Coming soon

Thomas Gleixner – Kernel Recipes 2023

Coming soon?

On preempt_model_none() or preempt_model_voluntary() configurations rescheduling of kernel threads happens only when they allow it, and only at explicit preemption points, via calls to cond_resched() or similar. That leaves out contexts where it is not convenient to periodically call cond_resched() -- for instance when executing a potentially long running primitive (such as REP; STOSB.)

This means that we either suffer high scheduling latency or avoid certain constructs.

Define TIF_ALLOW_RESCHED to demarcate such sections.

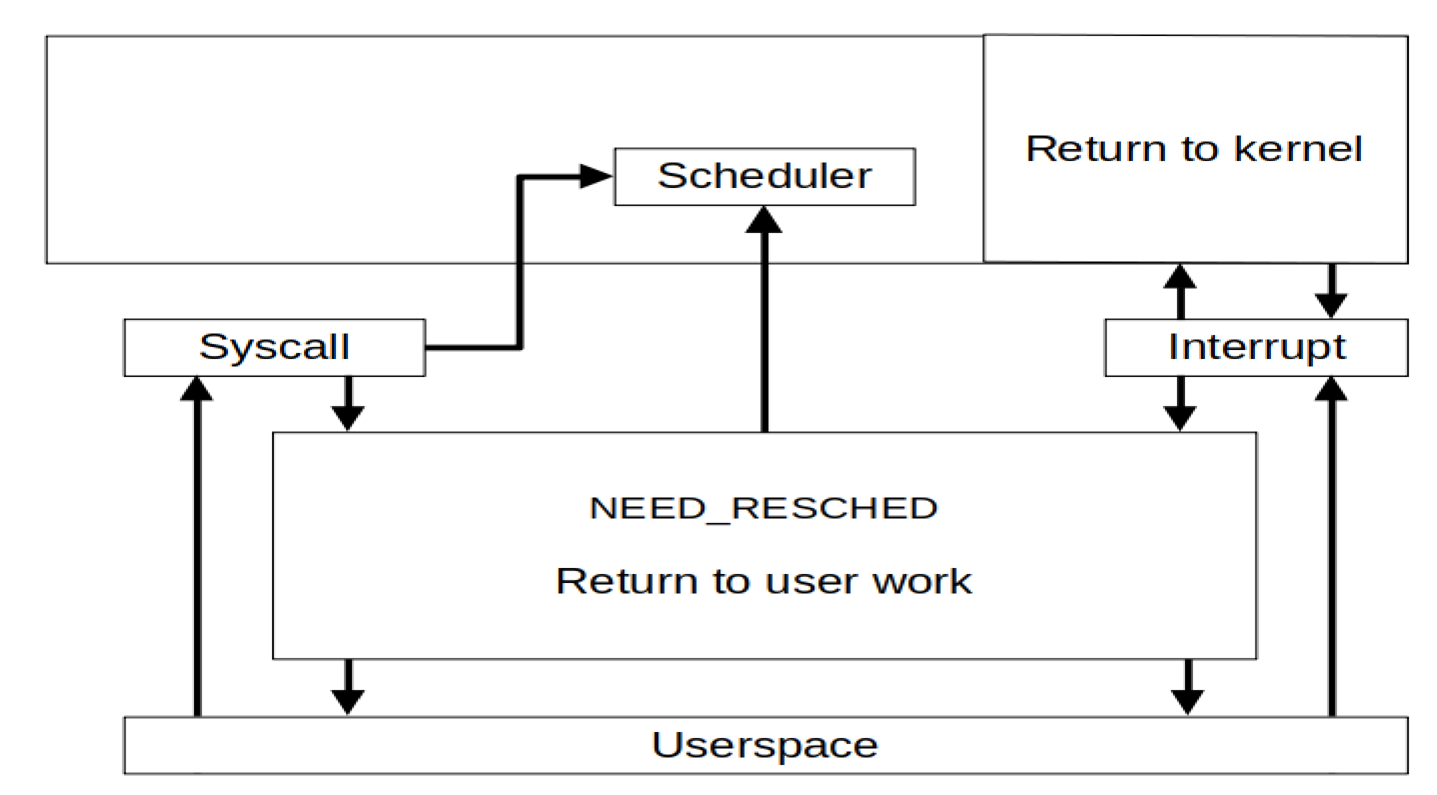
Preemption models

- PREEMPT_NONE
- PREEMPT_VOLUNTARY
- PREEMPT_FULL
- PREEMPT_RT

Preemption model NONE

- Preemptive multitasking in userspace
  - Timeslicing, priority
- Cooperative multitasking in the kernel
- Kernel code runs to completion
  - Preemption point on return to user space
  - Task invokes schedule()

