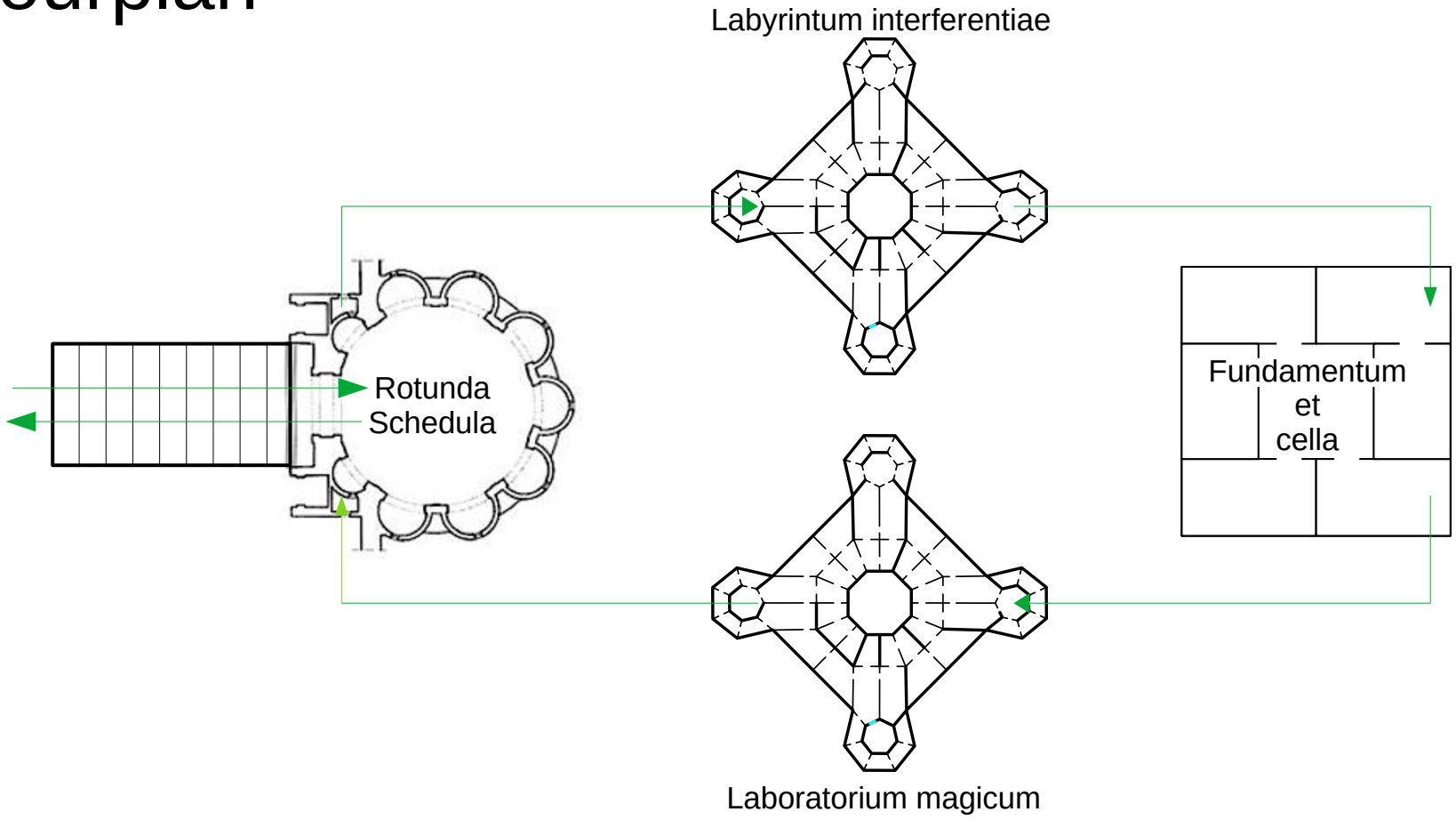
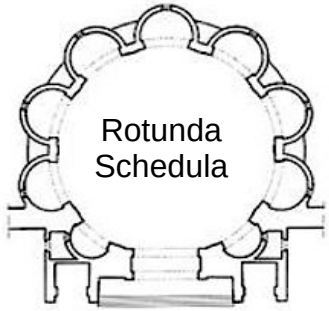
ELISA May 2021 special: A guided tour through the Preempt-RT castle

Tour guide: Thomas Gleixner

Tourplan

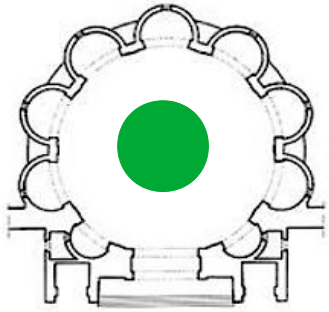

Labyrintum interferentiae
Rotunda
Schedula
Fundamentum
et
cella
Laboratorium magicum

