Real-Time is coming to Linux

What does that mean for you?

Steven Rostedt
10/24/2018

vmware

Who is this talk for?

- Linux kernel developers
  - Core kernel code
  - Driver code
  - File System code
  - Pretty much anyone touching the Linux Kernel
- Those that want to know why PREEMPT_RT makes Linux different

Who is this talk for?

- Linux kernel developers
  - Core kernel code
  - Driver code
  - File System code
  - Pretty much anyone touching the Linux Kernel
- Those that want to know why PREEMPT_RT makes Linux different
- Those that want to see how fast Steven talks in Real Time

Review

- What is Real-Time?

Review

- What is Real-Time?

  What is your favourite colour?

Review

- What is Real-Time?

  What is your favourite colour? **BLUE**

Review

- What is Real-Time?

  What is your favourite colour? no **RED!**

Review

- What is Real-Time?
  - The term is ambiguous

Review

- What is Real-Time?
  - The term is ambiguous
  - Top definition from http://urbandictionary.com

> REAL TIME instantaneous; taking place at once as other things are also in progress. “When I surveyed the situation in real time, there were only 4 people who met the qualifications”.
>
> #instantaneous #simultaneously #survey #in progress #process #momentary

Review

- What is Real-Time?
  - The term is ambiguous
  - Top definition from http://WhatIs.com

> REAL TIME is a level of computer responsiveness that a user senses as sufficiently immediate or that enables the computer to keep up with some external process.

Review

- What is Real-Time?
  - The term is ambiguous
  - Top definition from http://WhatIs.com

> REAL TIME is an adjective pertaining to computers or processes that operate in real time.

Review

- What is Real-Time?
  - The term is ambiguous
  - Top definition from http://WhatIs.com

> REAL TIME describes a human rather than a machine sense of time.

Review

- What is Real-Time?
  - What does it mean as per PREEMPT_RT (aka “The Real-Time Patch”)?

Review

- What is Real-Time?
  - What does it mean as per PREEMPT_RT (aka “The Real-Time Patch”)?
    - Means determinism
      - Has nothing to do with speed
      - Only latency
    - Can calculate worse case scenarios
    - Can determine what will happen
    - Can determine when it will happen

Review

- What is Real-Time?
  - What does it mean as per PREEMPT_RT (aka “The Real-Time Patch”)?
    - Means determinism
      - Has nothing to do with speed
      - Only latency
    - Can calculate worse case scenarios
    - Can determine what will happen
    - Can determine when it will happen
- Should be called a Deterministic Operating System

Review

DOS!

Review

- What's the strategy?
  - To make the kernel as preemptive as possible
  - Remove preemption and interrupt disabling
    - Where we can
  - Allow scheduling to happen (almost) everywhere!

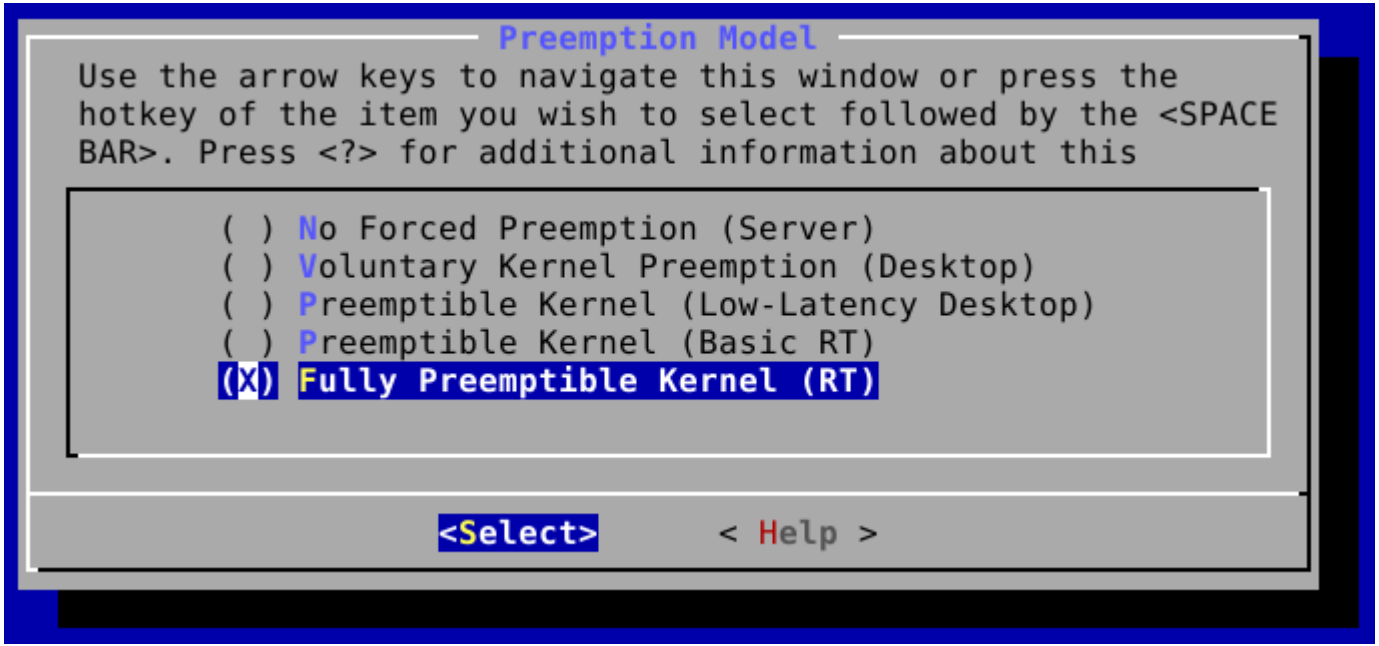
Review

- What's the strategy?
  - To make the kernel as preemptive as possible
  - Remove preemption and interrupt disabling
    - Where we can
  - Allow scheduling to happen (almost) everywhere!

Let the most important (highest priority) task run, when it wants to run.

