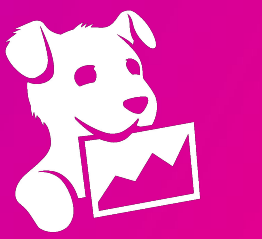
# Performance and reliability pitfalls of eBPF

Usama Saqib @ FOSDEM/ebpf

January 2026



DATADOG

# About Me

- SWE @ Datadog
- Working with eBPF as part of the ebpf-platform team
- Enjoy exploring and working with kernel internals

# Contents

- Kretprobe scaling pains
- A deadlock in fentry
- A production incident due to uprobe

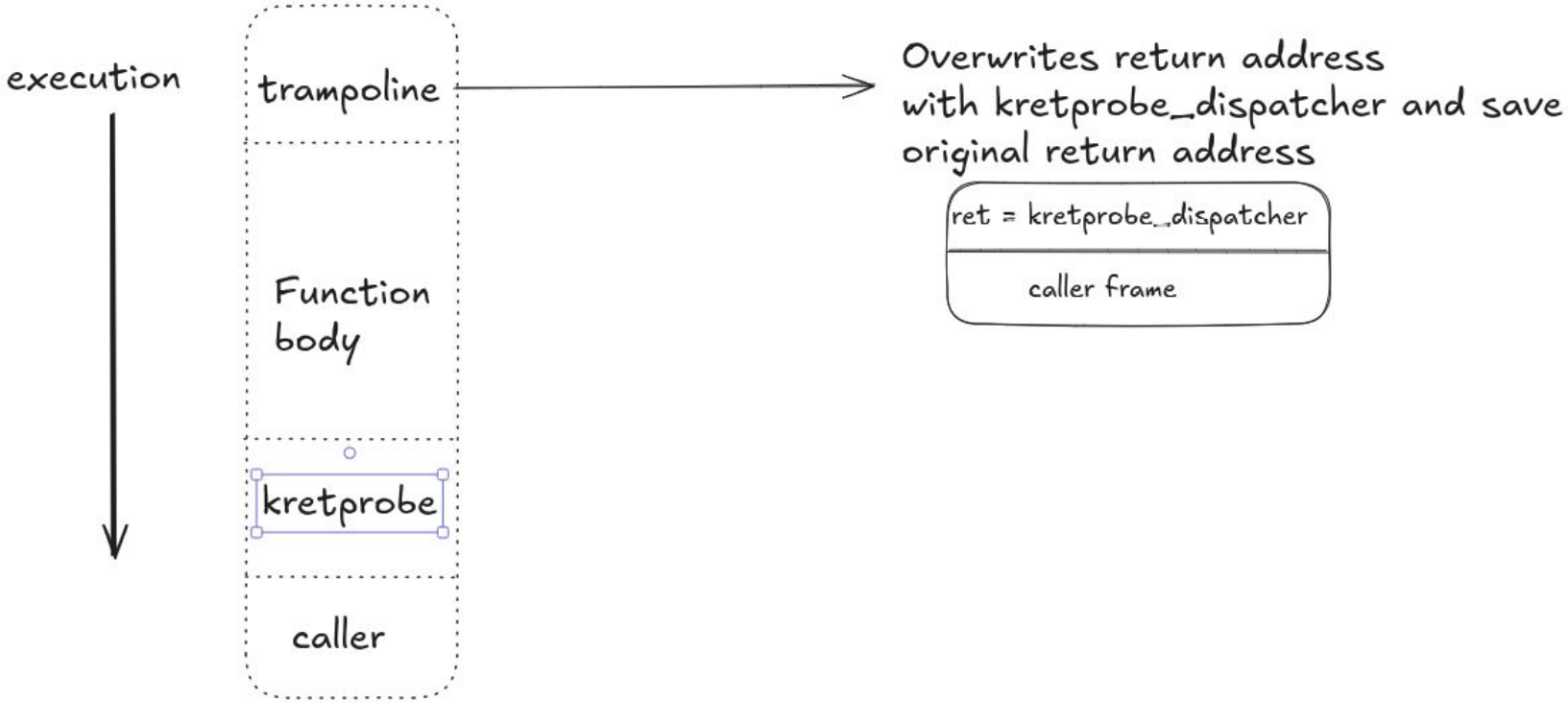
# Kretprobe scaling pains - Spinlocks

- Customer impact
- Running amazon linux 2 with old 4.14 kernel
- Large instance with 96 cores
- Large impact to network throughput


| Distro | Kernel | CNM? | Throughput |
|---|---|---|---|
| AL2 | 4.14 | ✗ | 4.238M |
| AL2 | 4.14 | ✓ | 119.1K |

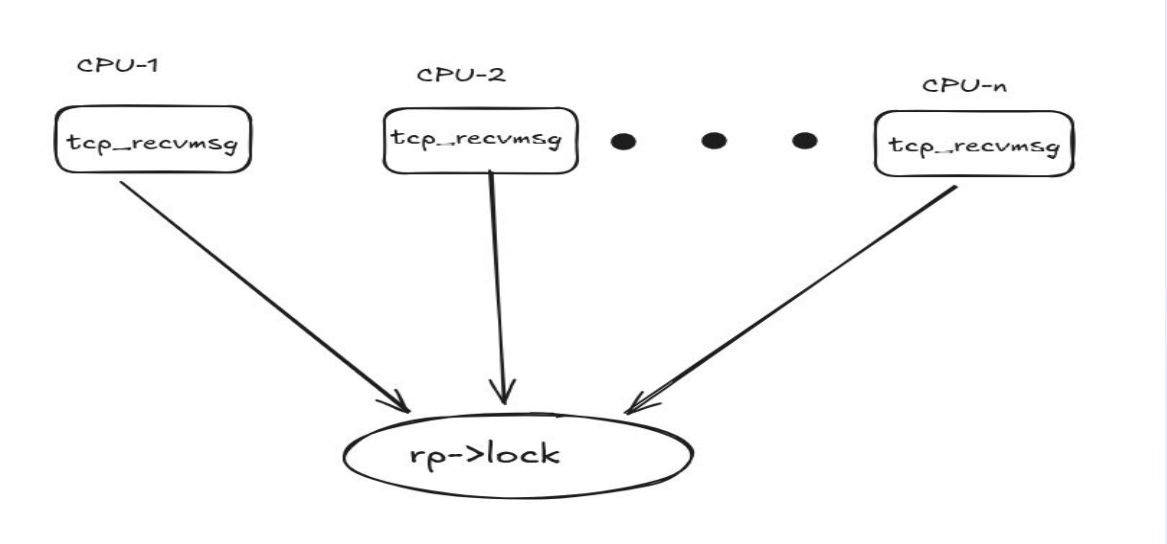
# Kretprobe scaling pains - Internals (kernel 4.14)

- The trampoline saves the return address for each kretprobe invocation in a unique `struct kretprobe_instance`
- The list has max_active slots configured during kprobe registration
- Each invocation requires it's own instance.



# Kretprobe scaling pains - Bottleneck (kernel 4.14)

- The free list of instances is guarded by a spinlock
- Parallel executions of the same kretprobe bottleneck here.