Rotunda Schedula



- The centerpiece of the castle
- The nicest place of the castle
- Each niche is a wonderful place to look at

Rotunda core



- The central scheduling decision function
- Picks the most eligible task to run on a CPU
- Invocation is voluntary or involuntary

schedule()

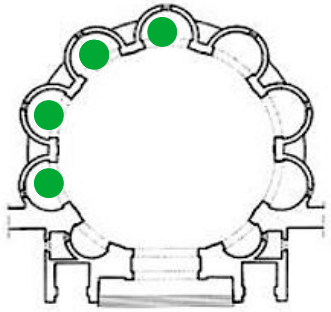
Voluntary invocation

- Task waits for an event
- Task waits for a resource

Involuntary invocation

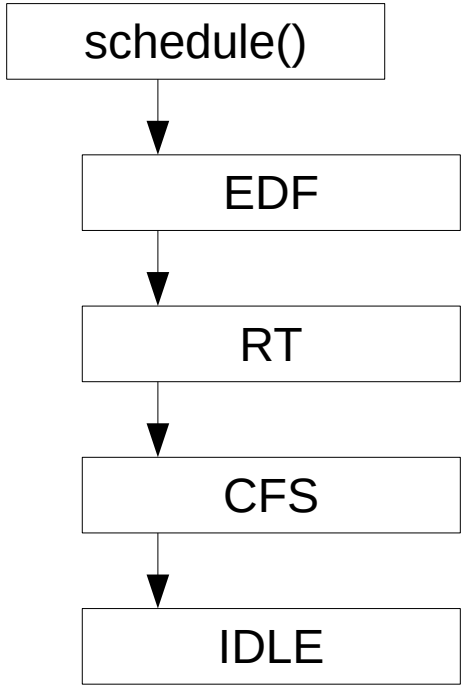
- A more eligible task is available, aka. preemption
- The timeslice of the task is exhausted

Rotunda niches

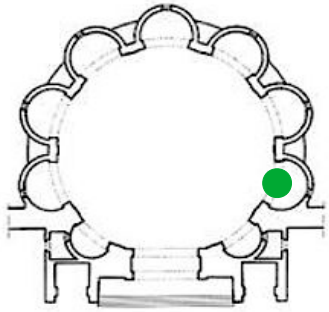


Scheduling class	Policy
Early Deadline First (EDF)	SCHED_DEADLINE
Realtime (POSIX RT)	SCHED_RR, SCHED_FIFO
Completely fair (CFS)	SCHED_OTHER, SCHED_BATCH, SCHED_IDLE
Idle	N/A

Scheduling decision order



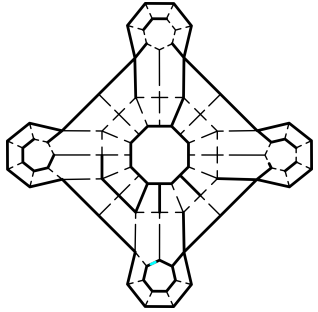
Rotunda niches



SMP load balancing

- Distribute runnable tasks to ensure CPU utilization
- Relevant for performance, latencies and power management
- Various balancing points (idle, task activation, task deactivation)
- Push and pull mechanisms

Labyrinthus interferentiae



A large collection of interference sources

- Exceptions
- Interrupts
- Softinterrupts
- Interrupt disable
- Preemption disable
- Concurrency controls
- Resource allocation
- ...

Preemption disable

Prevents the scheduler from preempting the task

Side effects:

- The task cannot move to a different CPU
- CPU hotunplug is prevented

Usage:

- Lightweight concurrency control between tasks on the same CPU
- Common pattern: Protect per CPU variables

Preemption disable

Semantical issue:

- The protection scope is not specified
- Acts like a per CPU big kernel lock

Realtime specific issues:

- Can cause large unbound latencies
- Used under the hood by other mechanisms

Interrupts disable

Prevents interrupt delivery to the CPU

Side effects:

- Implies preemption disable

Usage:

- Protection against interrupts in critical sections

Interrupts disable

Semantical issue:

- The protection scope is not specified
- Acts like a per CPU big kernel lock

Realtime specific issues:

- Can cause large unbound latencies
- Used under the hood by other mechanisms

Exceptions

Different classes of exceptions:

- Debug exceptions
- Error catching exceptions
- Fault handling

Debug exceptions

- User/admin controlled
- Therefore „harmless“ vs. Realtime behaviour

Error catching exceptions

- Hardware malfunction
  - Machine check exceptions
  - Can be fatal
- Software malfunction
  - Division by 0
  - Undefined opcodes
  - …
- Realtime is the least of the problems if those happen

Fault handling exceptions

- Mostly related to memory management (User space mappings)
- Depending on the fault type (minor, major) the impact can be significant

- Mitigation possible through careful design and setup of the realtime application through existing interfaces and mechanisms.

