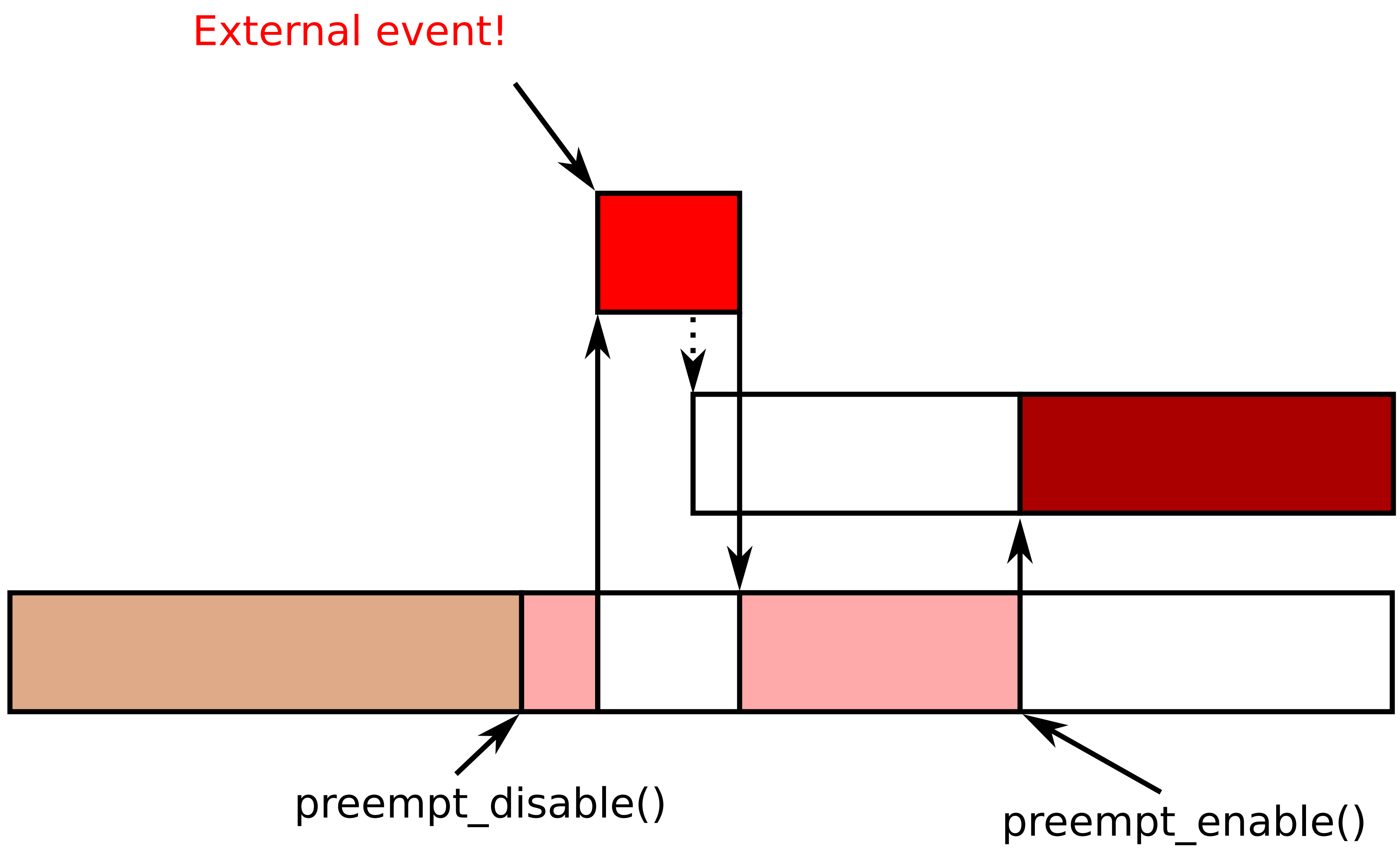

CPU0
CPU1
CPU2
CPU3
CPU4
CPU5
CPU6
CPU7
174us

Additional MMI-woes:

- MMIO access incurs latency due to device power state transition.