Preemption model NONE



Preemption model NONE

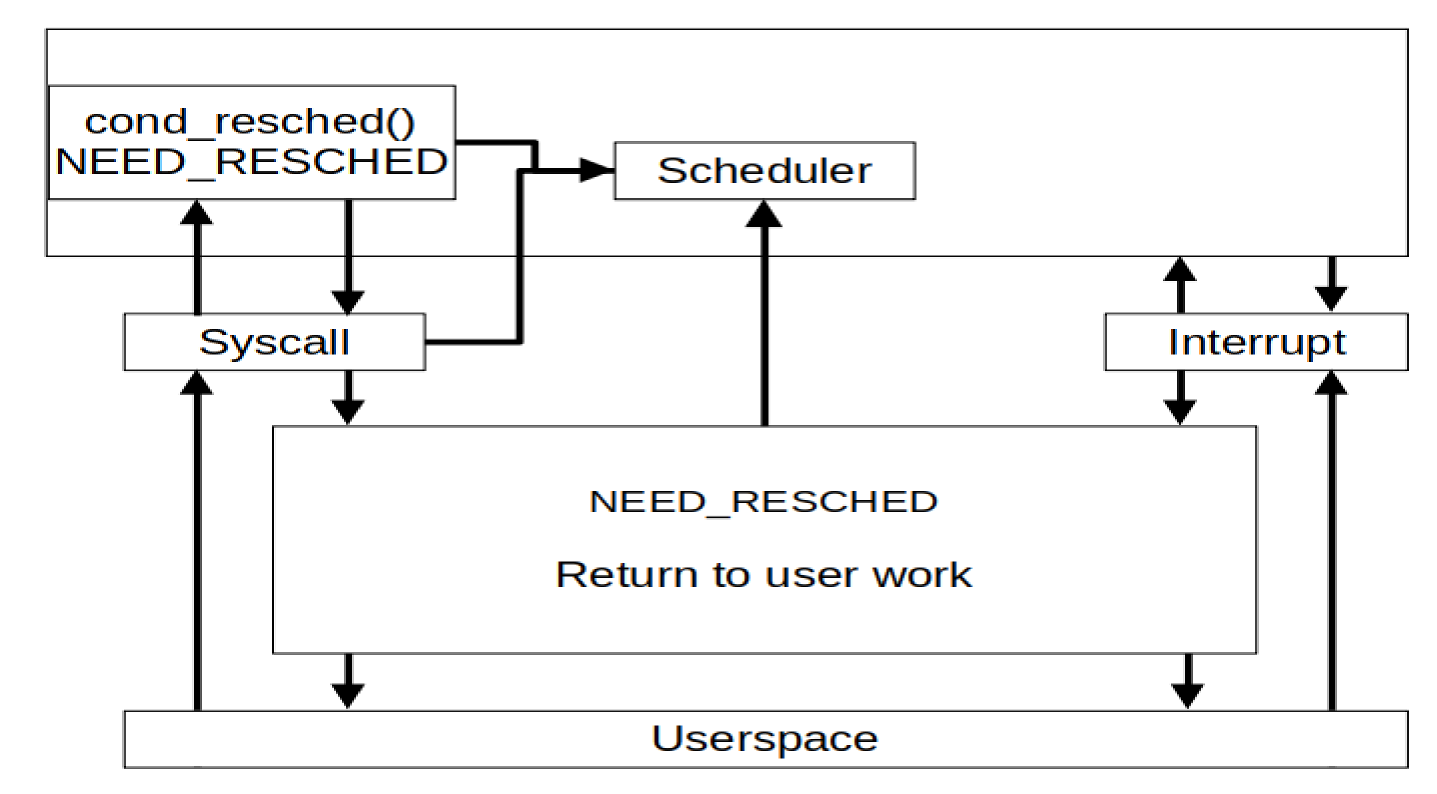
- What could go wrong?
  - Long running tasks can cause latencies
  - Long running tasks can starve the system
- Detectable but no mitigation possible
  - Scheduler has no knowledge whether preemption is safe