- Virtualization related faults are similar, but harder to mitigate

Interrupts

- Interprocessor Interrupts (IPI)
- Device Interrupts

- Can have long running interrupt handlers which introduce unbound latencies

Soft-Interrupts

- Execution:
  - On return from Interrupt with interrupts enabled
  - From a dedicated kernel thread
- Side effects:
  - Softinterrupt processing implicitly disables preemption
  - Softinterrupt disable implicitly disables preemption
- Realtime issues:
  - Softinterrupt processing and disabling can cause unbound latencies

Concurrency controls

- Two main flavours:
  - Blocking locks
  - Spinning locks

Blocking locks

- Types
  - Counting semaphore
  - Reader/Writer semaphore
  - Per CPU Reader/Writer semaphore
  - Mutex
  - WW-Mutex
  - RT-Mutex

- Realtime issues:
  - All except RT-Mutex can lead to priority inversion which can cause unbound latencies

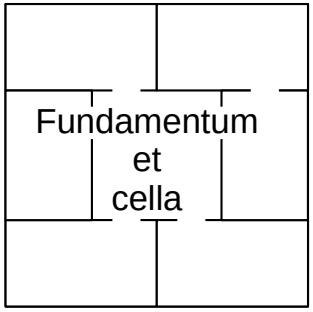
Spinning locks

- Types
  - Spin locks
  - Reader/Writer locks
- Side effects:
  - Implicitly disable preemption
  - Depending on context the lock function must disable soft interrupts or hardware interrupts
- Realtime issues:
  - Disabling preemption and interrupts can cause unbound latencies

Resource allocations

- Depending on the resource type, e.g. memory, allocations can cause unbound latencies
- User space has mechanisms to mitigate by preallocating and locking memory.
- Kernel side allocations in latency sensitive or atomic regions require code changes.

Fundamentum et cella



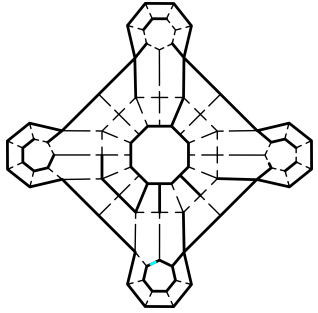
The foundation:

- Low level entry code
- Low level exception and interrupt handling
- Low level CPU and memory management
- ...

The horror cabinets in the cellar:

- Badly designed code
- Layering violations
- Performance optimizations
- …

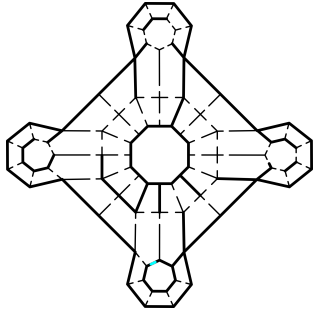
Laboratorium magicum



The place where the (not so) magic mechanisms have been invented to mitigate the realtime issues which are exhibited in Labyrinthum interferentiae and in Fundamentum et cella.

It‘s not a coincidence that the layout of the laboratory is the same as the layout of the labyrinth.

Laboratorium magicum



The trivial (or maybe not so trivial) mitigations:

- Force hard interrupt handling into thread context so it becomes scheduler controlled
- Force soft interrupt handling into thread context so it becomes scheduler controlled
- Enhance blocking lock mechanisms which can lead to priority inversion with support for priority inheritance

Enforced interrupt threading

- Trivial for regular device interrupts, but...
- Not applicable for IPIs and the per CPU timer interrupt

Enforced interrupt threading of device interrupts

- Only the first step for solving the problem
- The preemption disable nature of disabling softinterrupts around the handler invocation does not magically go away

IPIs and per CPU timer interrupt

- Need deeper inspection

- Possible mitigations:
  - Splitting out functionality into different contexts, e.g. POSIX CPU timer signal handling
  - Avoid expensive IPIs completely and implement the required functionality differently

Enforced soft interrupt threading

- Logical consequence of forced interrupt threading
- Do not allow softirq processing on return from hard interrupt

- Does not solve the problem that softirq processing disables preemption

- Forcing all soft interrupt processing into ksoftirqd context can have performance impact for non-RT workloads.

Enhance blocking locks with priority inheritance

- Provide new RT-Mutex based implementations
- Trivial and straightforward for MUTEX
- All other blocking lock variants need more thought

Counting semaphores

- No strict owner semantics
- Cannot support priority inheritance
- Most usage is historical and has been replaced
- Left unmodified

Reader/writer semaphores

- Full PI support would require multi-reader inheritance
- PI is only supported when write locked
  - As a consequence it becomes writer unfair
  - Workloads which suffer from writer unfairness are not typical realtime workloads

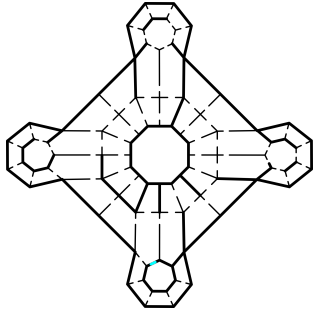
Per CPU reader/writer semaphores

- No PI support possible
- Usage is not really realtime sensitive, e.g. CPU hotplug locking. CPU hotplug is a latency source by itself.