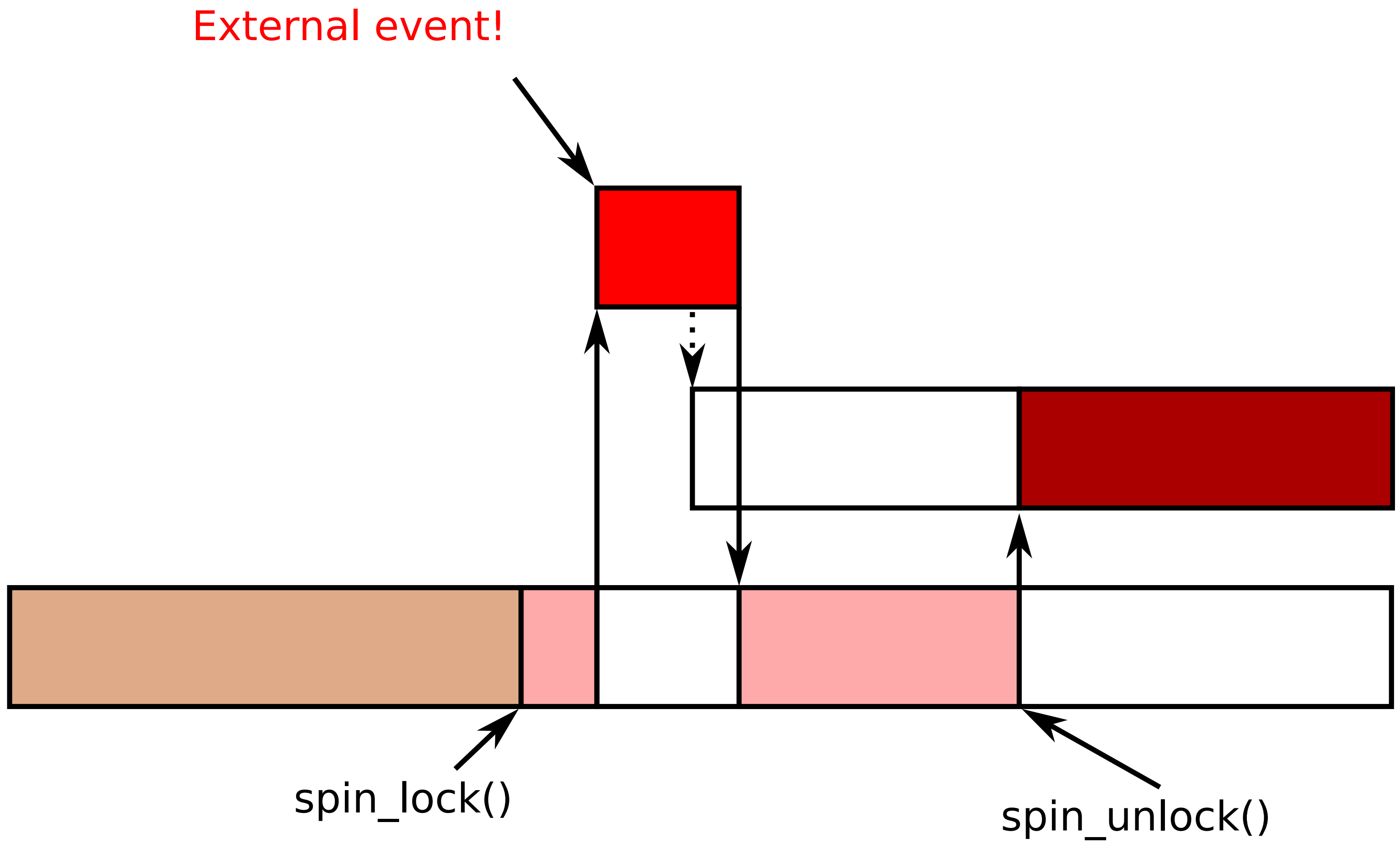
- MMIO access is on incredibly slow bus (MMIO-mapped SPI bus).