Preemption model NONE

- How to prevent latencies and starvation?
  - Manual placement of voluntary scheduling opportunities, i.e. cond_resched()

```
static inline void cond_resched(void)
{
    if (need_resched())
        schedule();
}
```

Preemption model NONE



Preemption model NONE

- cond_resched()

```
for (i = 0; i < limit; i++) {
    process(data[i]);
    cond_resched();
}
```

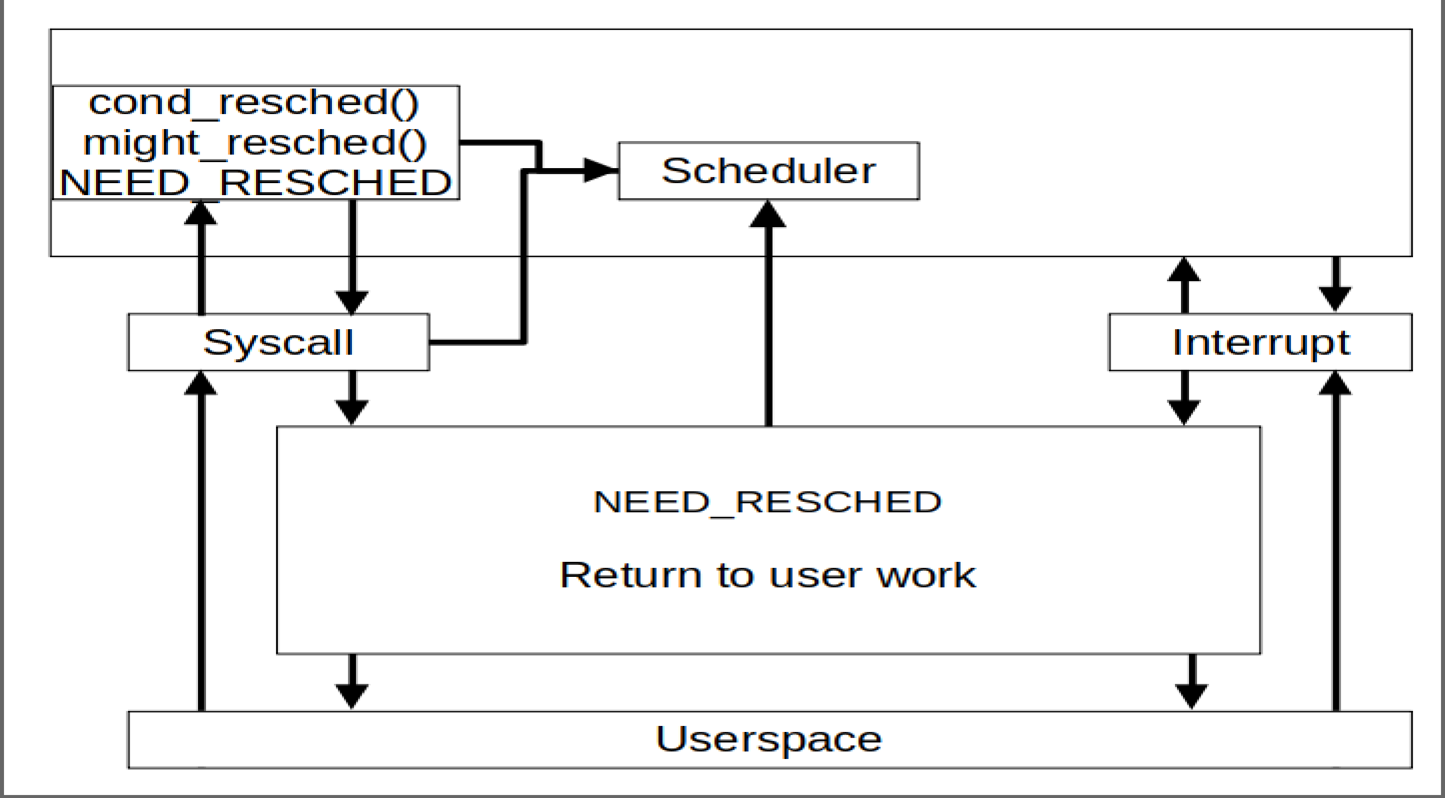
```
for (i = 0; i < limit; i++) {
    mutex_lock(m);
    process(data[i]);
    cond_resched();
    mutex_unlock(m);
}
```

```
for (i = 0; i < limit; i++) {
    mutex_lock(m);
    process(data[i]);
    mutex_unlock(m);
    cond_resched();
}
```