Enabling PREEMPT_RT

- In “Processor type and features”



Enabling PREEMPT_RT

- In “Processor type and features”
  - “No Forced Preemption” - CONFIG_PREEMPT_NONE
  - “Voluntary Kernel Preemption” - CONFIG_PREEMPT_VOLUNTARY
  - “Preemptible Kernel (Low-Latency Desktop)” - CONFIG_PREEMPT_LL
  - “Preemptible Kernel (Basic RT) - CONFIG_PREEMPT_RTB
  - “Full Preemptible Kernel” - CONFIG_PREEMPT_RT_FULL

Enabling PREEMPT_RT

- In “Processor type and features”
  - “No Forced Preemption” - CONFIG_PREEMPT_NONE
  - “Voluntary Kernel Preemption” - CONFIG_PREEMPT_VOLUNTARY
  - “Preemptible Kernel (Low-Latency Desktop)” - CONFIG_PREEMPT_LL
  - “Preemptible Kernel (Basic RT) - CONFIG_PREEMPT_RTB
  - “Full Preemptible Kernel” - CONFIG_ **PREEMPT_RT** _FULL

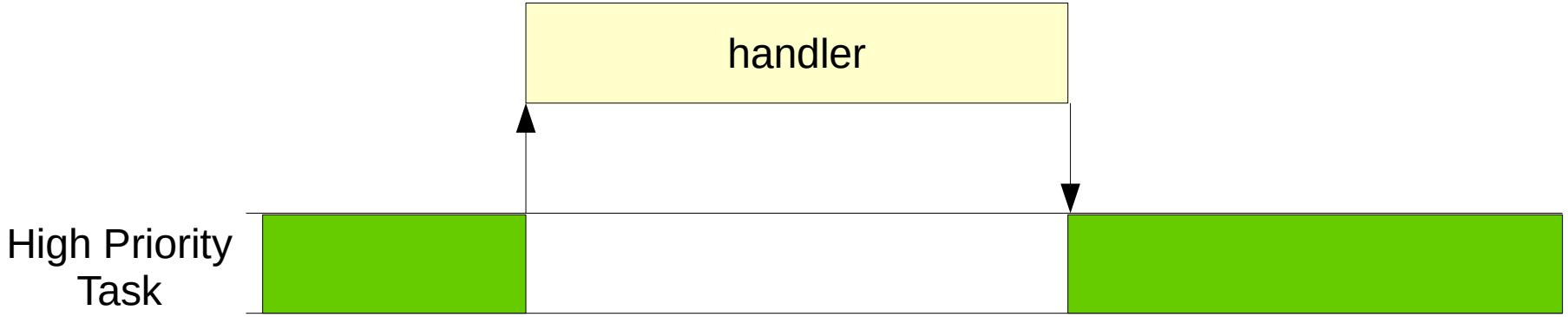
Enabling PREEMPT_RT

- Interrupts as threads
  - requst_threaded_irq( ) - Been in the kernel since 2009!
  - All interrupts as threads (“threadirqs”) - Been in the kernel since 2011
- Not all interrupts become Threads
  - IRQF_NO_THREAD
    - Timer Interrupts
    - IPI (Inter-Processor Interrupts)
  - IRQF_PERCPU
  - IRQF_ONESHOT

Interrupts

- Normal Interrupt

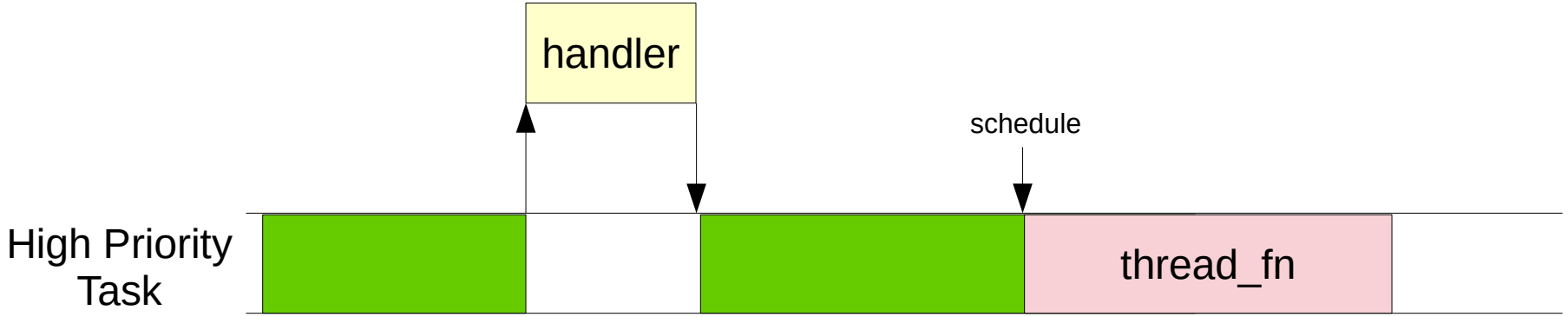
request_irq(unsigned int irq, irq_handler_t **handler**, unsigned long flags,
const char *name, void *dev)



Interrupts

- Threaded Interrupt

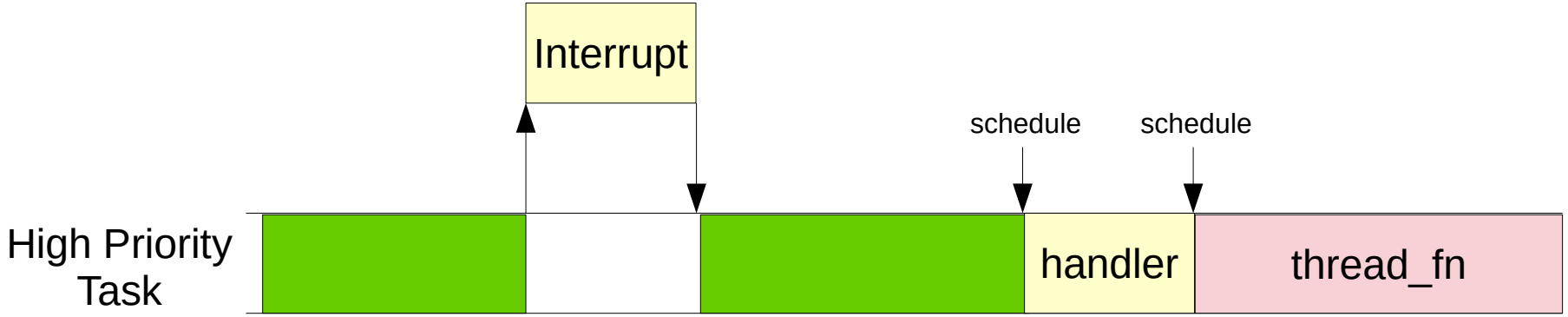
```
request_threaded_irq(unsigned int irq, irq_handler_t handler,
        irq_handler_t thread_fn,
        unsigned long flags, const char *name, void *dev)
```