- Perf top reflects this observation.
- CPU time spent in the pre-handler for high contention case
- With massive drop in throughput

```
--38.63%--inet_recvmsg
          tcp_recvmsg
          __key.28673
          ftrace_ops_assist_func
          kprobe_ftrace_handler
          aggr_pre_handler
          |
           --38.63%--pre_handler_kretprobe
                     |
                      --38.60%--_raw_spin_lock_irqsave
                                |
                                 --38.60%--queued_spin_lock_slowpath
                                           native_queued_spin_lock_slowpath
```

# Kretprobe scaling pains - Cache contention (kernel 6.6)

- Customer impact
- Running newer kernel version 6.6
- Very large instance type

| Distro | Kernel | CNM? | Throughput |
|---|---|---|---|
| COS | 6.6 | ✗ | 8.5M |
| COS | 6.6 | ✓ | 85K |

# Kretprobe scaling pains - New mechanism (kernel 6.6)

- Kernel patch removes the spinlock bottleneck in favor of a lockless algorithm.

```
tree f4e6ad94a04db211bfacc7cc6a931a96ecd897fd
parent e563604a5f5a891283b6a8db4001cee833a7c6b8
author Peter Zijlstra <peterz@infradead.org> Sat Aug 29 22:03:56 2020 +0900
committer Ingo Molnar <mingo@kernel.org> Mon Oct 12 18:27:28 2020 +0200

kprobes: Replace rp->free_instance with freelist

Gets rid of rp->lock, and as a result kretprobes are now fully
lockless.

Signed-off-by: Peter Zijlstra (Intel) <peterz@infradead.org>
Signed-off-by: Masami Hiramatsu <mhiramat@kernel.org>
Signed-off-by: Ingo Molnar <mingo@kernel.org>
Link: https://lore.kernel.org/r/159870623583.1229682.17472357584134058687.stgit@devnote2
```

- Hint of the issue can be found in the kernel comments.

```
/*
 * Copyright: cameron@moodycamel.com
 *
 * A simple CAS-based lock-free free list. Not the fastest thing in the world
 * under heavy contention, but simple and correct (assuming nodes are never
 * freed until after the free list is destroyed), and fairly speedy under low
 * contention.
 *
 * Adapted from: https://moodycamel.com/blog/2014/solving-the-aba-problem-for-lock-free-free-lists
 */
```

# Kretprobe scaling pains - Bottleneck (kernel 6.6)

- New lock less design
  - No lock so no lock contention!

- On a big enough machine memory becomes the bottleneck.
  - The freelist manipulation causes cache bouncing
  - The problem gets worse with more CPUs

- Shown by perf stat where we see we are bottlenecked on memory

# Kretprobe scaling pains - Objpool (kernel 6.6+)

```
tree db31d9a59c0ac9b21a7aa243bb8a2d02f0525c4f
parent ef8a257b4e499a979364b1f9caf25a325f6ee8b8
author Masami Hiramatsu <mhiramat@kernel.org> Sat Mar 26 11:27:05 2022 +0900
committer Alexei Starovoitov <ast@kernel.org> Mon Mar 28 19:38:09 2022 -0700

kprobes: Use rethook for kretprobe if possible

Use rethook for kretprobe function return hooking if the arch sets
CONFIG_HAVE_RETHOOK=y. In this case, CONFIG_KRETPROBE_ON_RETHOOK is
set to 'y' automatically, and the kretprobe internal data fields
switches to use rethook. If not, it continues to use kretprobe
specific function return hooks.

Suggested-by: Peter Zijlstra <peterz@infradead.org>
Signed-off-by: Masami Hiramatsu <mhiramat@kernel.org>
Signed-off-by: Alexei Starovoitov <ast@kernel.org>
Link: https://lore.kernel.org/bpf/164826162556.2455864.12255833167233452047.stgit@devnote2
```

```
/*
 * objpool: ring-array based lockless MPMC queue
 *
 * Copyright: wuqiang.matt@bytedance.com,mhiramat@kernel.org
 *
 * objpool is a scalable implementation of high performance queue for
 * object allocation and reclamation, such as kretprobe instances.
 *
 * With leveraging percpu ring-array to mitigate hot spots of memory
 * contention, it delivers near-linear scalability for high parallel
 * scenarios. The objpool is best suited for the following cases:
 * 1) Memory allocation or reclamation are prohibited or too expensive
 * 2) Consumers are of different priorities, such as irqs and threads
 *
```

# An Fentry and RCU bug

- Lets move away from kretprobes and into the modern world of fentry

- Reports of failures when exiting the agent container

```
root@ddvm:~# docker container stop dd-agent-foo
Error response from daemon: cannot stop container: dd-agent-foo: tried to kill container, but did not receive an exit event
```

# An Fentry and RCU bug

**echo w > /proc/sysrq-trigger**

Two tasks in uninterruptible sleep

1. System-probe (agent)
2. rcu_tasks_kthread

```
[  183.865554] task:system-probe    state:D stack:0     p
[  183.865559] Call Trace:
[  183.865562]  <TASK>
[  183.865567]  __schedule+0x2cb/0x760
[  183.865578]  schedule+0x63/0x110
[  183.865583]  schedule_timeout+0x157/0x170
[  183.865593]  wait_for_completion+0x88/0x150
[  183.865603]  __wait_rcu_gp+0x17e/0x190
[  183.865611]  synchronize_rcu_tasks_generic+0x64/0xe0
[  183.865617]  ? __pfx_call_rcu_tasks+0x10/0x10
[  183.865623]  ? __pfx_wakeme_after_rcu+0x10/0x10
[  183.865632]  synchronize_rcu_tasks+0x15/0x20
[  183.865638]  ftrace_shutdown.part.0+0xd1/0x1f0
[  183.865651]  ? 0xffffffffc11f0000
[  183.865686]  unregister_ftrace_function+0x47/0x170
[  183.865694]  ? 0xffffffffc11f0000
[  183.865702]  unregister_ftrace_direct+0x60/0x1e0
[  183.865710]  ? bpf_trampoline_6442514787+0xba/0x1000
[  183.865723]  ? 0xffffffffc11f0000
[  183.865729]  ? __pfx___ia32_sys_setregid16+0x10/0x10
[  183.865736]  bpf_trampoline_update+0x505/0x5f0
[  183.865745]  ? __radix_tree_delete+0x9e/0x150
[  183.865757]  bpf_trampoline_unlink_prog+0x9d/0x130
[  183.865765]  bpf_tracing_link_release+0x16/0x50
[  183.865772]  bpf_link_free+0x52/0x80
[  183.865778]  bpf_link_release+0x26/0x40
[  183.865782]  __fput+0xf9/0x2c0
[  183.865790]  ____fput+0xe/0x20
[  183.865794]  task_work_run+0x5e/0xa0
[  183.865799]  exit_to_user_mode_loop+0x105/0x130
[  183.865805]  exit_to_user_mode_prepare+0xa5/0xb0
[  183.865808]  syscall_exit_to_user_mode+0x29/0x60
[  183.865814]  do_syscall_64+0x62/0x90
[  183.865821]  ? __rseq_handle_notify_resume+0x37/0x70
[  183.865828]  ? exit_to_user_mode_loop+0xe5/0x130
[  183.865832]  ? exit_to_user_mode_prepare+0x30/0xb0
[  183.865836]  ? syscall_exit_to_user_mode+0x37/0x60
[  183.865840]  ? do_syscall_64+0x62/0x90
[  183.865843]  ? do_syscall_64+0x62/0x90
[  183.865846]  ? do_syscall_64+0x62/0x90
[  183.865850]  entry_SYSCALL_64_after_hwframe+0x73/0xdd
[  183.865856] RIP: 0033:0x1ce792e
```

```
[  183.865273] task:rcu_tasks_kthre state:D stack:0
[  183.865284] Call Trace:
[  183.865288]  <TASK>
[  183.865322]  __schedule+0x2cb/0x760
[  183.865340]  schedule+0x63/0x110
[  183.865345]  schedule_timeout+0x157/0x170
[  183.865357]  wait_for_completion+0x88/0x150
[  183.865367]  __synchronize_srcu+0x91/0xd0
[  183.865374]  ? __pfx_wakeme_after_rcu+0x10/0x10
[  183.865383]  ? ktime_get_mono_fast_ns+0x3c/0xa0
[  183.865392]  synchronize_srcu+0xb9/0x110
[  183.865399]  rcu_tasks_postscan+0x39/0x60
[  183.865405]  rcu_tasks_wait_gp+0x116/0x310
[  183.865411]  ? _raw_spin_unlock_irqrestore+0x11/0x60
[  183.865418]  ? rcu_tasks_need_gpcb+0x199/0x2f0
[  183.865427]  ? __pfx_rcu_tasks_kthread+0x10/0x10
[  183.865433]  rcu_tasks_one_gp+0x108/0x140
[  183.865440]  rcu_tasks_kthread+0x33/0x40
[  183.865447]  kthread+0xef/0x120
[  183.865456]  ? __pfx_kthread+0x10/0x10
[  183.865464]  ret_from_fork+0x44/0x70
[  183.865472]  ? __pfx_kthread+0x10/0x10
[  183.865479]  ret_from_fork_asm+0x1b/0x30
[  183.865495]  </TASK>
```

# An Fentry and RCU bug - echo t > /proc/sysrq-trigger

s6-svscan in interruptible sleep.