Preemption model VOLUNTARY

- Same properties as NONE
- Additional opportunistic preemption points
  - might_sleep()

Preemption model VOLUNTARY



Preemption model VOLUNTARY

- might_sleep()
  - might_sleep() is a debug mechanism
  - cond_resched() is glued into it
  - Easy to misplace
  - Automatically injected by lock and wait primitives

Preemption model VOLUNTARY

might_sleep()

```
...
wait_for_completion(&c);
return_to_userspace();    ← Preemption point


...
wait_for_completion(c)
    might_sleep()
        cond_resched();     ← Preemption point
    while (!complete(c)
        schedule();
return_to_userspace();    ← Preemption point
```

The embedded cond_resched() can result in redundant task switching

Preemption model VOLUNTARY

might_sleep()

```
mutex_lock(A);
mutex_lock(B);
do_work();
mutex_unlock(B);
mutex_unlock(A);

mutex_lock(A);
mutex_lock(B)
    might_sleep()
        cond_resched();       ← Preemption point
```

The embedded cond_resched() can result in redundant task switching and lock contention on mutex A.

Preemption model VOLUNTARY

- Provides better latencies than NONE
- Otherwise the same issues as NONE
- More contention possible

Preemption model FULL

- Full preemptive multitasking
  - Timeslicing, priority
  - Restricted in non-preemptible kernel code sections