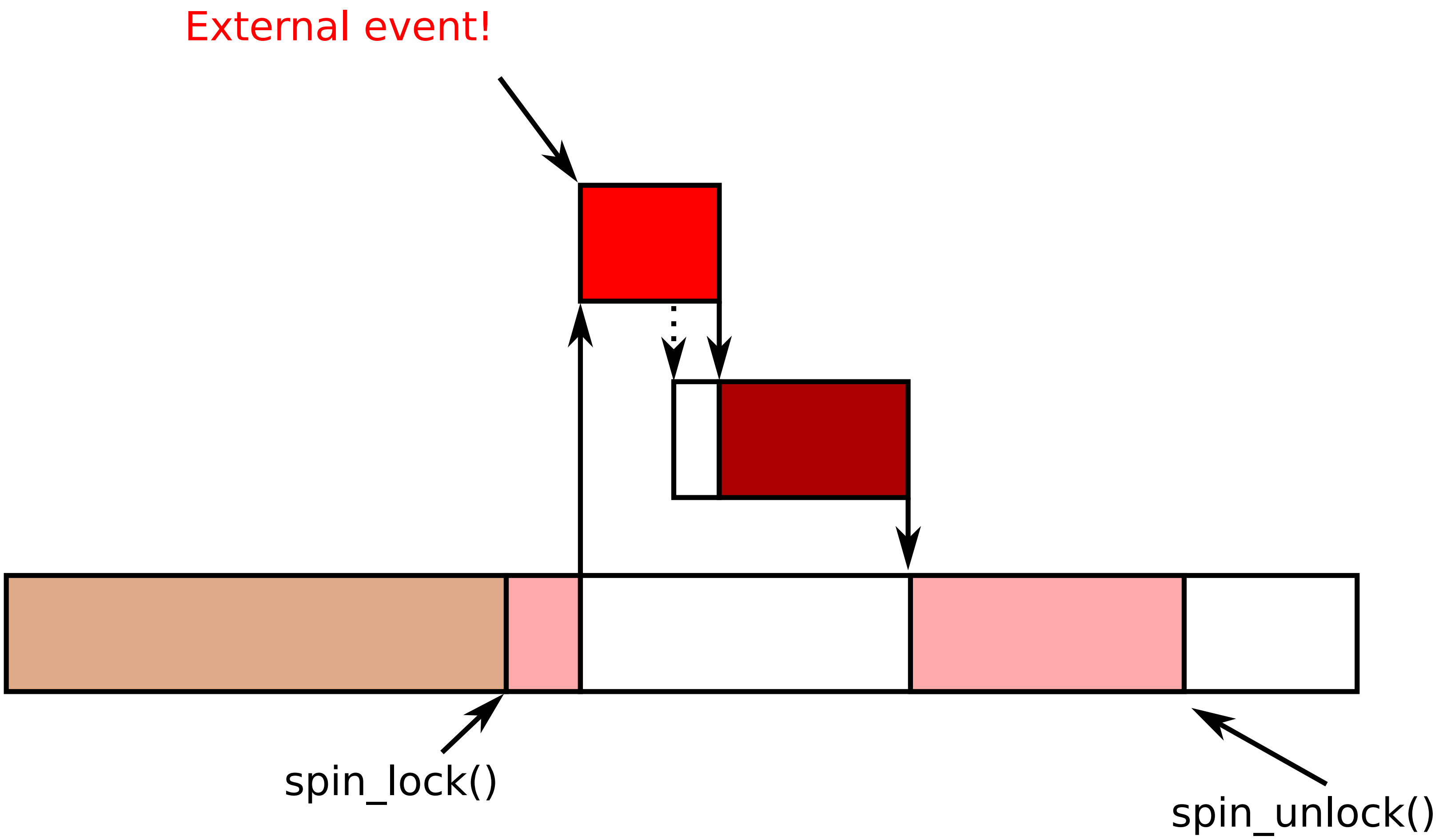

Δ
δirq_dispatch
+
δscheduling


External event!
preempt_disable()
preempt_enable()

The only reason a driver should be using preempt_disable()/preempt_enable():

- If the driver is in some way by the scheduler; for example, cpufreq, cpuidle.


External event!
spin_lock()
spin_unlock()


External event!
spin_lock()
spin_unlock()

Good news for driver developers:

Most drivers require no changes to have their spin_lock critical sections preemptible.

```
typedef /* … */ raw_spinlock_t;

raw_spin_lock_init(lock);

raw_spin_lock(lock);
raw_spin_lock_irq(lock);
raw_spin_lock_irqsave(lock, flags);

raw_spin_unlock_irqrestore(lock, flags);
raw_spin_unlock_irq();
raw_spin_unlock();
```

If your drivers aren’t involved in interrupt dispatch, then you shouldn’t use local_irq_disable(), use local locks.

Consider MMIO access patterns and their impact to RT.

If your drivers aren’t involved in scheduling, then you shouldn’t use preempt_disable(), use local locks or per-cpu access primitives.

If your drivers <u>are</u> involved in interrupt dispatch or scheduling, they must use raw_spin_lock(), and all critical sections need to be minimal and bounded.

?

julia@{ni.com,kernel.org}

@juliasramblings

linux-rt-users@vger.kernel.org

#linux-rt on OFTC