WW-Mutex

- Non-deterministic by design
- PI support might be possible, but does not really make sense
- Main usage in graphics drivers

Laboratorium magicum



The next level of mitigations:

- Substitute spinning locks

Spinning locks

Spinlock

- Usage in low level management code requires the existing preemption/interrupt disable semantics
- Usage in other areas can be substituted

Reader/Writer lock

- No usage in low level management code

Spinlock

Seperate low level usage and general usage:

- raw_spinlock
- spinlock

raw_spinlock

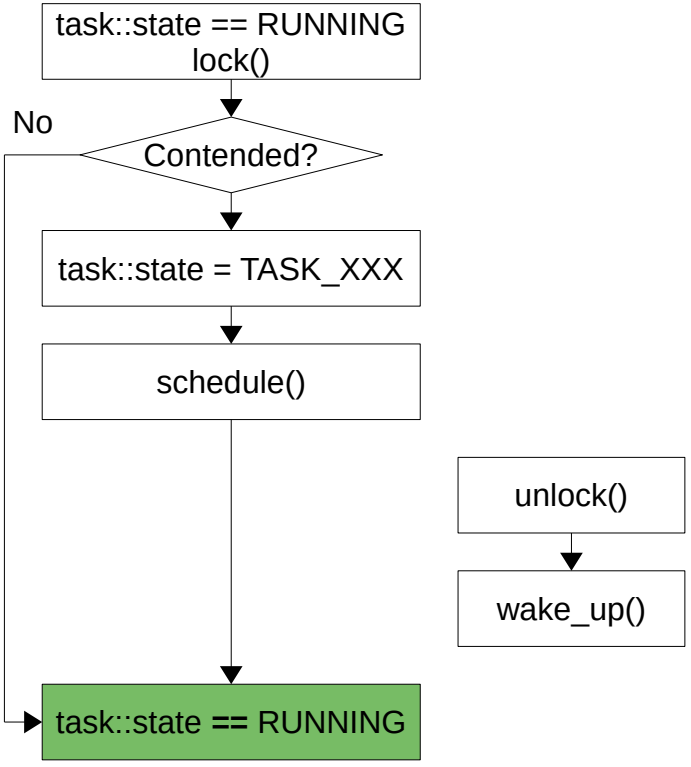
- Preserve the spinning, preempt disabling and eventually interrupt disabling semantics

spinlock

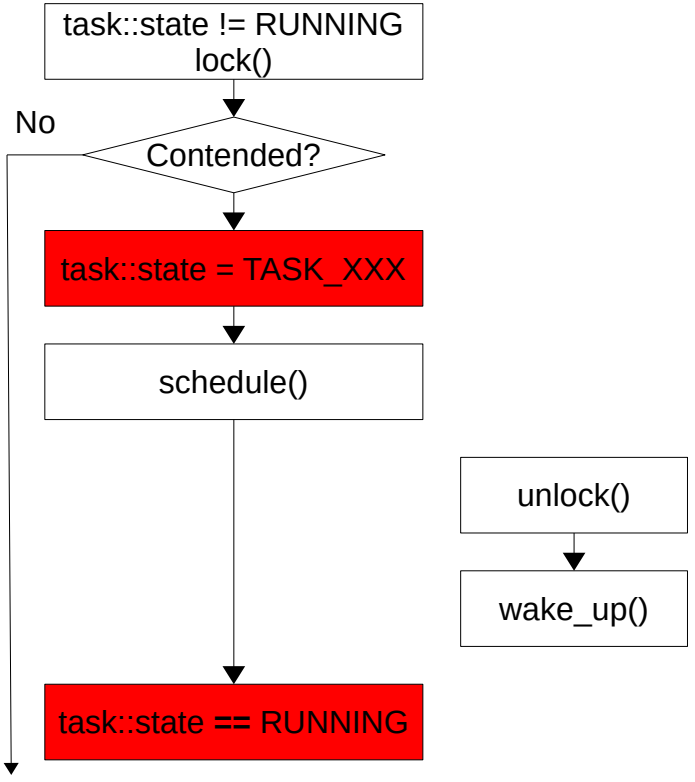
- Mapped to raw_spinlock for !RT kernels
- Substituted with a RT-Mutex based implementation for RT