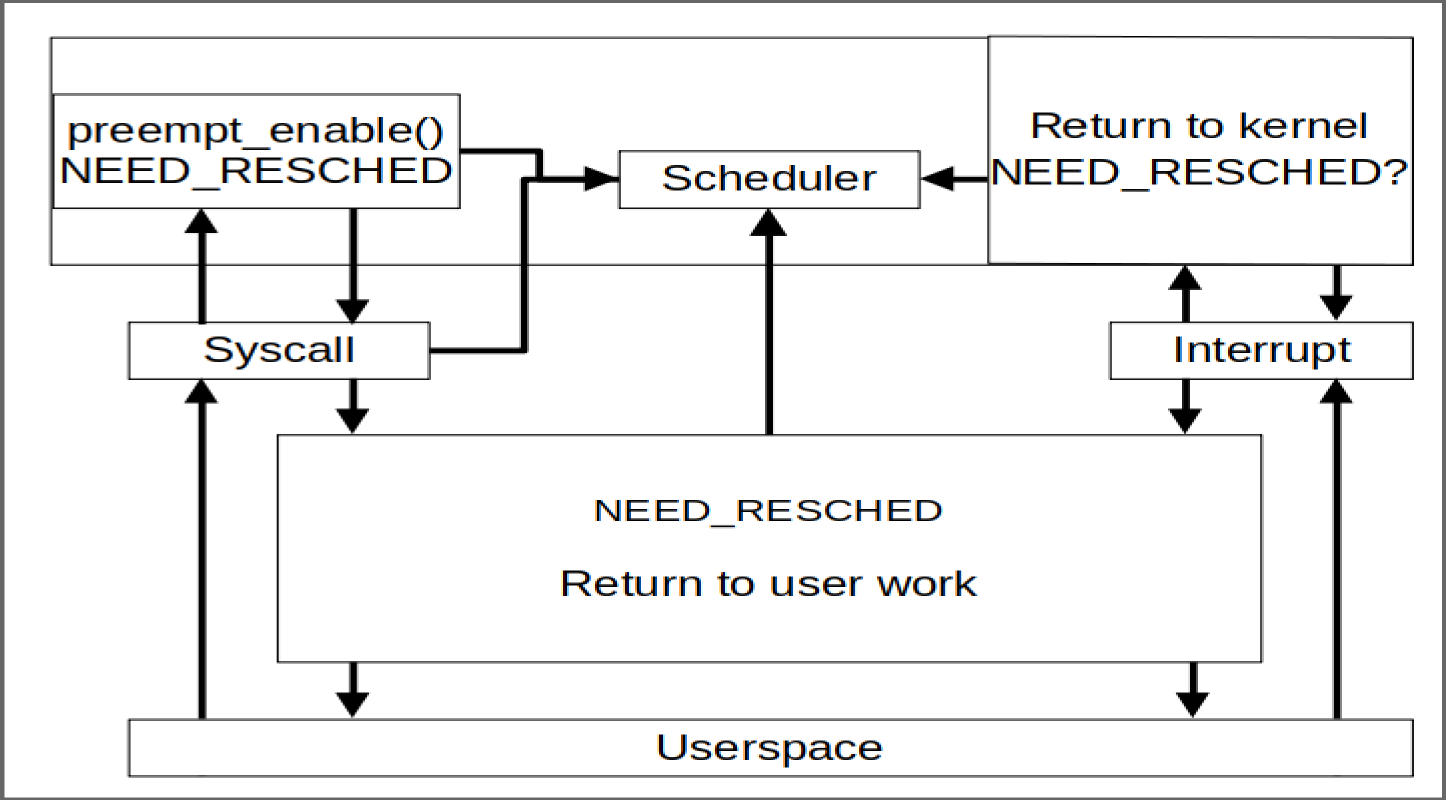
Preemption model FULL

- Implicit non-preemptible kernel code sections
  - [spin|rw]locks are held
  - [soft]interrupts and exceptions
  - local_irq_disable(), local_bh_disable()
  - Per CPU accessors
- Explicit non-preemptible kernel code sections
  - preempt_disable()

Preemption model FULL

- Non-preemptible sections
  - Prevent preemption
  - Prevent migration
  - No blocking operations allowed
- Migration prevention can be made preemptible
  - migrate_disable()

Preemption model FULL



Preemption model FULL

- Scheduler knows when preemption is safe
  - Reduced latencies
  - Agressive preemption can cause contention
  - Tradeoff versus throughput

Preemption model RT

- Full preemptive multitasking
  - Preemption model is the same as FULL
- RT further reduces non-preemtible sections
  - [spin|rw|local]locks become sleeping locks
  - Most interrupt handlers are force threaded
  - Soft interrupt handling is force threaded

Preemption model RT

- Further restrictions for non-preemptible sections
  - No memory allocations or other functions which might acquire rw/spinlocks as they are sleepable in RT
- Same benefits and tradeoffs as FULL, but:
  - Smaller worst case latencies
  - More tradeoff versus throughput

Preemption model RT

- The throughput tradeoff
  - Affects usually non-realtime workloads
  - Caused by overeager preemption and the resulting lock and resource contentions

Preemption model RT

- Mitigating the throughput tradeoff
  - LAZY preemption mode for non-RT tasks
  - lock held sections disable lazy preemption
  - Still can be force preempted by the scheduler

Preemption model NONE/VOLUNTARY woes

- X86 REP MOV/STO for memcpy()/set()
  - Very efficient
  - Can be interrupted, but NONE and VOLUNTARY cannot preempt
  - Large copies/clears cause latencies
  - Chunk based loop processing required with cond_resched() which fails to utilize hardware

Preemption model NONE/VOLUNTARY woes

- Proposed solution: TIF_ALLOW_RESCHED
  - Wrapped in allow_resched() and disallow_resched()
  - Annotate sections which are safe to preempt on NONE and VOLUNTARY

https://lore.kernel.org/lkml/20230830184958.2333078-8-ankur.a.arora@oracle.com

Preemption model NONE/VOLUNTARY woes

- Seriously?
  - cond_resched(), might_sleep(), preempt_disable(), preempt_enable(), allow_resched(), disallow_resched()
  - The reverse semantics of preempt_disable() and allow_resched() are just bad