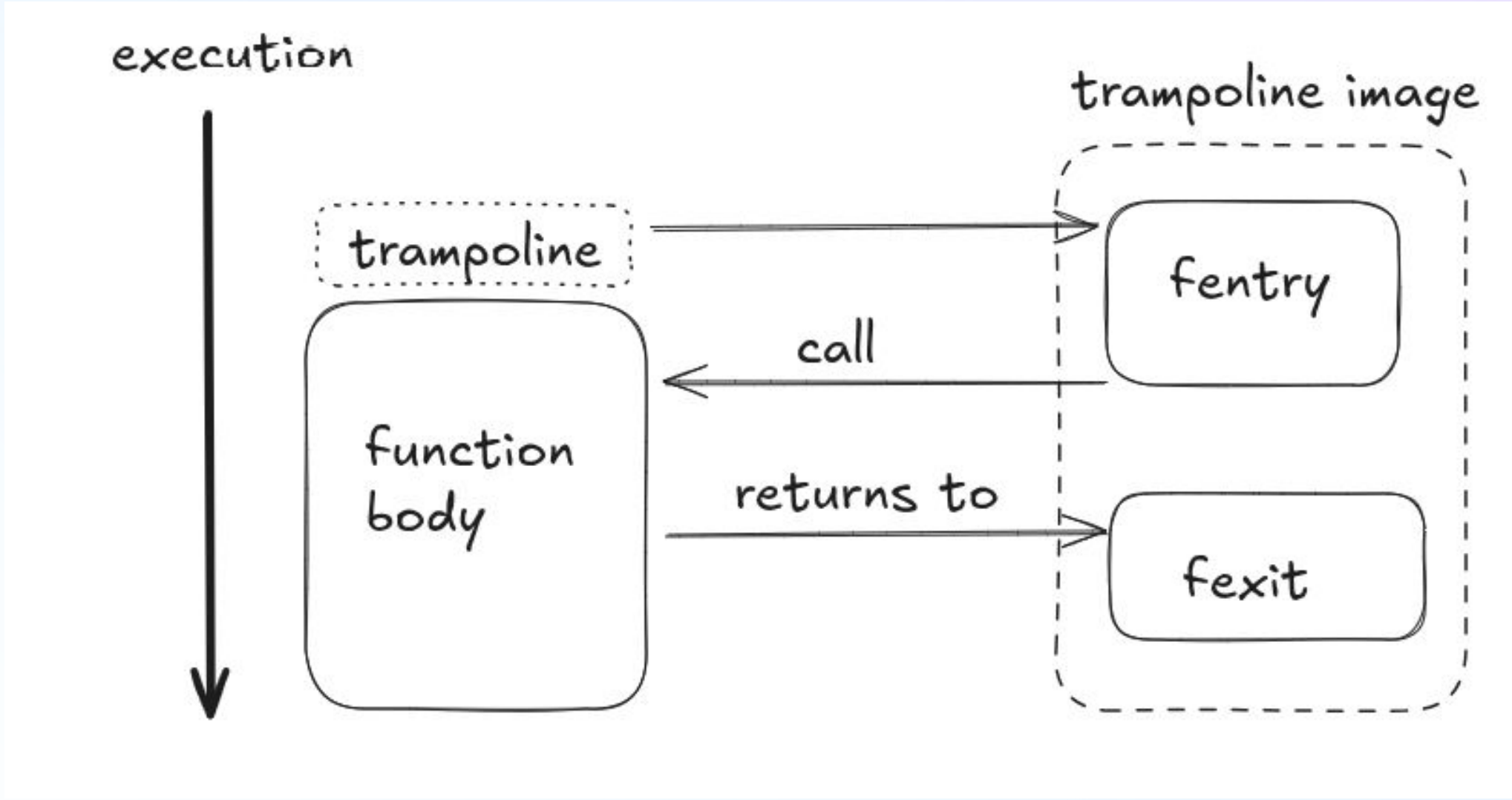
S6 is used by docker for process management.

Parent of system-probe

Pid 1 in the pid namespace of the container.

```
[  322.722346] task:s6-svscan       state:S stack:0     pi
[  322.722347] Call Trace:
[  322.722348]  <TASK>
[  322.722349]  __schedule+0x2cb/0x760
[  322.722352]  schedule+0x63/0x110
[  322.722353]  do_wait+0x173/0x320
[  322.722356]  kernel_wait4+0xbd/0x170
[  322.722359]  ? __pfx_child_wait_callback+0x10/0x10
[  322.722362]  zap_pid_ns_processes+0x115/0x1b0
[  322.722365]  forget_original_parent+0x356/0x370
[  322.722367]  ? cgroup_exit+0xf3/0x1b0
[  322.722370]  do_exit+0x5a9/0x6f0
[  322.722371]  ? do_wait+0x1b1/0x320
[  322.722374]  do_group_exit+0x35/0x90
[  322.722376]  get_signal+0x8d8/0x940
[  322.722377]  ? __pfx_child_wait_callback+0x10/0x10
[  322.722381]  arch_do_signal_or_restart+0x39/0x120
[  322.722385]  exit_to_user_mode_loop+0x9a/0x130
[  322.722386]  exit_to_user_mode_prepare+0xa5/0xb0
[  322.722388]  syscall_exit_to_user_mode+0x29/0x60
[  322.722390]  do_syscall_64+0x62/0x90
[  322.722391]  ? exit_to_user_mode_prepare+0x9b/0xb0
[  322.722392]  ? syscall_exit_to_user_mode+0x37/0x60
[  322.722394]  ? do_syscall_64+0x62/0x90
[  322.722398]  ? switch_fpu_return+0x55/0xf0
[  322.722401]  ? exit_to_user_mode_prepare+0x9b/0xb0
[  322.722403]  ? syscall_exit_to_user_mode+0x37/0x60
[  322.722404]  ? do_syscall_64+0x62/0x90
[  322.722406]  entry_SYSCALL_64_after_hwframe+0x73/0xdd
```