Interrupts

- Forced Threaded Interrupts

```
request_threaded_irq(unsigned int irq, irq_handler_t handler,
        irq_handler_t thread_fn,
        unsigned long flags, const char *name, void *dev)
```



Enabling PREEMPT_RT

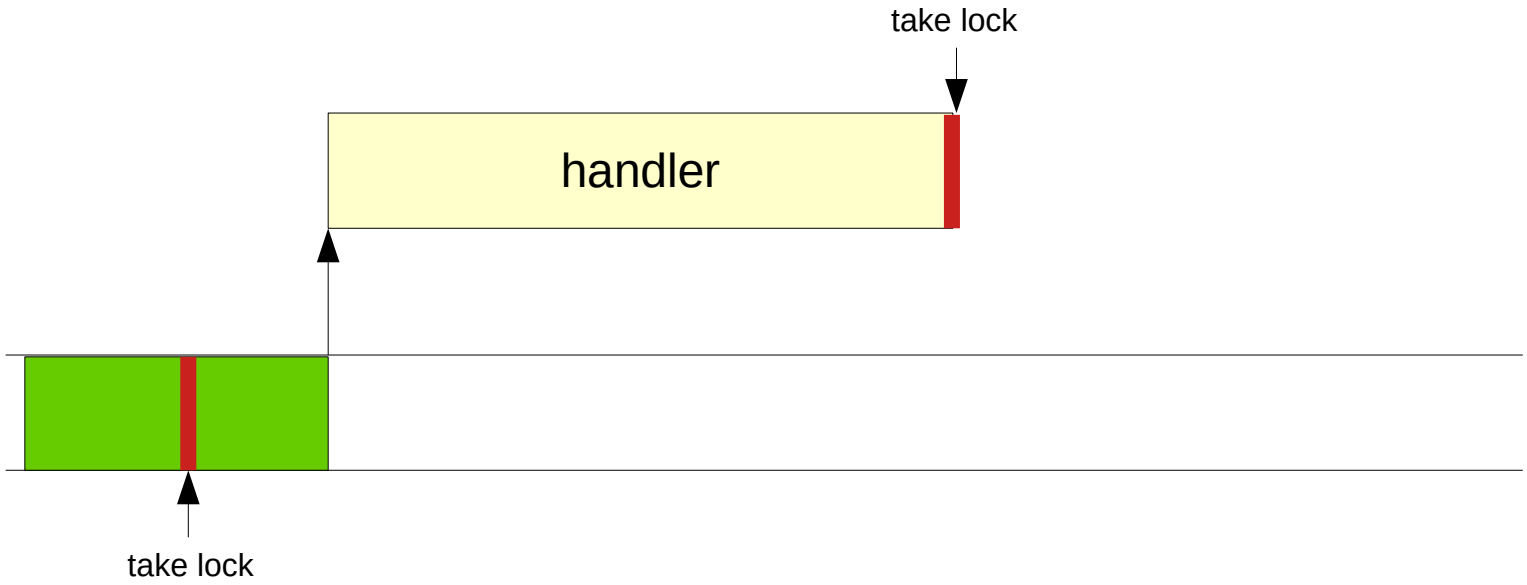
- spin_lock*( ) becomes a mutex
  - Well, they are not really spinning locks anymore, are they?
  - They do not disable preemption
  - They do not disable interrupts (even spin_lock_irq*( ))

Enabling PREEMPT_RT

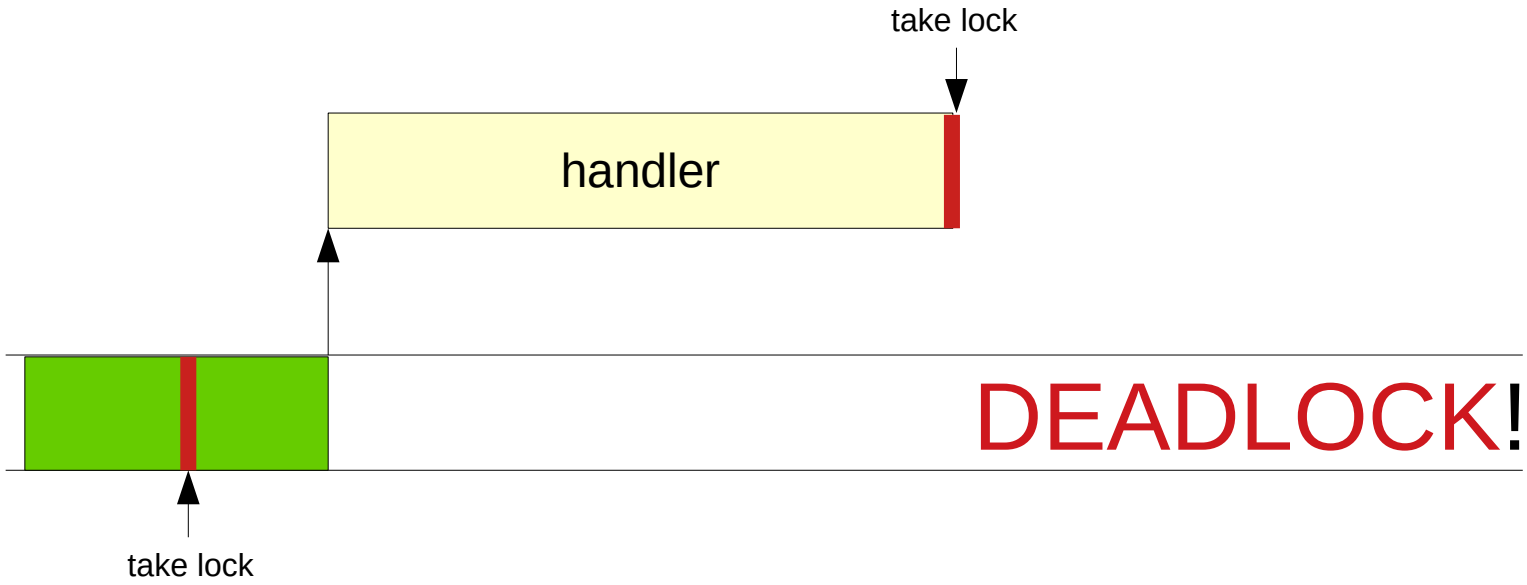
- spin_lock*( ) becomes a mutex
  - Well, they are not really spinning locks anymore, are they?
  - They do not disable preemption
  - They do not disable interrupts (even spin_lock_irq*( ))