Spinlock substitution – Twist #1

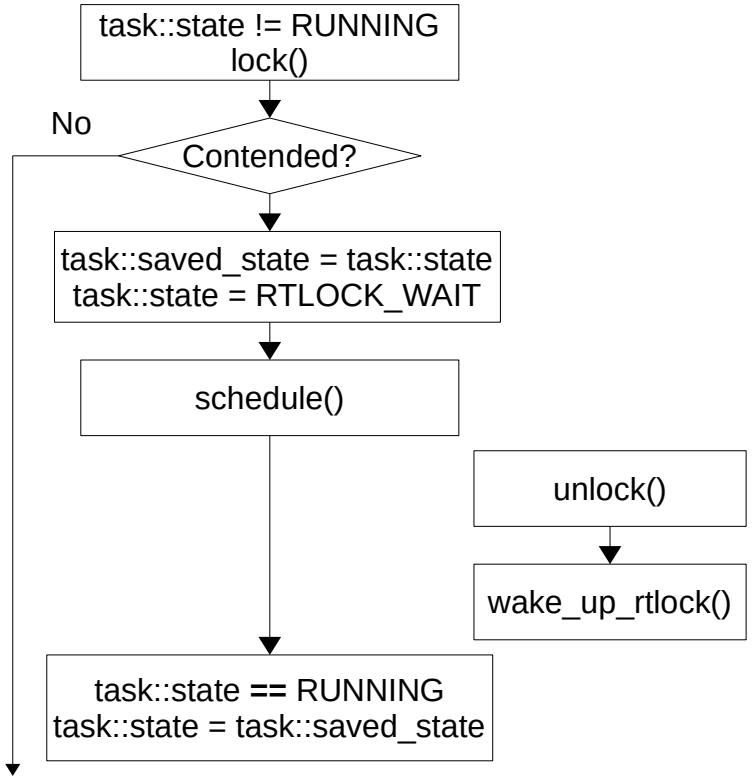
Blocking lock

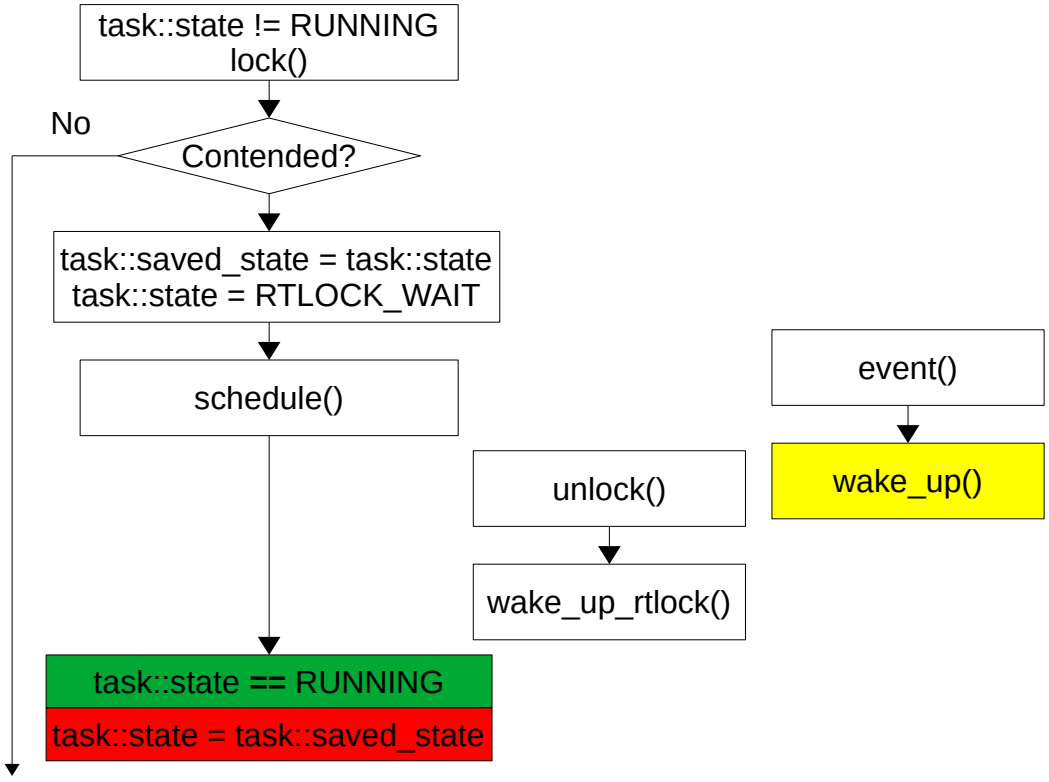


Spinning lock

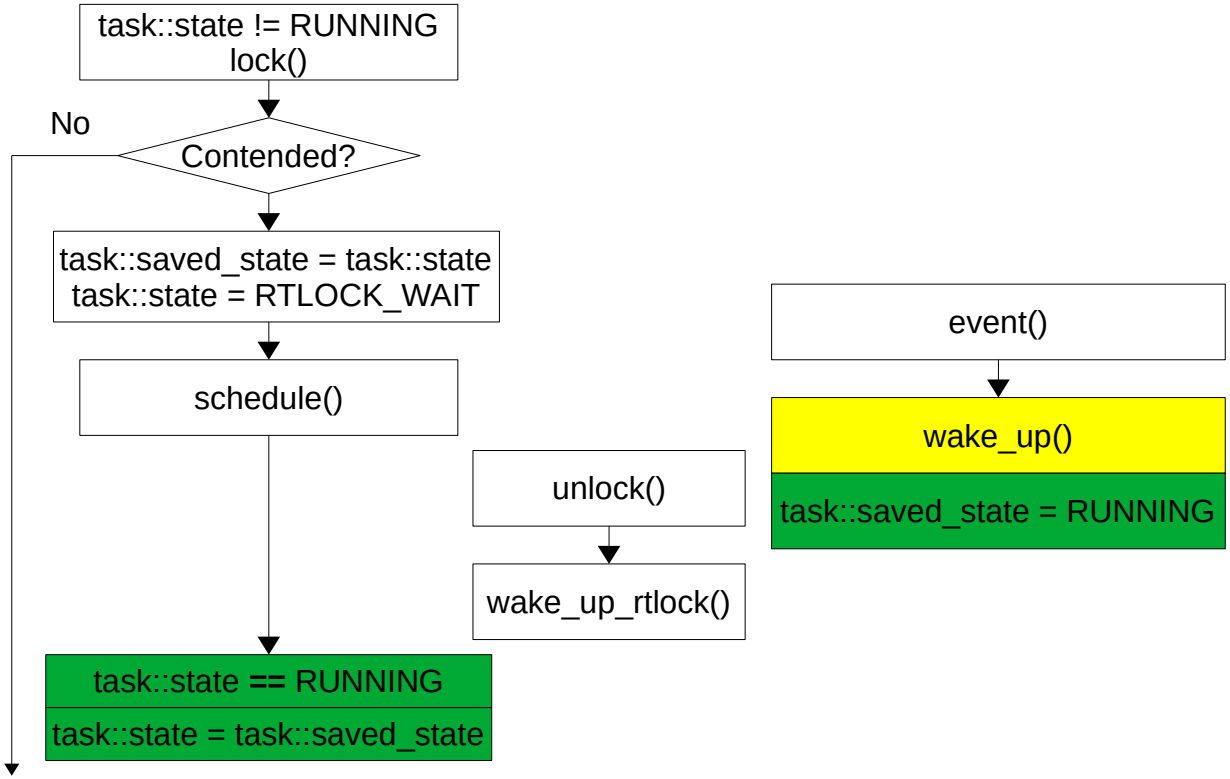


Spinlock substitution – Twist #1





Spinlock substitution – Twist #1



Spinlock substitution – Twist #2

Spinlocked section is now preemptible

- Scheduler can migrate task to a different CPU: FAIL

Spinlocked sections guarantee that the task cannot migrate

- Required for per CPU data correctness

Solution: Disable migration for spinlock held sections

Migration disable

The obvious but not so popular solution:

- Has an impact on schedulability
- Not well studied in scheduling theory

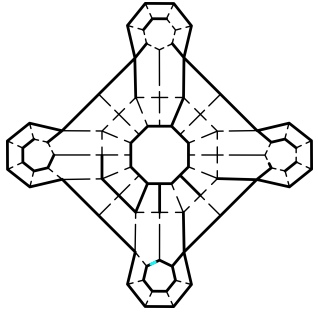
Useful for other purposes:

- kmap_atomic() to kmap_local() conversion (already upstream)
- Other RT mechanisms

Reader/writer locks

- No split into raw_rwlock and rwlock required
- Full PI support would require multi-reader inheritance
- PI is only supported when write locked
  - As a consequence it becomes writer unfair
  - No writer starvation observed so far as many of the critical use cases have been replaced with RCU based solutions