# An Fentry and RCU bug - Fentry Internals



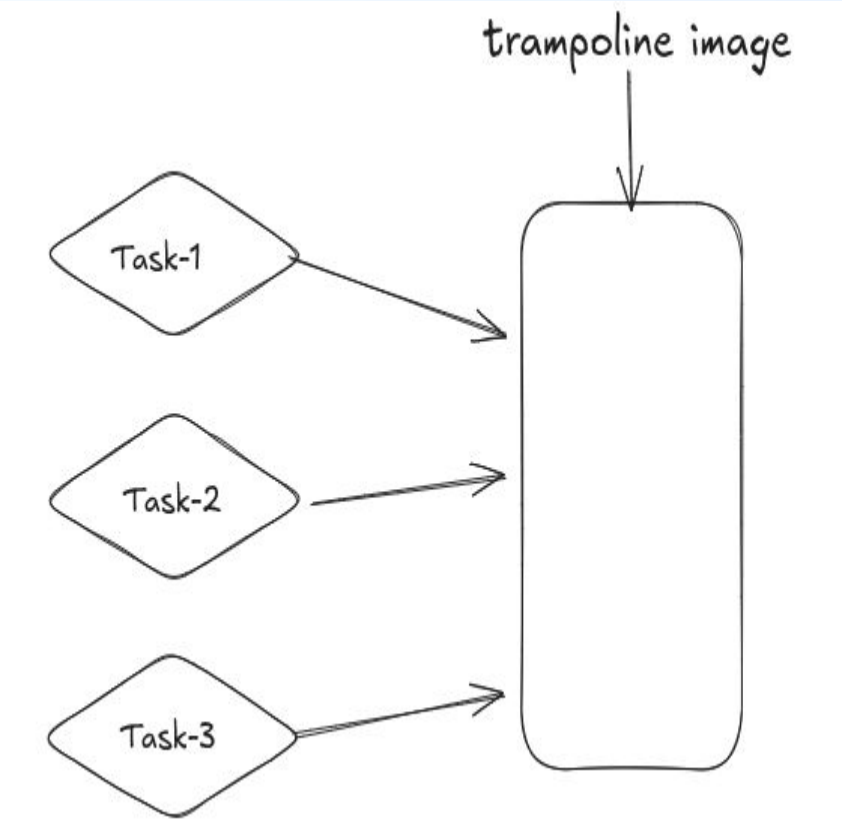
# An Fentry and RCU bug - RCU-tasks

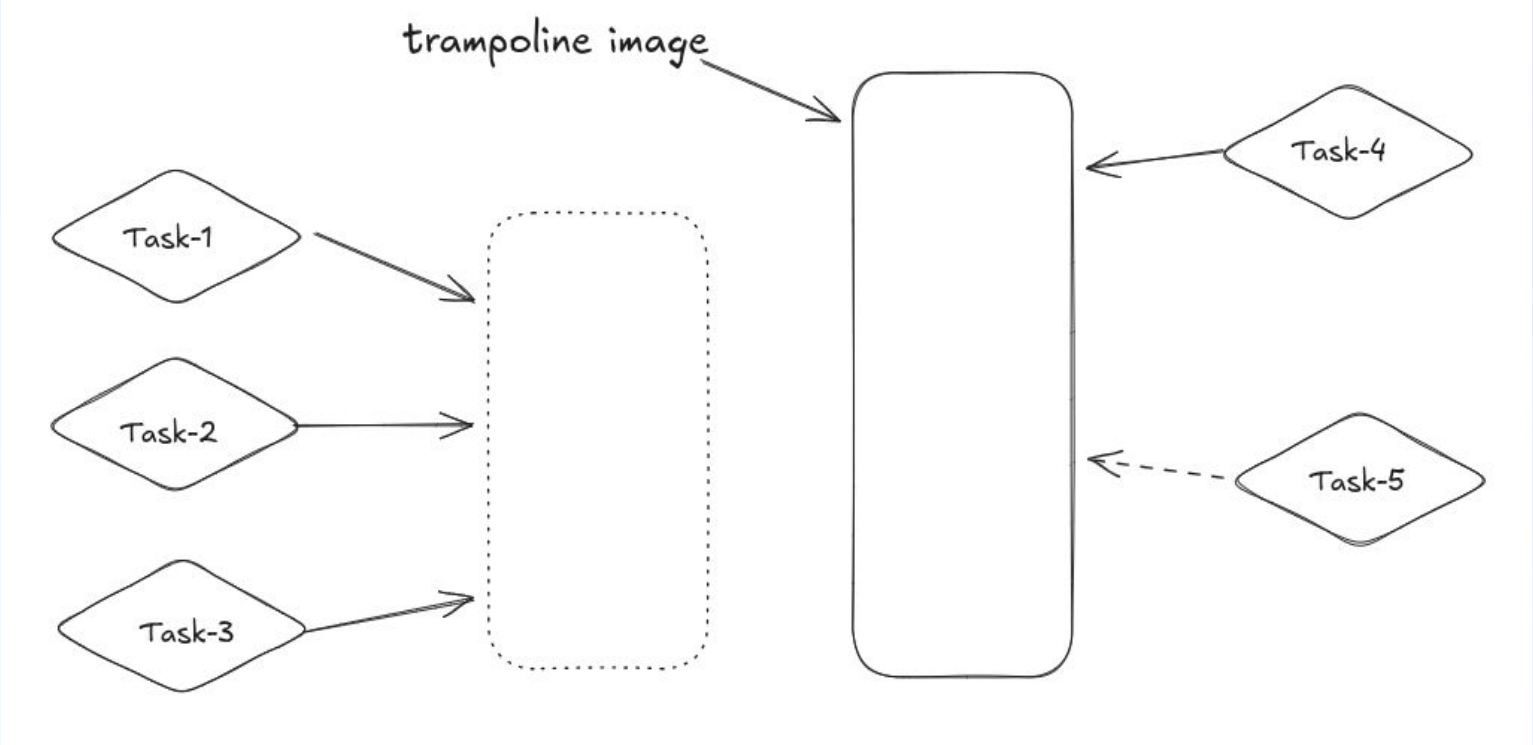
The RCU tasks subsystem is an RCU flavor designed to figure out when no **process** is holding a reference to a structure.

- Normal RCU allows to figure out when no CPU is holding a reference to a structure.
  - Only one process on a CPU can hold a reference
  - References can only be held in an “atomic context”
  - Meaning of “atomic context” in dependent on preemption modes
    - In all cases except PREEMPT_RT explicit blocking is always illegal
    - With BPF we are dealing with cases where blocking+preemption is not allowed.
    - Other preemption modes is a whole other can of worms as far as BPF is concerned.

- In contrast RCU-tasks
  - Multiple processes on a CPU can hold a reference
  - The quiescent state here is that all tasks have performed a voluntary context switch

# An Fentry and RCU bug - RCU-tasks



# An Fentry and RCU bug - RCU-tasks

1. Scan the tasklist for all tasks on the runqueue, and add them to the holdout list. The holdout list is all tasks which have not yet performed a voluntary context switch.

2. Wait for all tasks which are in late stage do_exit to complete. Tasks in late stage of do_exit are not on the tasklist, so therefore, are not visible to the scan done above. The wait is performed by calling synchronize_srcu(&tasks_rcu_exit_srcu). This blocks the current task until all tasks_rcu_exit_srcu read-side critical sections have completed.

3. Loop over the holdout list until all tasks in the list have been seen to perform a voluntary context switch. This is tracked by placing a hook in the scheduler.