    How? Why?

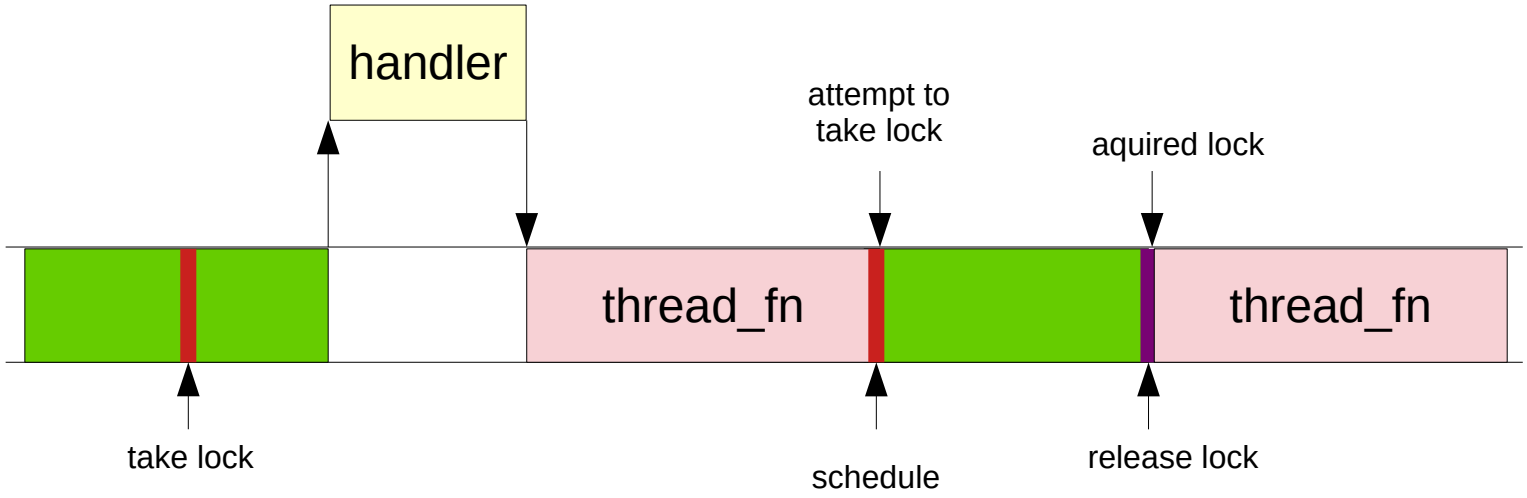
Spin Locks



Spin Locks



Spin Locks


handler
attempt to
take lock
aquired lock
thread_fn
thread_fn
take lock
schedule
release lock