Let’s take a step back

- The goal is to avoid preemption on NONE and VOLUNTARY
- Preemption on time slice exhaustion should be enforcable even on NONE and VOLUNTARY
- NONE and VOLUNTARY do not know about preemption safety

Let’s take a step back

- Preempt counter is not longer expensive
- Usually enabled anyway due to dynamic preemption model switching
- All preemption models can know when preemption is safe

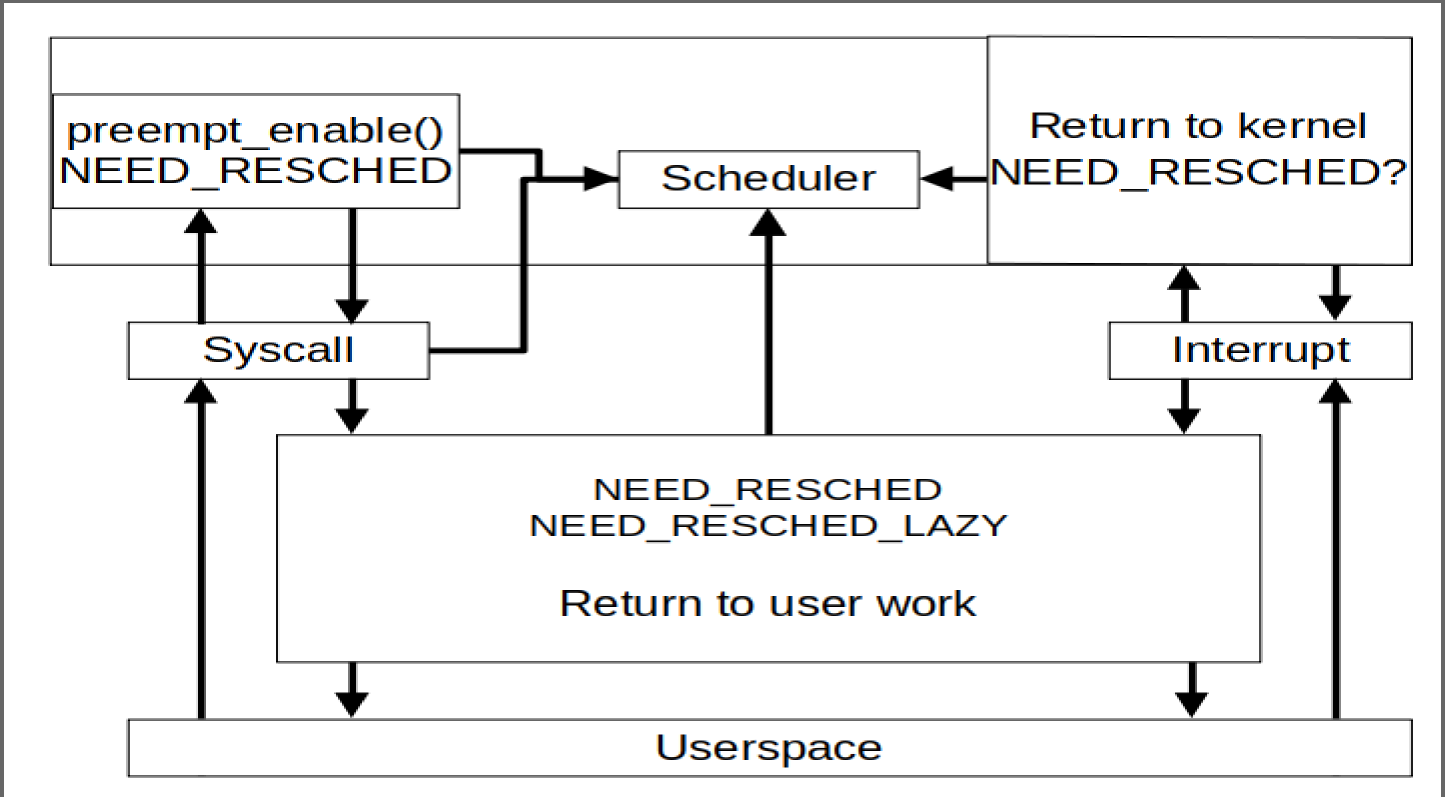
Preemption model reduction

- Enforce preempt counter enablement
- Provide lazy preemption similar to RT
  - TIF_NEED_RESCHED_LAZY
  - Lazy preemption only on return to userspace
- Enforced preemption: TIF_NEED_RESCHED