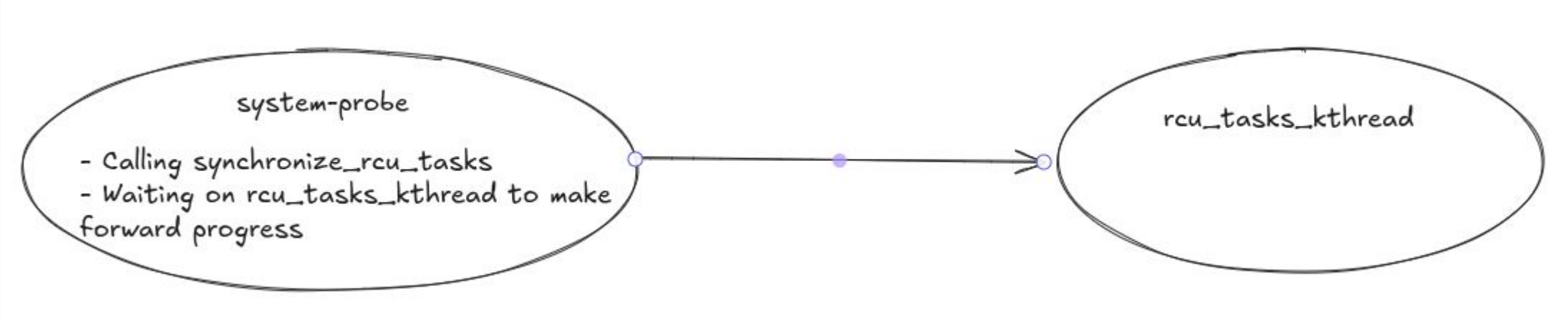
# An Fentry and RCU bug - RCU-tasks

```
tree 5b6d2c42aaf8b20397bd09c0ac31738618f57046
parent 53c6d4edf874d3cbc031a53738c6cba9277faea5
author Paul E. McKenney <paulmck@linux.vnet.ibm.com> Mon Aug 4 06:10:23 2014 -0700
committer Paul E. McKenney <paulmck@linux.vnet.ibm.com> Sun Sep 7 16:27:22 2014 -0700

rcu: Make TASKS_RCU handle tasks that are almost done exiting

Once a task has passed exit_notify() in the do_exit() code path, it
is no longer on the task lists, and is therefore no longer visible
to rcu_tasks_kthread().  This means that an almost-exited task might
be preempted while within a trampoline, and this task won't be waited
on by rcu_tasks_kthread().  This commit fixes this bug by adding an
srcu_struct.  An exiting task does srcu_read_lock() just before calling
exit_notify(), and does the corresponding srcu_read_unlock() after
doing the final preempt_disable().  This means that rcu_tasks_kthread()
can do synchronize_srcu() to wait for all mostly-exited tasks to reach
their final preempt_disable() region, and then use synchronize_sched()
to wait for those tasks to finish exiting.
```

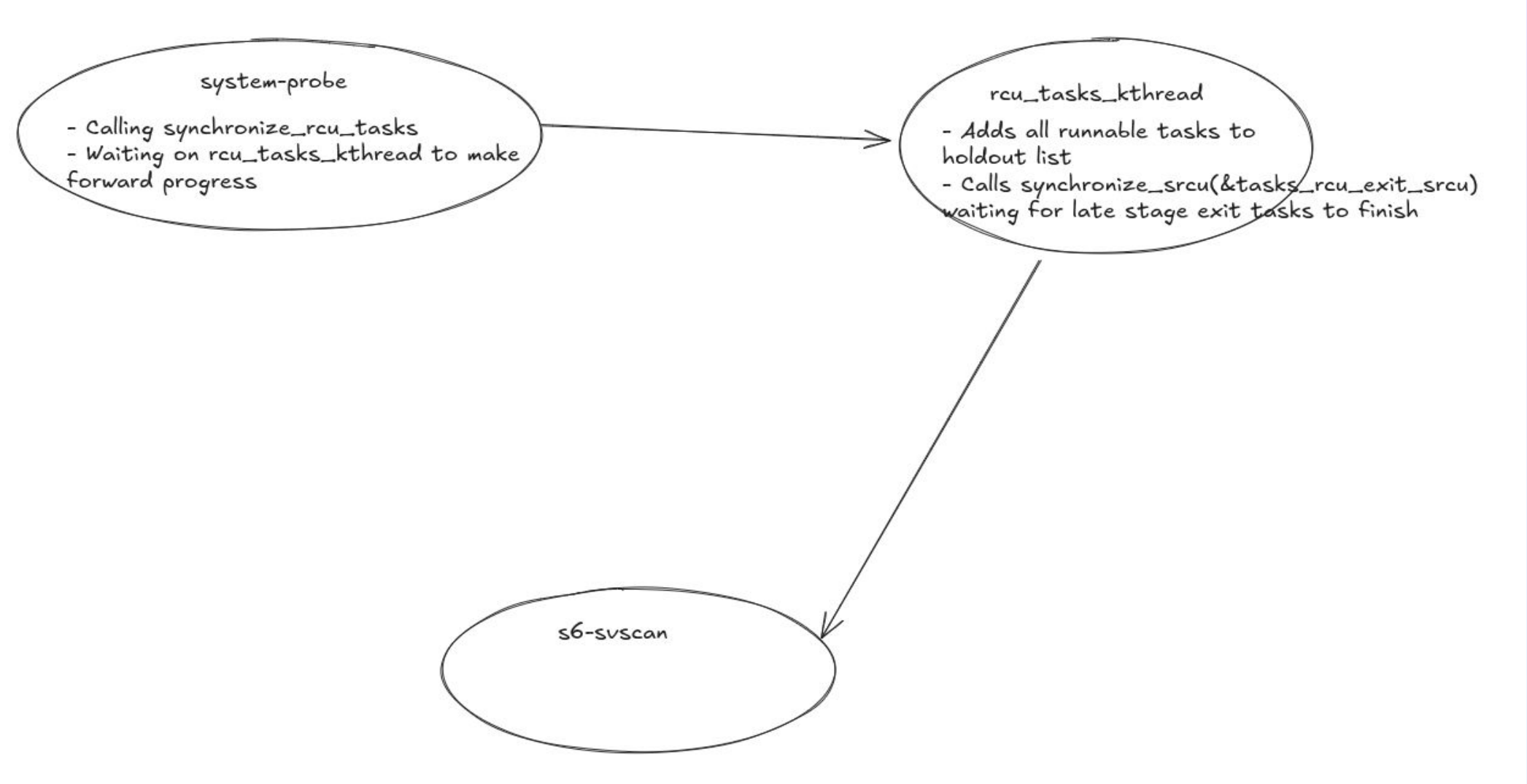
# An Fentry and RCU bug - Zapper

- Sends kill signal to all children
- Waits until they are dead
- Does this inside a tasks_rcu_exit_srcu read-side critical section **!!!!**