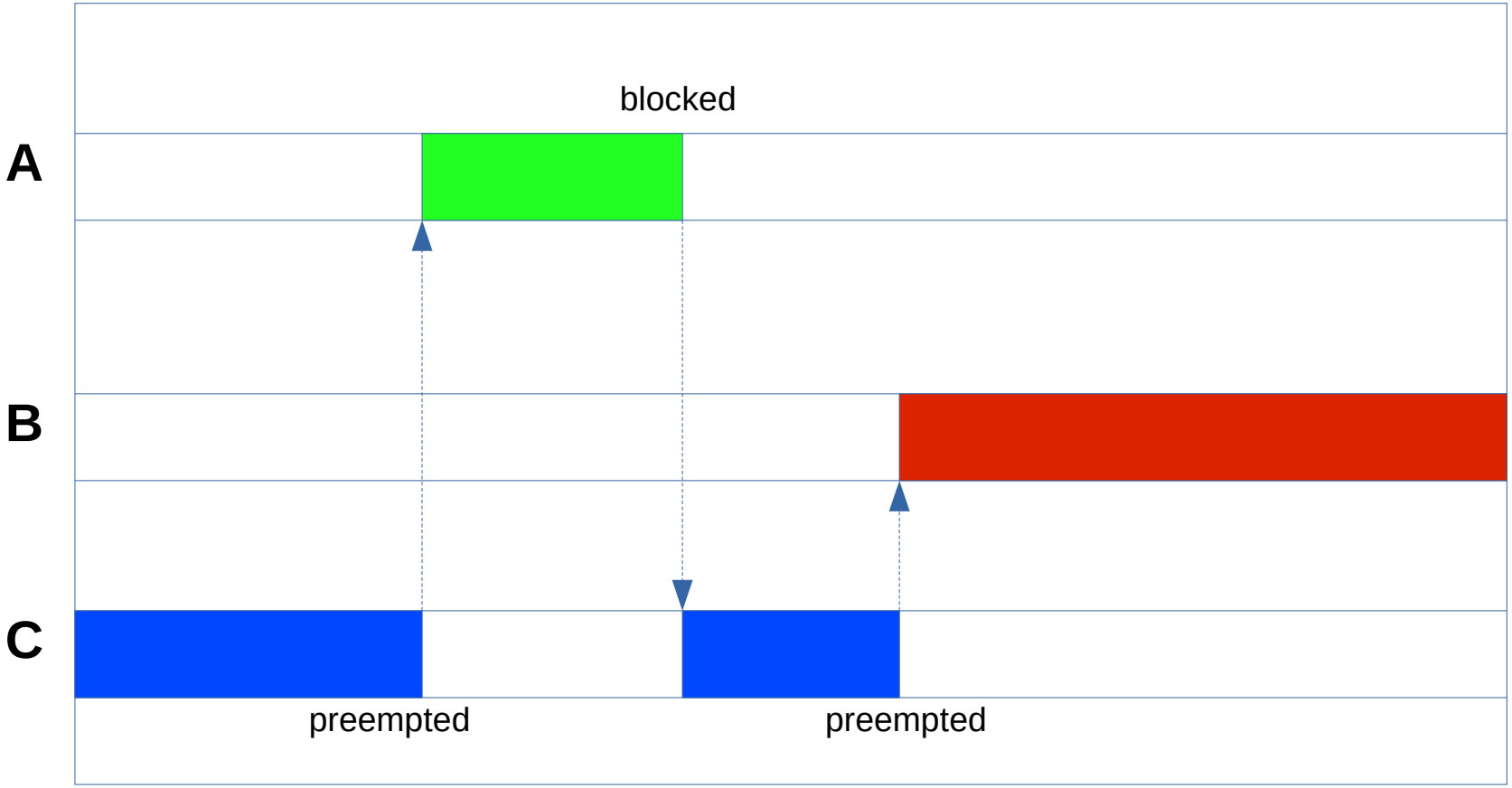
Priority Inheritance

- Prevents Priority Inversion
- Currently only implemented for futex (Fast User-space muTEX)

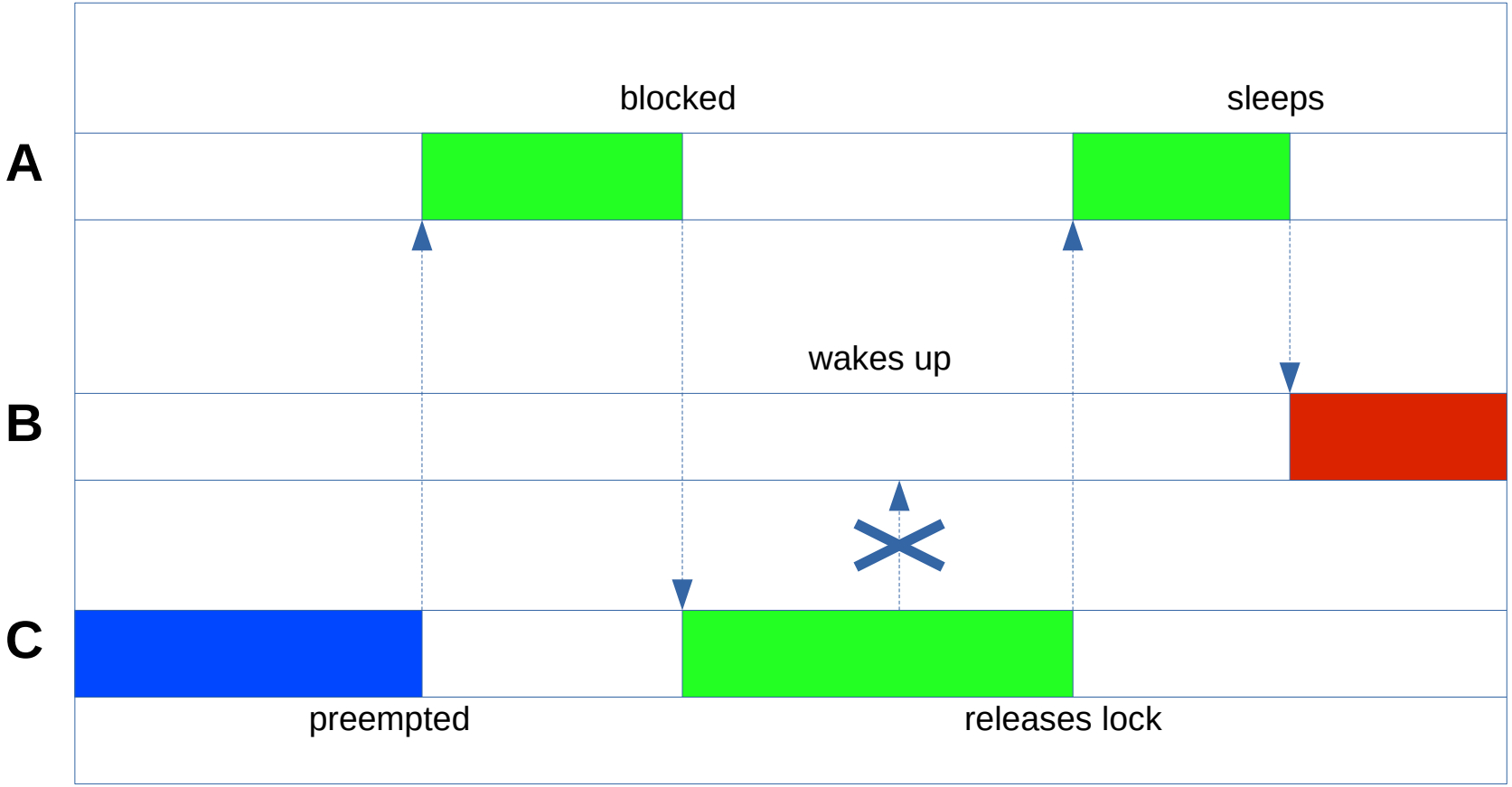
```
pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT)
```

- PREEMPT_RT adds it to spin_locks( ) and mutex_lock( )

