Red Hat
Enterprise Linux

Kernel Protocol Verifier

Using the kernel verifier to verify protocol behaviours using namespaces

Alexander Aring

Red Hat

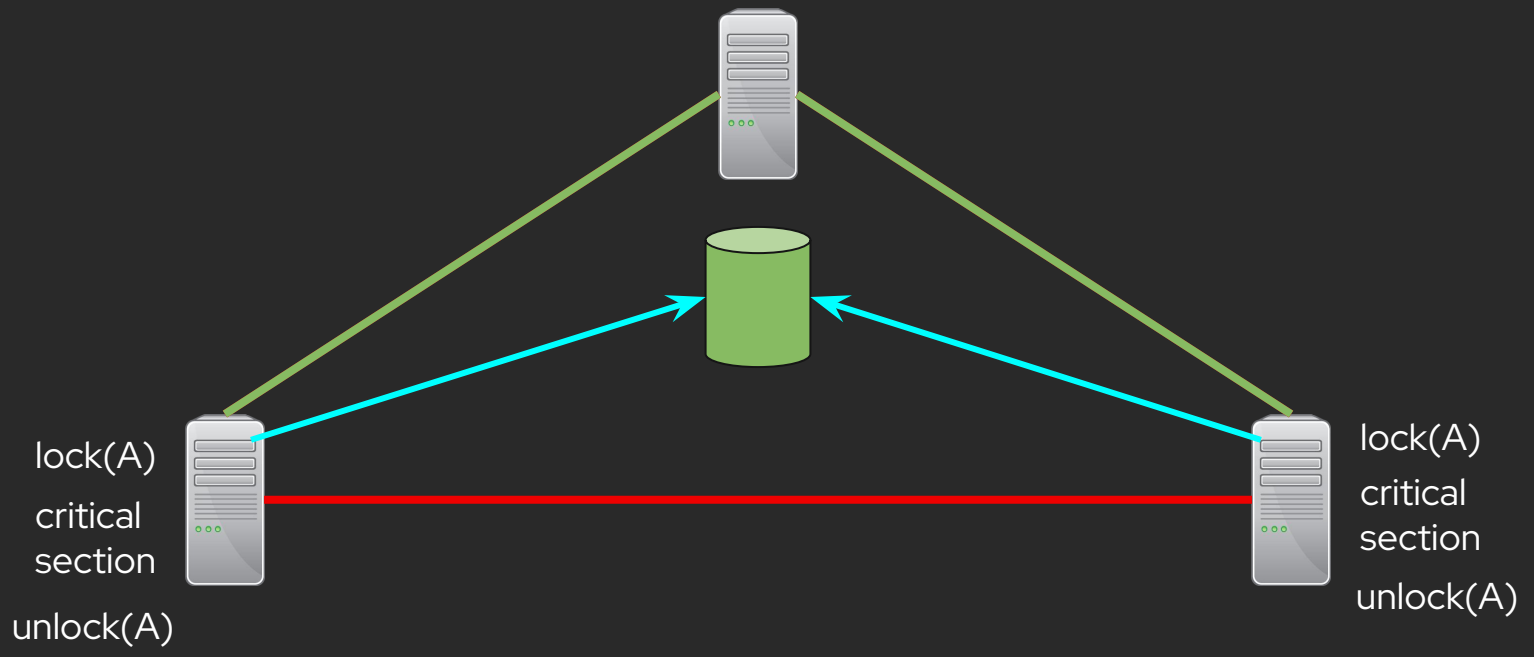
Term “Verification” in this talk

- Are we building the software right?
- Does it meet the expected requirements?
- If violated provide debugging information
- We DON’T do FORMAL Verification

What I do verify?