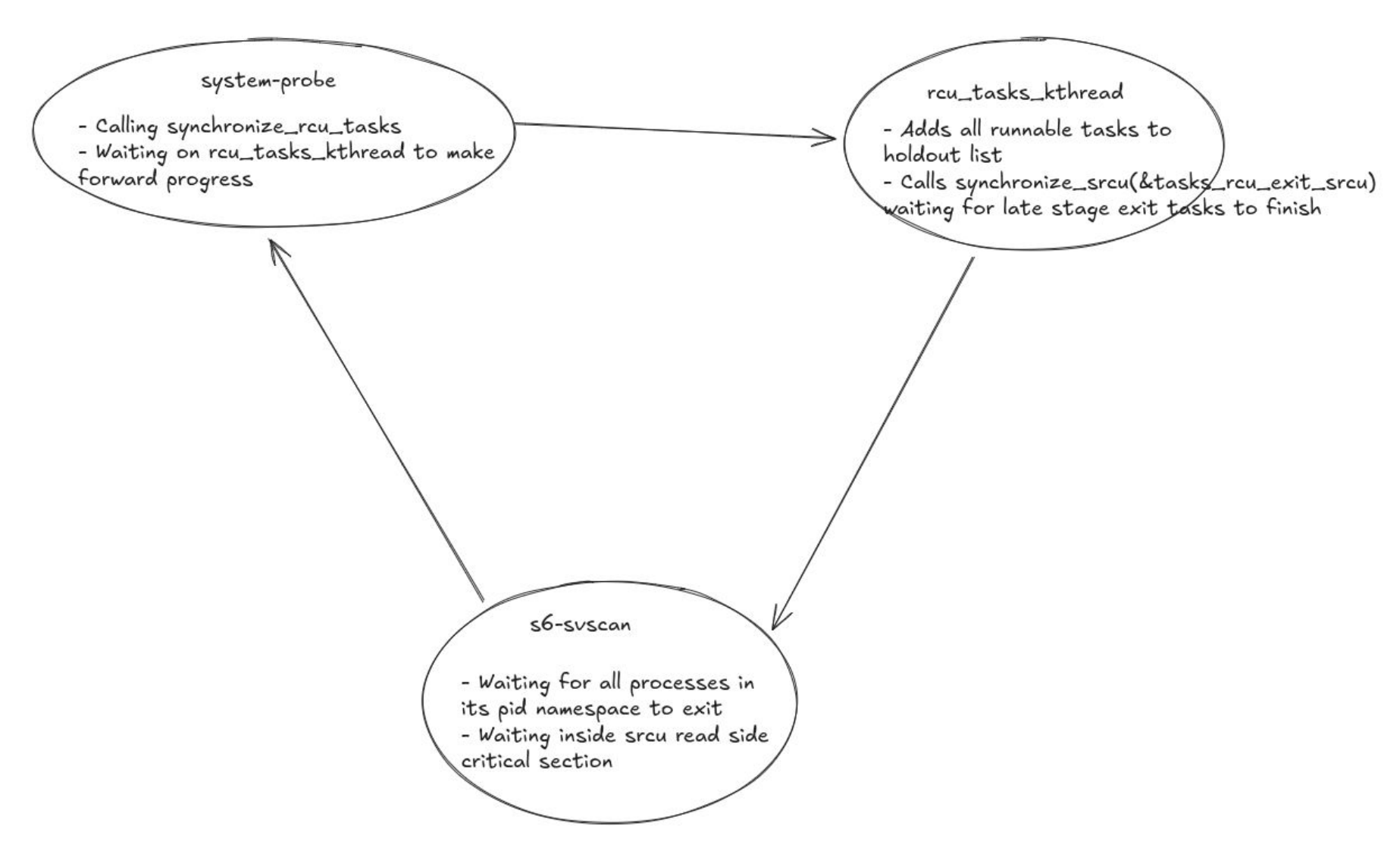
# An Fentry and RCU bug - Full picture



# An Fentry and RCU bug - Full picture



# An Fentry and RCU bug - Full picture



# An Fentry and RCU bug - Conclusion

- At that time we were blocked from using fentry since the fix for this was not backported

```
tree 3e97b5789d9c52bf95808d07be34c86d2eb27020
parent 46faf9d8e1d52e4a91c382c6c72da6bd8e68297b
author Paul E. McKenney <paulmck@kernel.org> Fri Feb 2 11:28:45 2024 -0800
committer Boqun Feng <boqun.feng@gmail.com> Sun Feb 25 14:21:43 2024 -0800

rcu-tasks: Maintain lists to eliminate RCU-tasks/do_exit() deadlocks

This commit continues the elimination of deadlocks involving do_exit()
and RCU tasks by causing exit_tasks_rcu_start() to add the current
task to a per-CPU list and causing exit_tasks_rcu_stop() to remove the
current task from whatever list it is on.  These lists will be used to
track tasks that are exiting, while still accounting for any RCU-tasks
quiescent states that these tasks pass though.

[ paulmck: Apply Frederic Weisbecker feedback. ]

Link: https://lore.kernel.org/all/20240118021842.290665-1-chenzhongjin@huawei.com/
```

# Production impact from uprobe

- Service unavailability on a few nodes in production

- Khungtaskd reporting a large number of tasks hung for more than 120 seconds!
  - Some cases only tasks associated with the production application
  - Some cases tasks from bash, systemd, runc, and many more

- Among these were tasks
  - Registering uprobes
  - Firing uprobes

| Uprobe Registration | Uprobe Firing |
| --- | --- |
| uprobe_register -> register_for_each_vma | handle_swbp -> find_active_uprobe |

# Production impact from uprobe - Investigation

Is it a deadlock again?

- We found an *almost* circular lock dependency but not complete.
- So cannot conclude deadlock.

Memory corruption?

- We found a patch missing in our kernel which discussed a possible memory corruption effect struct mm_struct.
- We built a custom kernel to deterministically trigger the condition for that corruption.
- Did not result in the symptoms we were seeing.

Livelock due to memory pressure?

- Hypothesis that the kernel was looping trying to allocate some memory but failing due to memory pressure.
- Evaluated all page allocation paths in the uprobe register and uprobe firing path and this did not seem possible.

# Production impact from uprobe - Investigation

Monitoring kernel locks

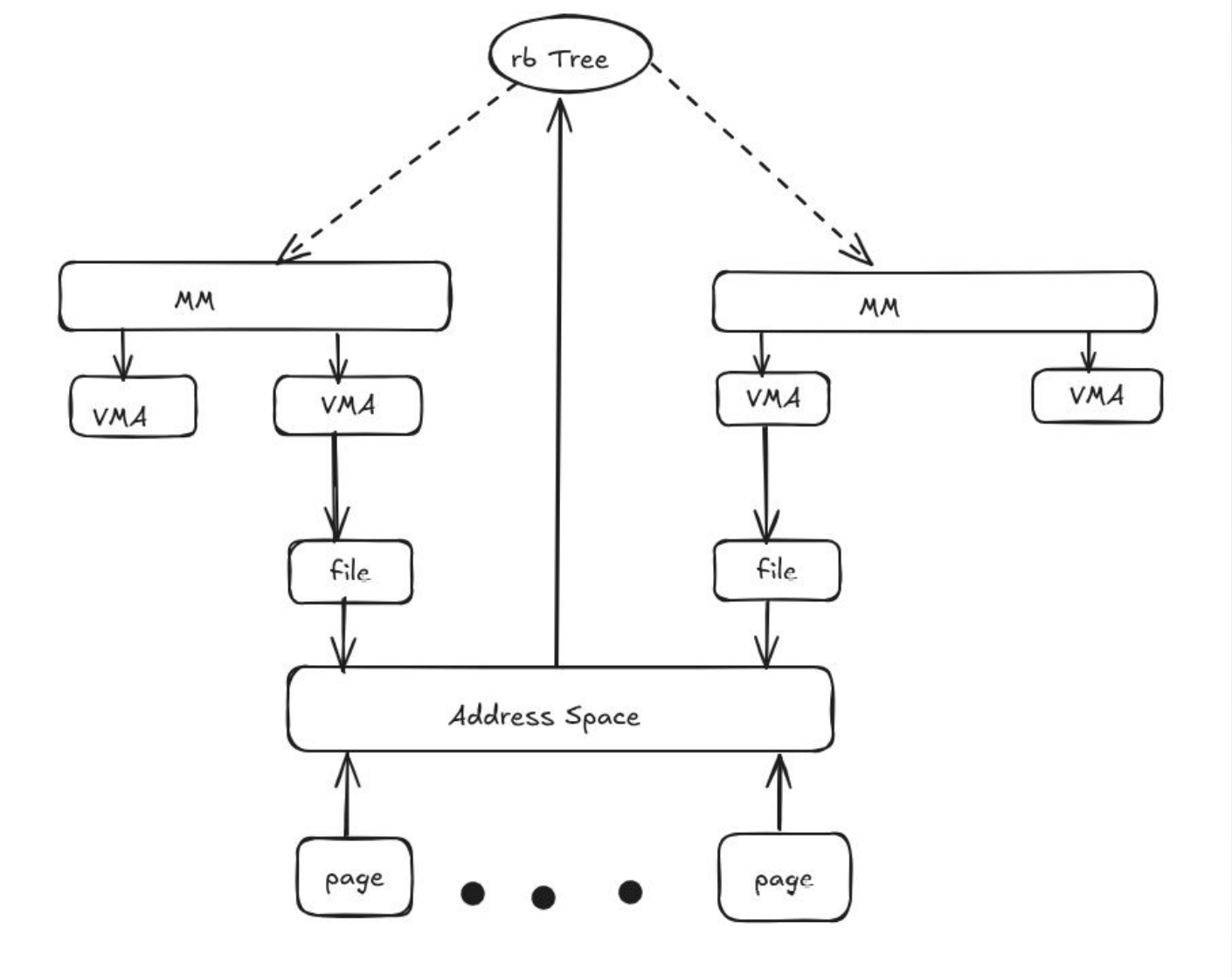
- Began monitoring kernel locks to establish deadlock and failed.

```
trace_contention_begin
trace_contention_end
```