Priority Inversion



Priority Inheritance



Enabling PREEMPT_RT

- rw_locks become more like rwsem
  - Sleepable reader / writer locks
- Readers DO NOT HAVE PRIORITY INHERITANCE!
- Writers do inherit priority
  - But they do not boost readers
- Try to avoid rw locks and sems
  - Horrible for cache lines (they do not scale)
  - Use RCU when you can

Sleeping Spin Locks

- The ‘trylock’ issue

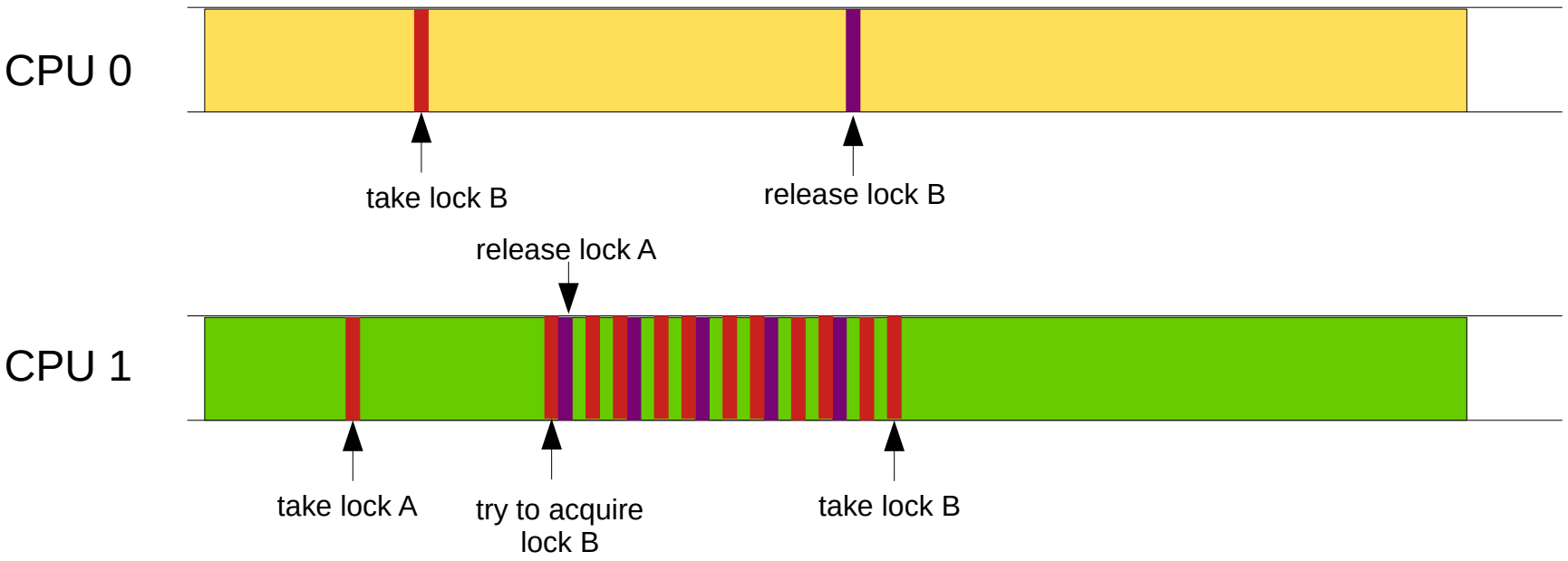
```
again:
   spin_lock(&a);
   [..]
   if (!spin_trylock(&b)) {
      spin_unlock(&a);
      goto again;
   }
```

Sleeping Spin Locks

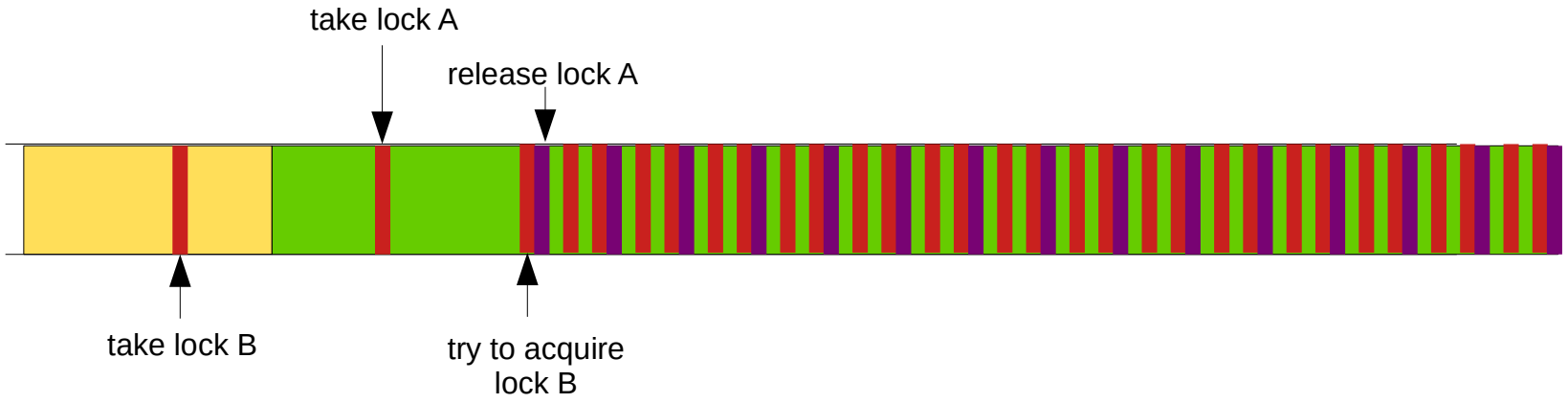
- The ‘trylock’ issue
- Works fine for spinning locks, but not for mutex

```
again:
   spin_lock(&a);
   [..]
   if (!spin_trylock(&b)) {
      spin_unlock(&a);
      goto again;
   }
```