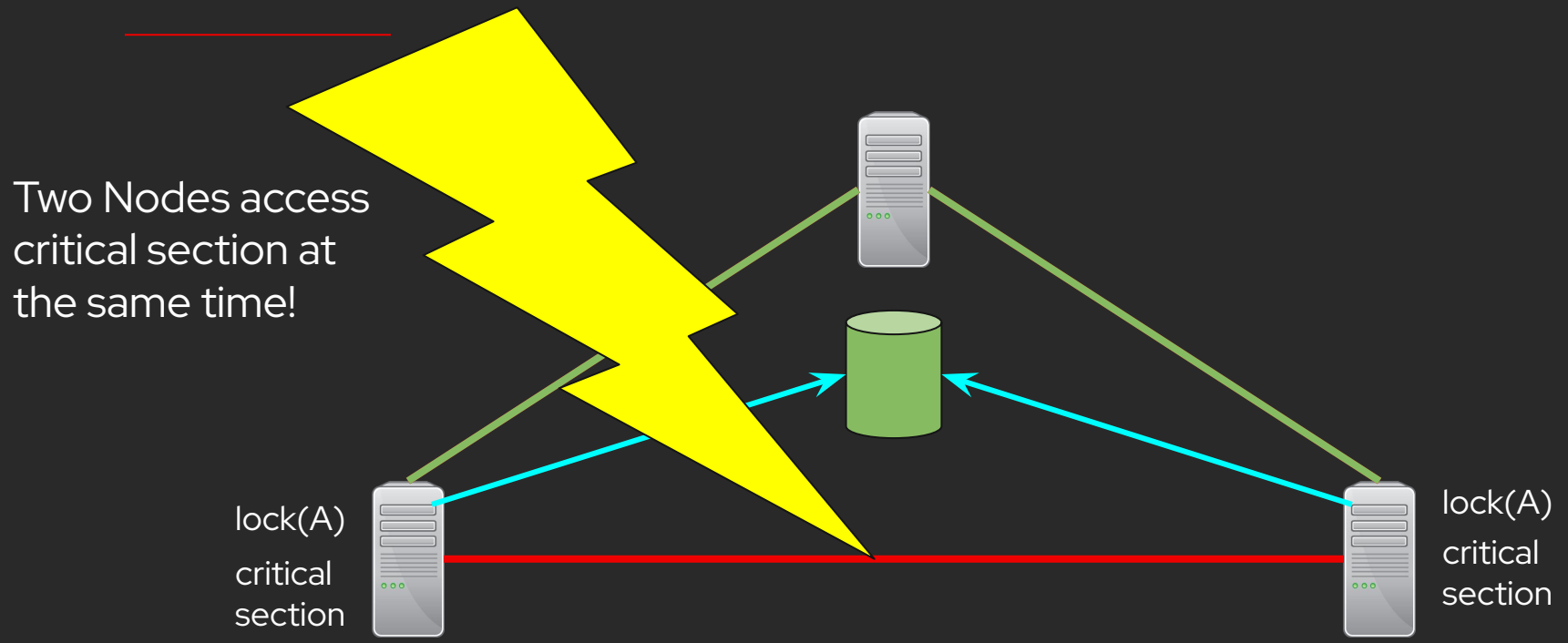
- Example: Distributed Lock Manager (DLM)
- Requirement: Control Mutual Access
- Networking Protocol
- Lock-Context per node (network-entity)
- We verify a DLM as a Blackbox

Distributed Locking Requirements



Red Hat

DLM Violation Example



Red Hat

DLM Requirement

DLM requirement is STRICT

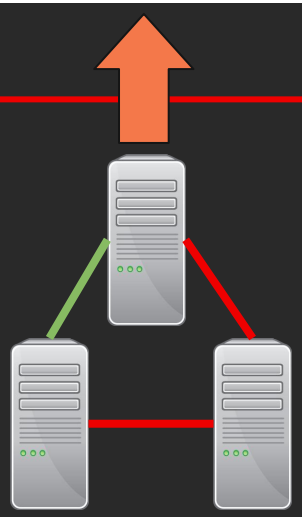
If violated, DLM is broken

DLM Users View

- Users system requirements view

```
1. //init DLM
2. …
3. lock(A);
4. do_critical_section();
5. unlock(A);
6. …
7. //do whatever more
```