Laboratorium magicum



<u>More magic mitigations:</u>

- Make soft interrupts preemptible
- Make threaded interrupts preemptible

Soft interrupt preemption

- Distangle soft interrupt serialization from preempt_count

- Use a per CPU lock with recursion support to handle nested local_bh_disable()
- Disable migration inside local_bh_disable() sections to preserve !RT semantics

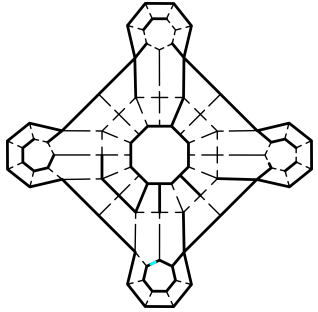
Challenge:

- Handle early boot correctly where interrupts and/or preemption are disabled

Threaded interrupt preemption

- Run with interrupts enabled – no interrupt nesting possible
- Lock based soft interrupt serialization allows preemption

Laboratorium magicum



<u>The remaining pain points:</u>

- Standalone usage of preempt/interrupt disable
- Nesting of ‚sleeping' spinlocks
- Trylock loops and spinwait

Standalone preempt/interrupt disable

Analyis of usage sites required:

- Valid usage in low level management code
- No impact by small well confined critical sections
- Open coded interrupt disable + spinlock instances
- ...