Real Spinning Locks



Sleeping Spin Locks

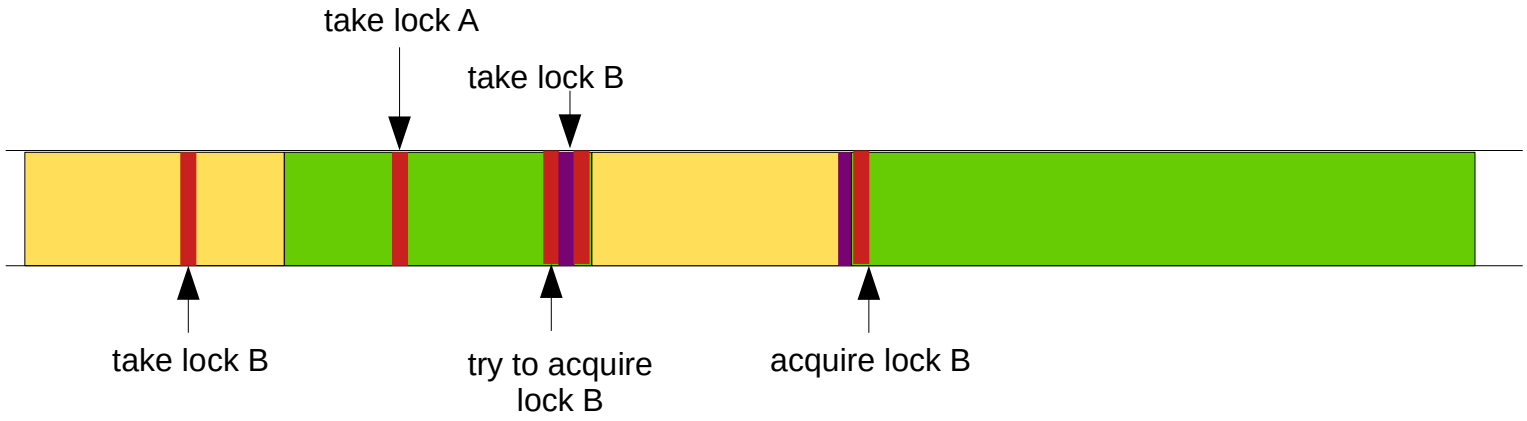


Sleeping Spin Locks

- One solution that works when applicable

```
again:
   spin_lock(&a);
   [..]
   if (!spin_trylock(&b)) {
      spin_unlock(&a);
      spin_lock(&b);
      spin_unlock(&b);
      goto again;
   }
```

Sleeping Spin Locks



Per CPU variables?

- Variables that are only accessed by their associated CPU
- Only need to disable preemption
- Spin Locks no longer disable preemption

Per CPU variables?

- Variables that are only accessed by their associated CPU
- Only need to disable preemption
- Spin Locks no longer disable preemption

```
spin_lock(&mylock);

x = this_cpu_read(myX);
y = this_cpu_read(myY);
z = x + y;
this_cpu_write(myZ, z);

spin_unlock(&mylock);
```

Per CPU variables?

- Variables that are only accessed by their associated CPU
- Only need to disable preemption
- Spin Locks no longer disable preemption
- Spin Locks do disable migration!

```
spin_lock(&mylock);

x = this_cpu_read(myX);
y = this_cpu_read(myY);
z = x + y;
this_cpu_write(myZ, z);

spin_unlock(&mylock);
```

Per CPU variables?

- May be protected by spin locks
- May be protected by preempt_disable( )

  NOT BOTH!

Task A

```
spin_lock(&mylock);

x = this_cpu_read(myX);
y = this_cpu_read(myY);
z = x + y;
this_cpu_write(myZ, z);

spin_unlock(&mylock);
```

Task B

```
preempt_disable();

x = this_cpu_read(myX);
y = this_cpu_read(myY);
z = x + y;
this_cpu_write(myZ, z);

preempt_enable();
```

Per CPU variables?

Task A

```
spin_lock(&mylock);

x = this_cpu_read(myX);
```

schedule

Task B

```
preempt_disable();

x = this_cpu_read(myX);
y = this_cpu_read(myY);
z = x + y;
this_cpu_write(myZ, z);

preempt_enable();
```

schedule

Task A

```
y = this_cpu_read(myY);
z = x + y;
this_cpu_write(myZ, z);

spin_unlock(&mylock);
```

Per CPU variables?

- Preempt Disable is not bad
- If it is short!

```
preempt_disable();

x = this_cpu_read(myX);
y = this_cpu_read(myY);
z = x + y;
this_cpu_write(myZ, z);

preempt_enable();
```

Per CPU variables?

- Preempt Disable is not bad
- If it is short!
- Don’t do THIS!

```
preempt_disable();

x = this_cpu_read(myX);
y = this_cpu_read(myY);

w = kmalloc(sizeof(*z), GFP_ATOMIC);
if (!w)
    goto out;

z = x + y;
this_cpu_write(myZ, z);
w->foo = z;

out:
preempt_enable();
```


Per CPU variables?

- Preempt Disable is not bad
- If it is short!
- DO this!

```
w = kmalloc(sizeof(*z), GFP_KERNEL);
if (!w)
    return;

preempt_disable();

x = this_cpu_read(myX);
y = this_cpu_read(myY);
z = x + y;
this_cpu_write(myZ, z);
w->foo = z;

preempt_enable();
```

Per CPU variables?

- Preempt Disable is not bad
- If it is short!
- DO this!

```
w = kmalloc(sizeof(*z), GFP_KERNEL);
if (!w)
    return;

preempt_disable();

x = this_cpu_read(myX);
y = this_cpu_read(myY);
z = x + y;
this_cpu_write(myZ, z);
w->foo = z;

preempt_enable();
```

Per CPU variables?

- Preempt Disable is not bad
- If it is short!
- Keep slow operations OUT of preempt disable critical sections
  - This is good for PREEMPT_RT

Per CPU variables?

- Preempt Disable is not bad
- If it is short!
- Keep slow operations OUT of preempt disable critical sections
  - This is good for PREEMPT_RT
  - This is good for mainline too!