- Use standard locking API
- No networking awareness



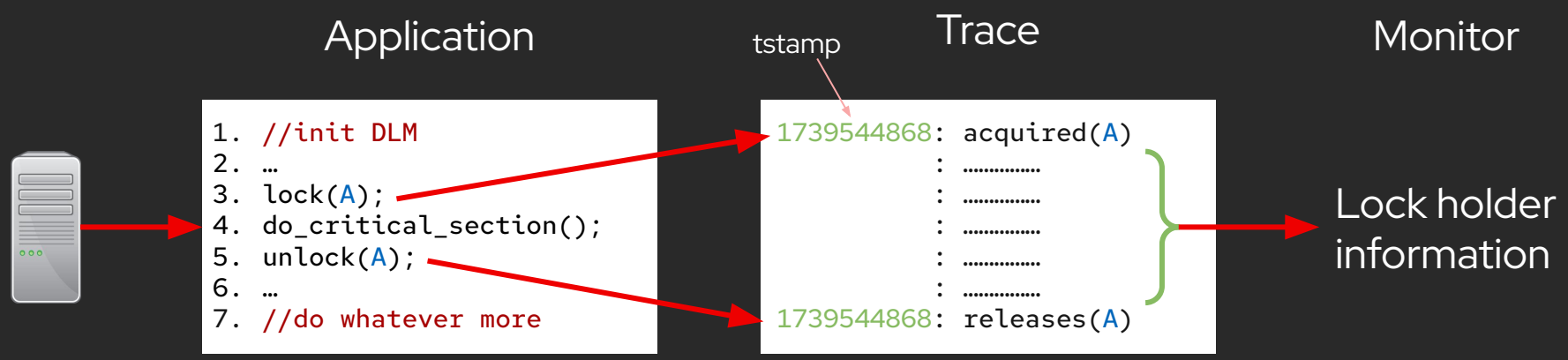
- Users see DLM internals as a Blackbox

- DLM protocol handled internally
- Protocol reflects above User view

DLM Verification by Users View

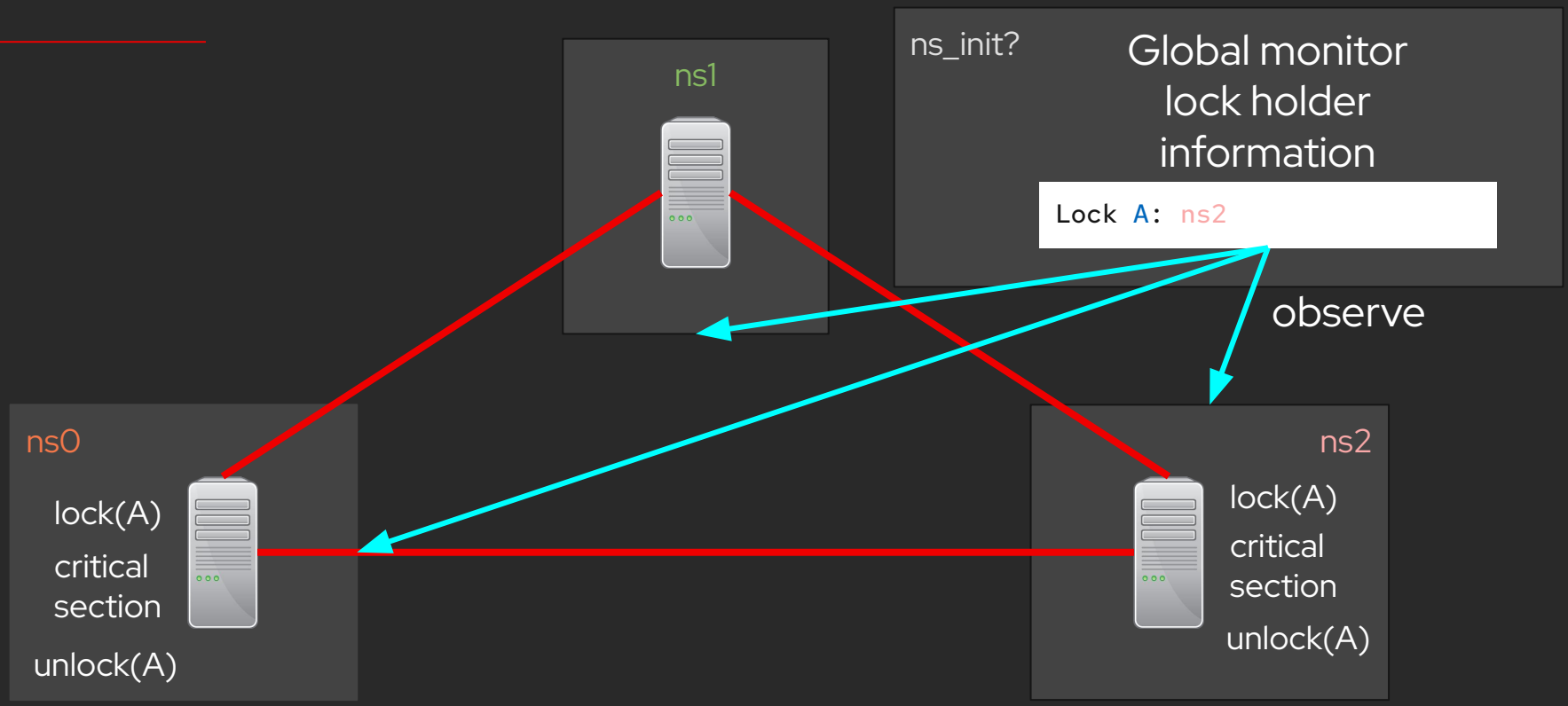
We verify the DLM protocol
from the users view!