Standalone preempt/interrupt disable

Local lock to the rescue

local_lock() is a strict per CPU lock construct

- Clearly defined protection scope
- Mapped to preempt/interrupt disable on !RT
- Provides lockdep coverage even on !RT
- RT uses a per CPU spinlock (sleeping variant)

Spinlock nesting issues

Substituted spinlocks can end up in preemption/interrupt disabled sections.

- Deep inspection required

Solutions:

- Code rework
- spinlock to raw_spinlock conversions
- New mechanisms, e.g. simple wait, to squash classes of problems

Trylock and spinwait loops

Trylock loops:

- Used to avoid lock order inversion

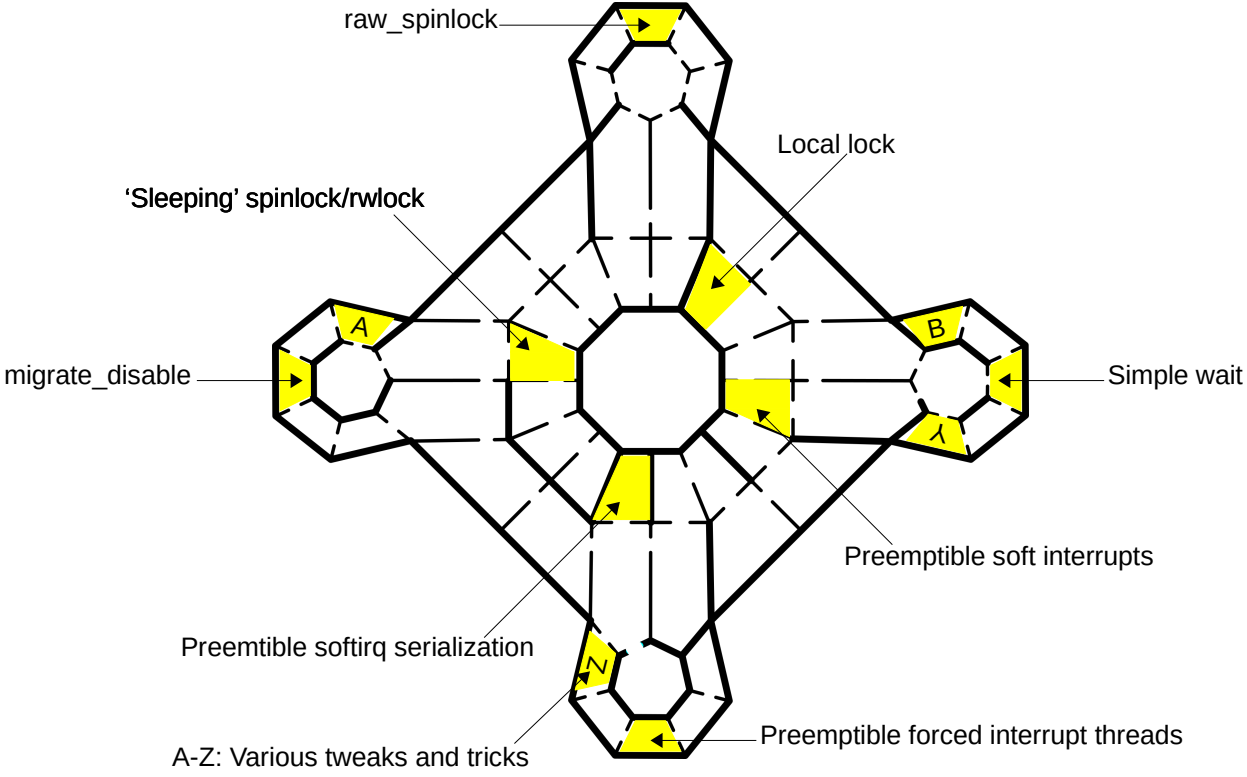
Spinwait loops:

- Used to busy wait for completion of a critical section or operation on a different CPU on !RT kernels