# Production impact from uprobe - Uprobe Registration

Input to uprobe registration

1. Inode of the file to be probed
2. Address to be probed

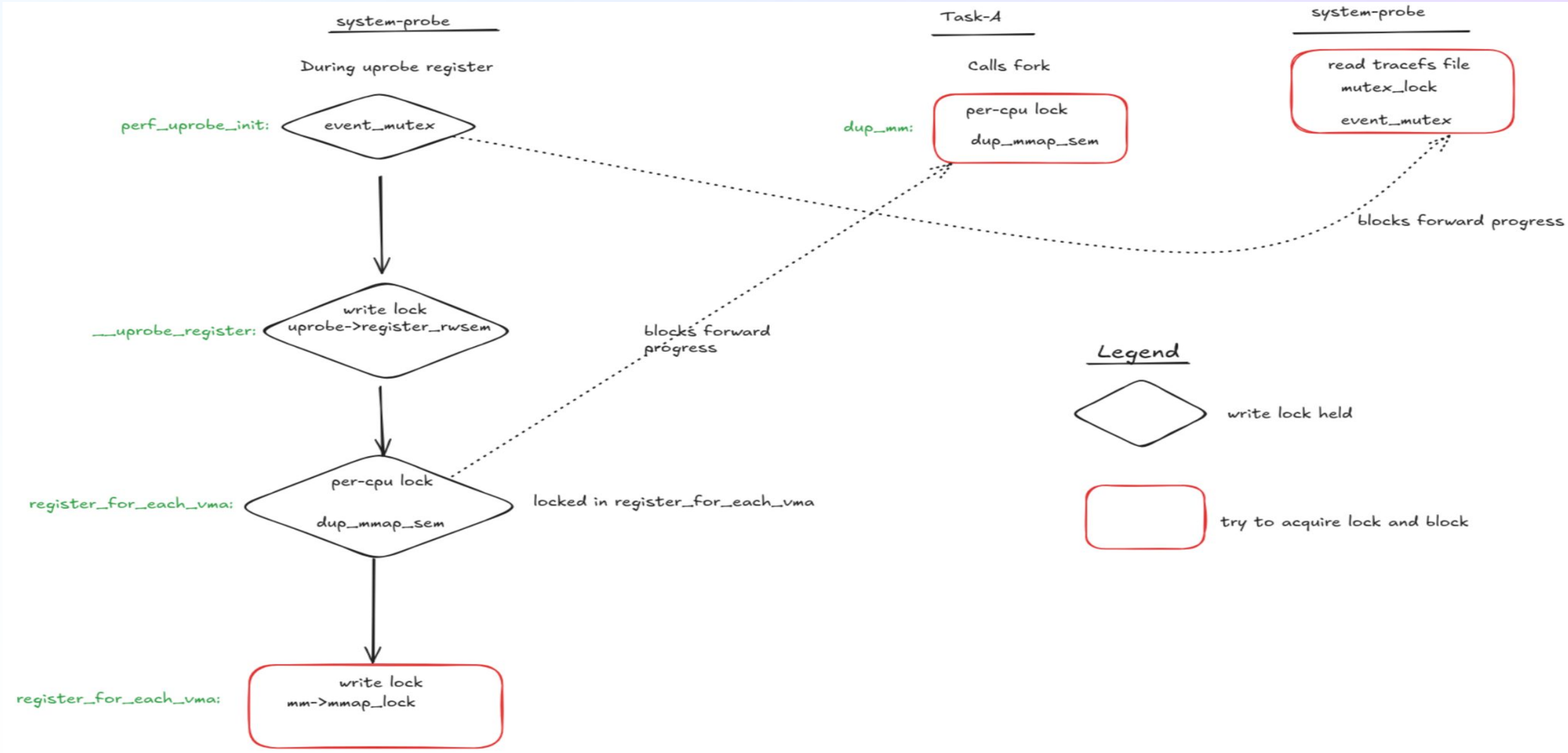


# Production impact from uprobe - Uprobe Registration

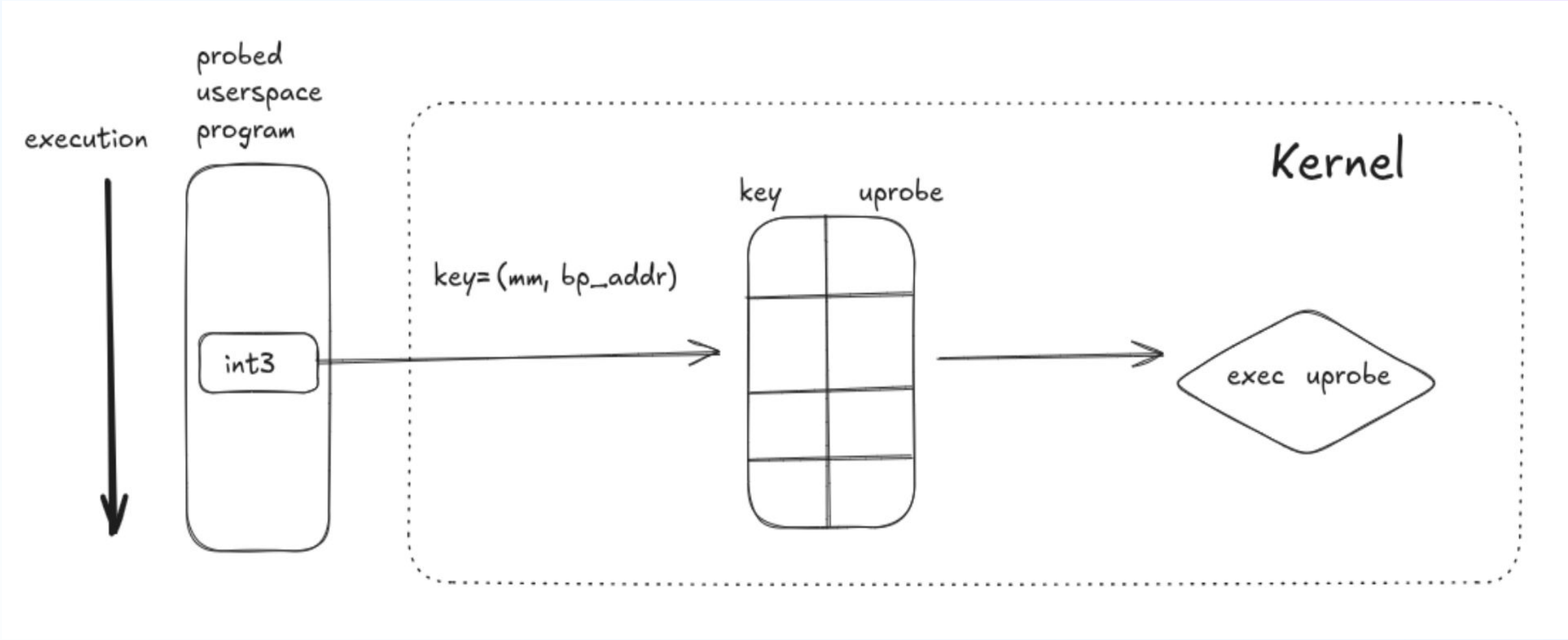
```
__uprobe_register
   down_write(&uprobe->register-rwsem) // write lock on semaphore
   register_for_each_vma
      percpu_down_write(&dup_mmap_sem) // write lock on semaphore
      for each interesting MM:
         mmap_write_lock(mm) // write lock on mmap_lock semaphore
         do_work();
         mm_write_unlock(mm) // unlock mmap_lock semaphore
```