Preemption model reduction

- NONE/VOLUNTARY: TIF_RESCHED_LAZY used for SCHED_OTHER
- Timeslice exhaustion enforces preemption with TIF_NEED_RESCHED
- FULL: Switch SCHED_OTHER to TIF_NEED_RESCHED

Preemption model reduction



Preemption model reduction

- Gives full control to the scheduler
  - VOLUNTARY semantics can be handled in the scheduler itself
- Allows to remove cond_resched()
- Avoids new ill defined annotations
  - Eventually proper hinting required
- Can be utilized for RT with minimal effort

Preemption model reduction

Scheduler hints for lazy preemption

- If required must be scope based
- Proper nesting
- Embeddable into locking primitives

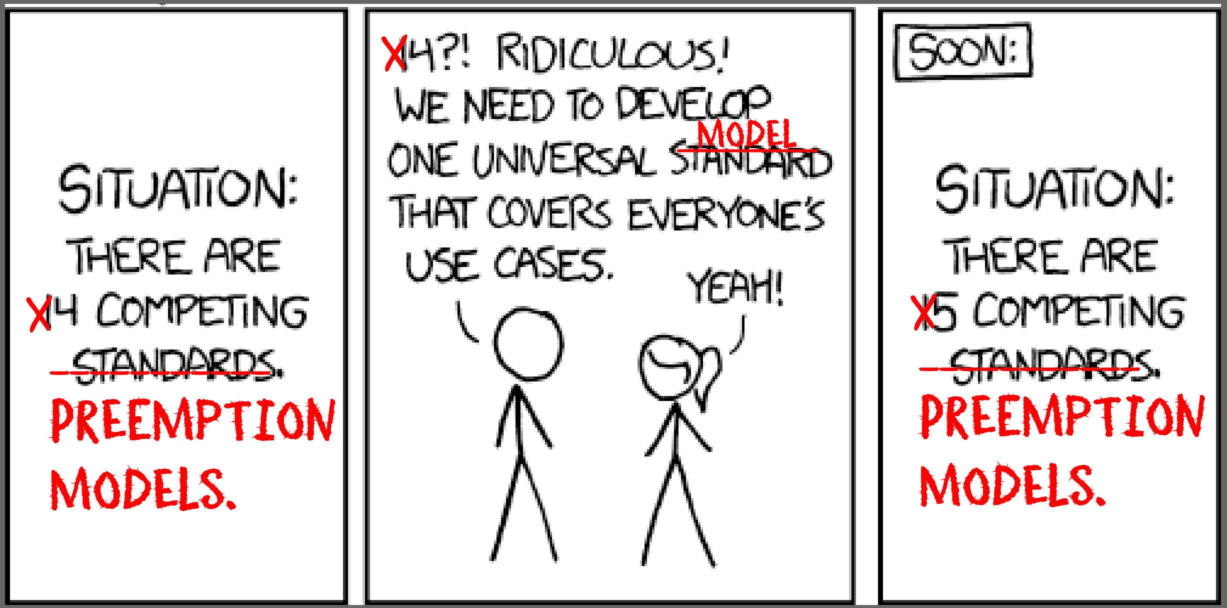
```
preempt_lazy_disable();    // Please avoid preemption
do_prep();
do_stuff()
   mutex_lock(m)
      preempt_lazy_disable();
   …
   mutex_unlock(m)
      preempt_lazy_enable();
preempt_lazy_enable();     // Now its fine to preempt
```

Preemption model reduction

- One preemption model with runtime switching solely at the scheduler level
- RT still separate and compile time selected
- PoC works and looks promising.
- A few museum architectures in the way.

https://lore.kernel.org/lkml/87jzshhexi.ffs@tglx/

Coming soon?



https://xkcd.com/927/