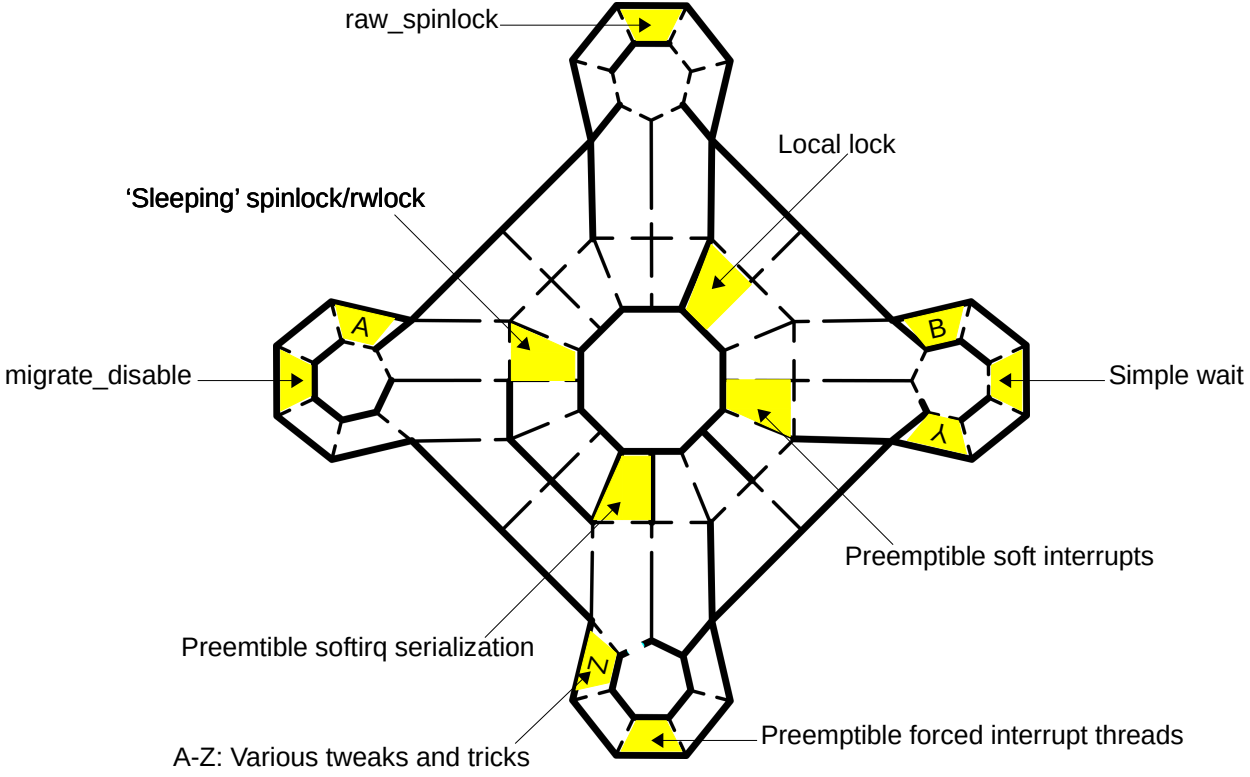
Solutions:

- Code rework
- New mechanisms, e.g. timer expiry locks, to squash classes of problems

Laboratorium magicum



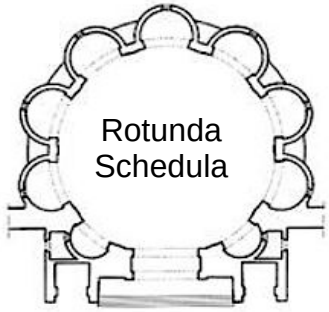
Laboratorium magicum



Challenges:

- Multiple and circular dependencies of mitigation mechanisms
- Unclear semantics in non-RT kernels due to implicit protections and unspecified protection scopes

Rotunda Schedula



With all that in place the scheduler has now maximum control over the CPU, but you should beware of the dragons...

Castle maintenance

- Funding is and always was a challenge
- Development and maintenance is currently stalled due to a funding gap
- Gap needs to be closed to ensure mainline integration

For further information please contact:

Kate Stewart <kstewart@linuxfoundation.com>

End of tour

Questions?

Questions captured

Q: What about firmware calls, e.g. EFI

A: Nothing the kernel can do about. It‘s a configuration and permission problem. The universal rule of UNIX: root can shoot itself in the foot.

Realtime systems have to be designed and audited as a whole. Just using a Realtime kernel does not make a realtime system.

Questions captured

Q: Are the locking rules documented?

A: Yes

https://www.kernel.org/doc/html/latest/locking/locktypes.html

https://www.kernel.org/doc/html/latest/locking/seqlock.html

Q: Is there a single comprehensive slide to explain all the rules a kernel developer should have in mind?

A: Not really – but see the next one (made after the talk)

The ultimate rule for kernel development

Use your brain!

References:

Title:
- https://commons.wikimedia.org/wiki/File:Neuschwanstein_Castle_LOC_print_rotated.jpg
- Author: Unknown
- License: Public domain

Rotunda:
- Sketch by Michelozzo, Manetti, Alberti, ~1440
- License: Public domain

Labyrinth:
- https://commons.wikimedia.org/wiki/File:Labyrinthus_Aedificium.svg
- Author: https://commons.wikimedia.org/wiki/User:Fulvio31
- License: Creative Commons Attribution-Share Alike 4.0 International
  https://creativecommons.org/licenses/by-sa/4.0/deed.en
- Modified by: Thomas Gleixner