**Mmap lock is required to synchronize with uprobe firing, incase the uprobe is being removed.**

# Production impact from uprobe - Uprobe Registration



# Production impact from uprobe - Uprobe Firing



# Production impact from uprobe - Uprobe Firing

- Larger number of tasks blocked on find_active_uprobe
- Blocked on acquire read lock on mmap_lock
- Lots of application threads piling up waiting on this lock

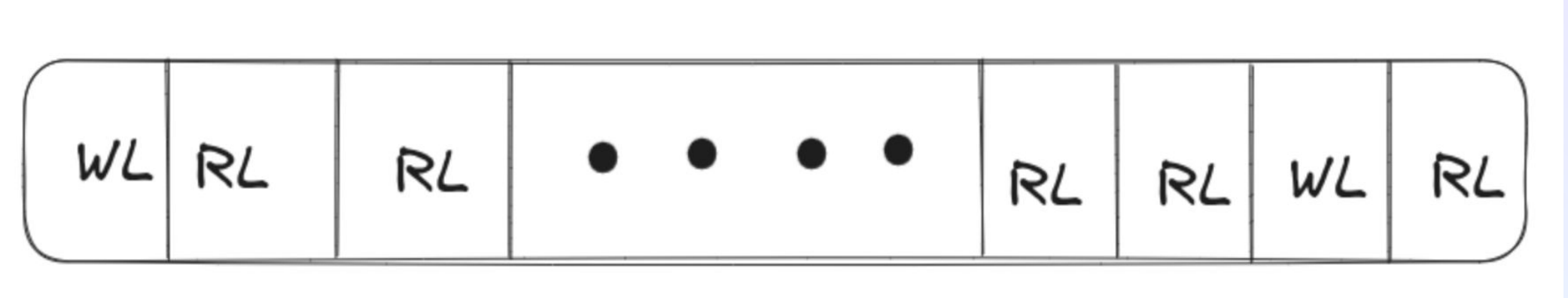
# Production impact from uprobe - mmap_lock & rw_semaphore

- Allows concurrent readers but exclusive writers

- Use contention tracepoints to build model
  - If reader is blocked, why?
    - Can we get the owner?
  - If write is blocked, why?
    - Reader owned or writer owned?
    - Can we get the owner?
  - Depth of wait queue
  - Contention duration
  - When is forward progress made

# Production impact from uprobe - Root Cause

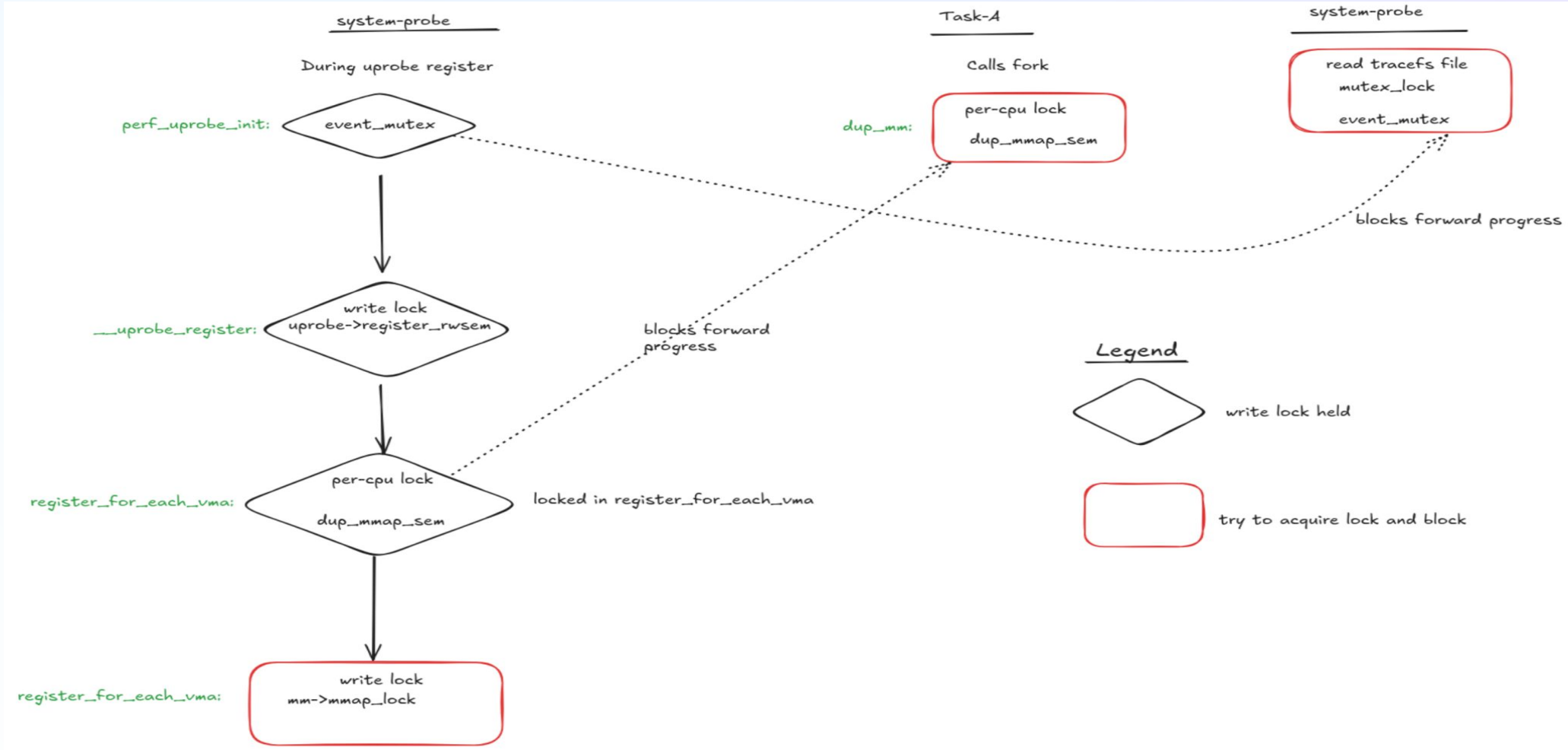
- Deep queue of mmap read locks interspersed with writes
  - Write from mmap, madvise, uprobe register




- Rw_semaphore wakes up only max 250 reader threads
  - To avoid thundering herd
  - But this means even batches of reads become serialized
  - No concurrent reader if writer at head of wait queue
    - Avoid writer starvation

# Production impact from uprobe - Root Cause

- If a uprobe_register write_lock is blocked then whole system bricks



# Production impact from uprobe - Root Cause

- If a uprobe firing read_lock is blocked then applicate suffer
  - Due to piling threads stuck in the kernel

# Production impact from uprobe - Kernel Fix

## Speculative mmap_lock acquisition

```
parent 83e3dc9a5d4d7402adb24090a77327245d593129
author Andrii Nakryiko <andrii@kernel.org> Thu Nov 21 19:59:22 2024 -0800
committer Peter Zijlstra <peterz@infradead.org> Mon Dec 2 12:01:38 2024 +0100

uprobes: add speculative lockless VMA-to-inode-to-uprobe resolution

Given filp_cachep is marked SLAB_TYPESAFE_BY_RCU (and FMODE_BACKING
files, a special case, now goes through RCU-delated freeing), we can
safely access vma->vm_file->f_inode field locklessly under just
rcu_read_lock() protection, which enables looking up uprobe from
uprobes_tree completely locklessly and speculatively without the need to
acquire mmap_lock for reads. In most cases, anyway, assuming that there
are no parallel mm and/or VMA modifications. The underlying struct
file's memory won't go away from under us (even if struct file can be
reused in the meantime).

We rely on newly added mmap_lock_speculate_{try_begin,retry}() helpers to
validate that mm_struct stays intact for entire duration of this
speculation. If not, we fall back to mmap_lock-protected lookup.
The speculative logic is written in such a way that it will safely
handle any garbage values that might be read from vma or file structs.
```

# Thank you

Questions?

DATADOG