Trace DLM user API



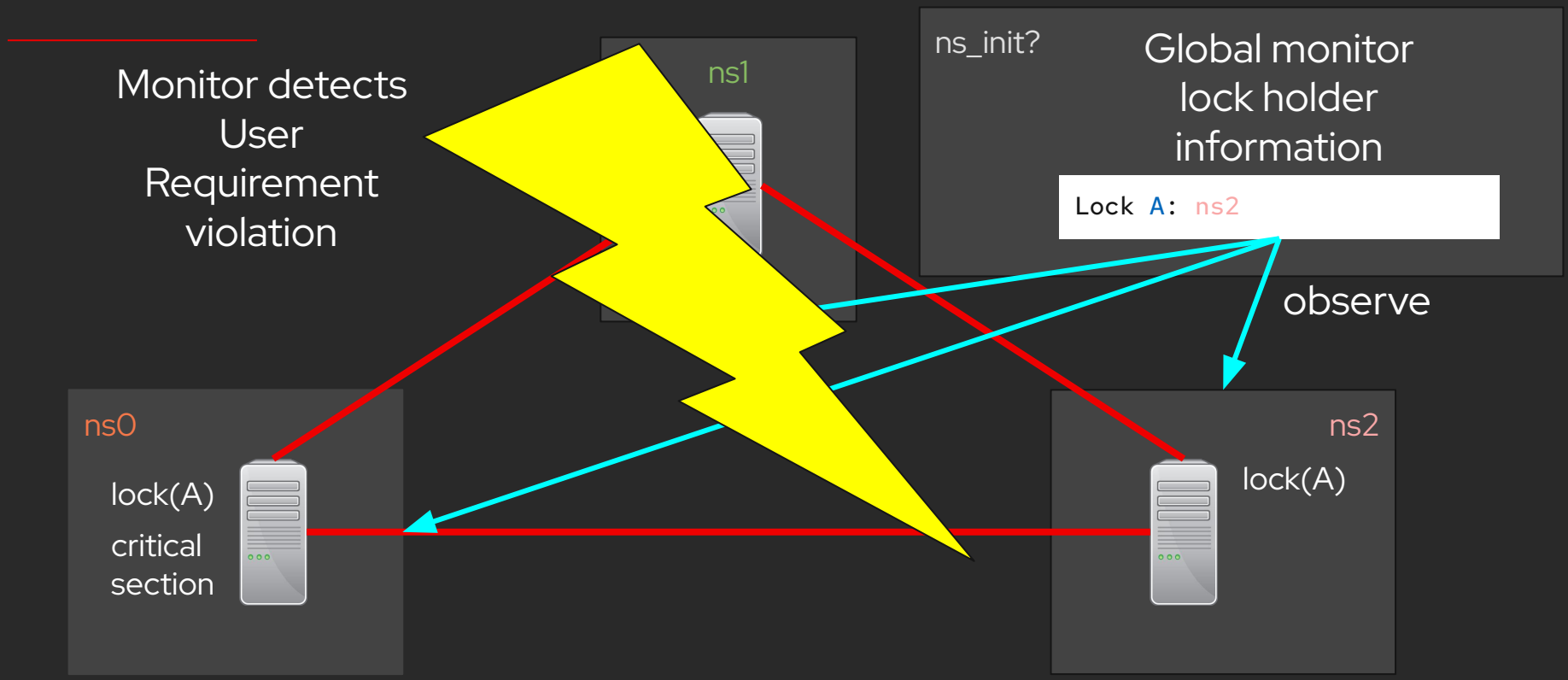
Red Hat

Net-Namespace Environment



Red Hat

Net-Namespace Environment



Red Hat

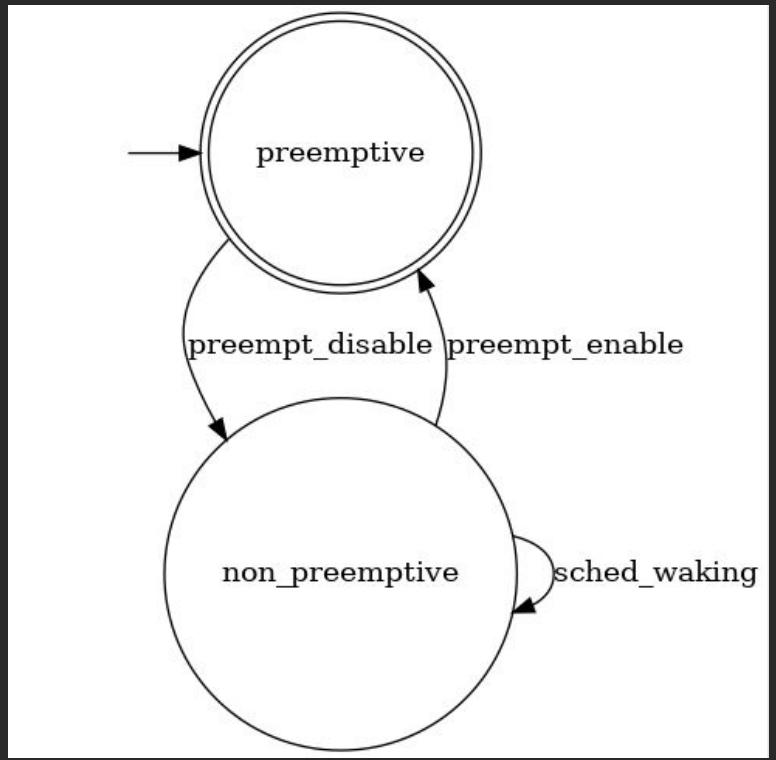
The Monitor-Instance

- Kernel-Verifier “kernel/trace/rv/(monitors)”
- By Daniel Bristot de Oliveira
- Attach/Detach Linux tracepoints
- Ordered API record as they appear -> causality
- Generated C code by compiler
- Verify to be inside model during runtime
- Context-Bases, e.g. per-CPU, per-Task or Global.

Monitor Automaton (Existing WIP)

wakeup in preemptive (WIP) - per CPU - dot file

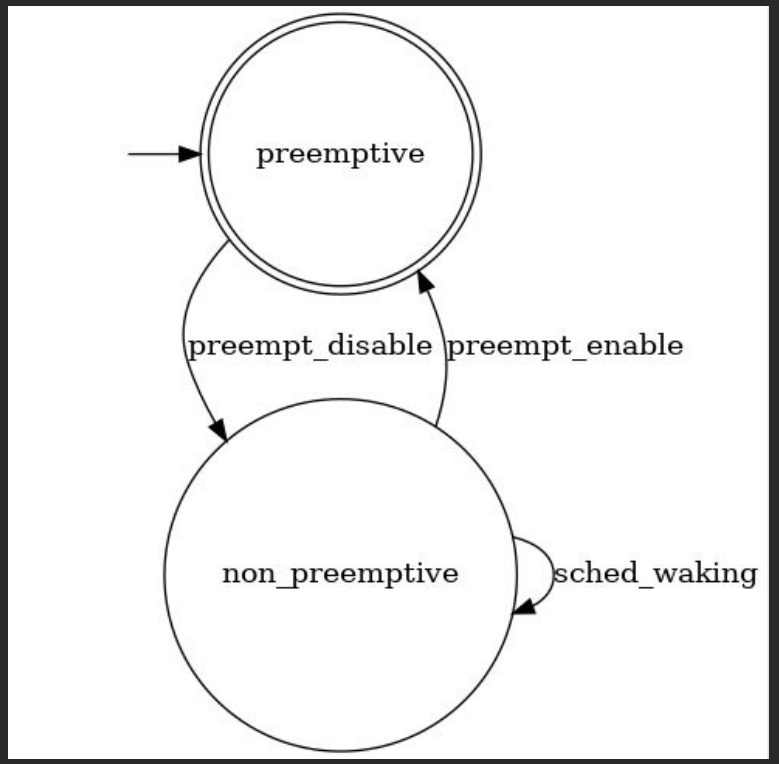
```
1. digraph state_automaton {
2.    {node [shape = circle] "non_preemptive"};
3.    {node [shape = plaintext, style=invis, label=""] "__init_preemptive"};
4.    {node [shape = doublecircle] "preemptive"};
5.    {node [shape = circle] "preemptive"};
6.    "__init_preemptive" -> "preemptive";
7.    "non_preemptive" [label = "non_preemptive"];
8.    "non_preemptive" -> "non_preemptive" [ label = "sched_waking" ];
9.    "non_preemptive" -> "preemptive" [ label = "preempt_enable" ];
10.   "preemptive" [label = "preemptive"];
11.   "preemptive" -> "non_preemptive" [ label = "preempt_disable" ];
12.   { rank = min ;
13.        "__init_preemptive";
14.        "preemptive";
15.   }
16. }
```



Red Hat

DOT Automaton

- “dot2c.py” compiler
- Edges are tracepoints
- Can be enabled with tracefs
- Any workload to verify behaviour
- What happens if violated?