Disabling interrupts

- Avoid local_irq_save( )
  - Most likely it’s a bug if you are using it
- Use spin_lock_irqsave( ) (or spin_lock_irq( ))
  - They disable interrupts on non PREEMPT_RT
  - They don’t on PREEMPT_RT (that’s what you want!)

Disabling interrupts

- Avoid local_irq_save( )
  - Most likely it's a bug if you are using it
- Use spin_lock_irqsave( ) (or spin_lock_irq( ))
  - They disable interrupts on non PREEMPT_RT
  - They don't on PREEMPT_RT (that's what you want!)

NEVER DO! ...

```
local_irq_save(flags);
spin_lock(&mylock);
```

or

```
spin_unlock(&mylock);
local_irq_restore(flags);
```

softirqs

- Are a real PITA
- They are “raised”
  - Asked to run
- Raised by interrupts
- Raised by tasks

softirqs

- local_bh_disable( ) - and spin_lock_bh( )
  - **disables** preemption non PREEMPT_RT
  - preemption **stays enabled** on PREEMPT_RT
  - migration disabled

softirqs

- Currently (in mainline)
  - Are indiscriminate in what they run
    - No priority between them
    - If one runs for a long time, no other one can run on that CPU
  - No priority between what takes precedence

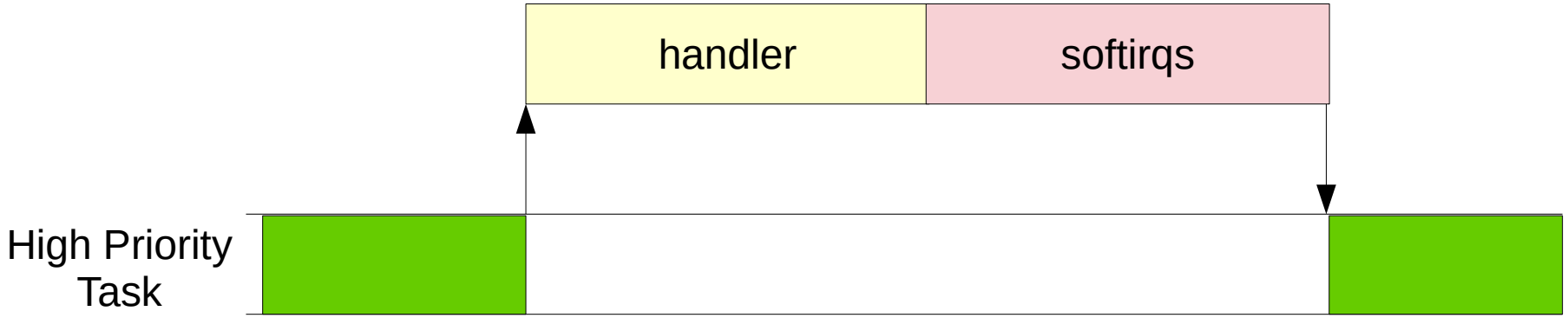
softirqs

- Currently (in mainline)
  - Are indiscriminate in what they run
    - No priority between them
    - If one runs for a long time, no other one can run on that CPU
  - No priority between what takes precedence
- In PREEMPT_RT
  - Runs by who raises them
    - A mask is used
  - local_bh_enable( )
    - Runs the softirqs raised by the task
  - ksoftirq - runs the rest of them

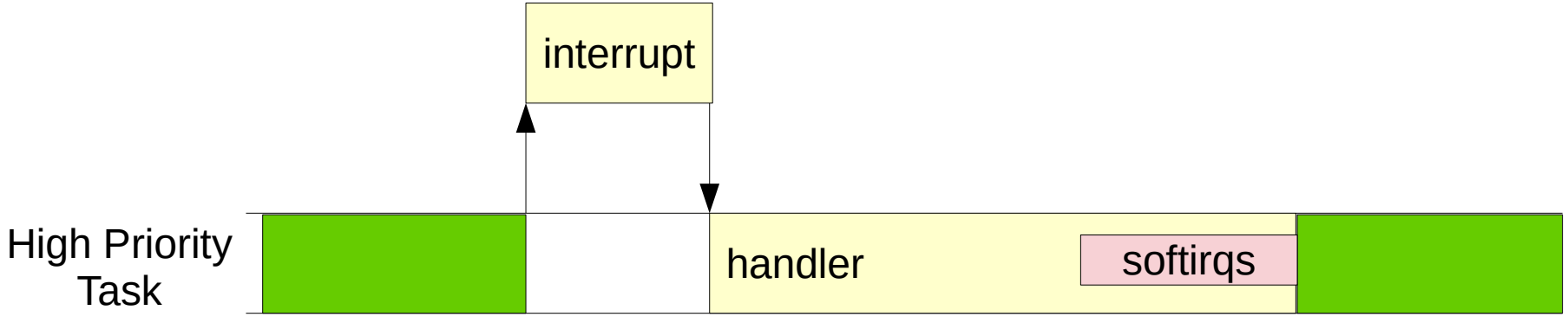
softirqs

- Mainline is currently suffering from softirqs
  - Starvation of one softirq by another
  - Frederic Weisbecker has a patch set out to help
  - Influenced by the work from the PREEMPT_RT patch
  - Using a mask and also allow softirqs to preempt each other

softirqs NON_PREEMPT



softirqs NON_PREEMPT



raw_spin_locks

- raw_spin_lock*( ) is still a spinning lock
  - They were introduced in Linux in 2009 (2.6.33!)
  - They are meaningless in mainline
    - Were added for for PREEMPT_RT only!
- Makes order important
  - Can not call raw_spin_lock( ) followed by spin_lock( )
  - Same as current spin_lock( ) followed by mutex_lock( )

raw_spin_locks

- Used when you definitely CAN NOT SLEEP!
- Don’ t use them just because you can’t “figure it out”
  - “scheduling while atomic”
  - Keep irq disabling short
  - Keep preempt disabling short
- Your lock is not as important as you think it is

Questions?