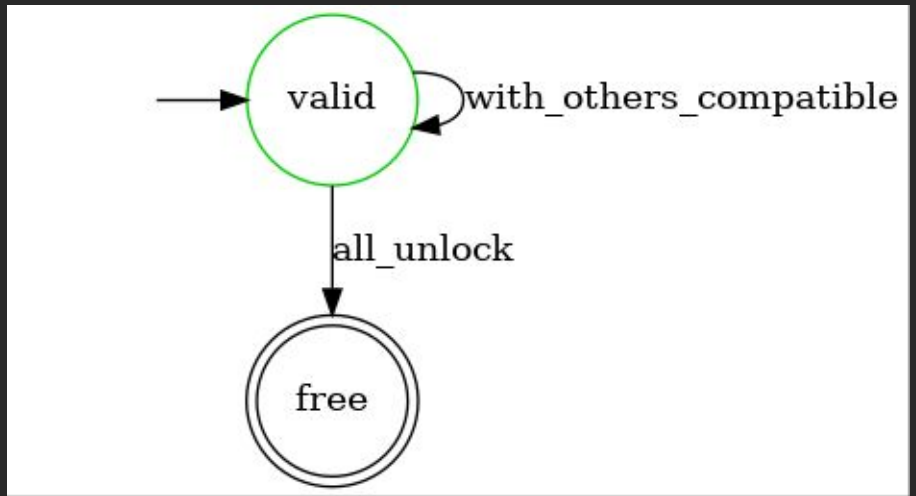


Reactors - What happens on Violation?

- Different reactors can be implemented
  - panic() - kdump, reboot
  - printk() - kernel log
  - Whatever you want?
- Information to debug violated kernel state and how we got there?

Adapting to DLM case

- Monitor is on per Lock context
- Lock context is above netns
- DLM is in reality more complex
- If unlock don’t track holders and free resources
- Violation: non-compatible move into INVALID state



DLM Tracepoint Attachments

```
1. rv_attach_trace_probe("dlm", dlm_acquire, handle_dlm_acquire);
2. rv_attach_trace_probe("dlm", dlm_release, handle_dlm_release);
```

```
1. void handle_dlm_acquired(int *data, struct net *net, u32 id);
2. {
3. …
4.       //lookup or create lock
5.       lk = lookup_or_create_lock(net, id);
6.       if (lk_compatible_with_others(lk)) {
7.            //generated C automaton code -> meet requirements
8.            da_handle_event_dlm(..., with_others_compatible_dlm);
9.       } else {
10.           //violates requirement -> reactor hits!
11.           da_handle_event_dlm(..., event_max_dlm);
12.      }
13. }
14.
15. //remove holder on handle_dlm_release() if no holder do kfree()
16. …
```

```
1. //above netns global hash
2. static struct rhashtable rv_dlm_hash;
3.
4.
5. struct rv_dlm_lock_ctx {
6.       //per monitor ctx
7.       union rv_dlm_lock_monitor rv;
8.       //global lock identifier
9.       u32 lock_id;
10.      //is already acquired?
11.      bool is_acquired;
12       //from which ns holds lock?
13.      const struct *net;
14.      …
15. };
```

RFC Patch Series

https://lore.kernel.org/gfs2/20240827180236.316946-8-aahringo@redhat.com/

*DLM is in the reality more complex as shown, 5 different lock modes
*most interesting part after lock states after recovery

Future Work I

- Net-Namespace based monitor?
  - Currently netns handling in my global monitor
  - Easier to implement netns based monitor
- Netns related Reactors?
  - printk - able to separate netns (or ns general)?
  - Combine with network traffic e.g. tshark analyzer?
- User space Tracing as Kernel-Verifier user?

Future work II

- Without Net-Namespace?
  - Real network environment
  - Time synchronized tracing
- Networkify Tracing 0x18 - Still TODO
  - Using PTP, keeping causality?

Thank you

Red Hat