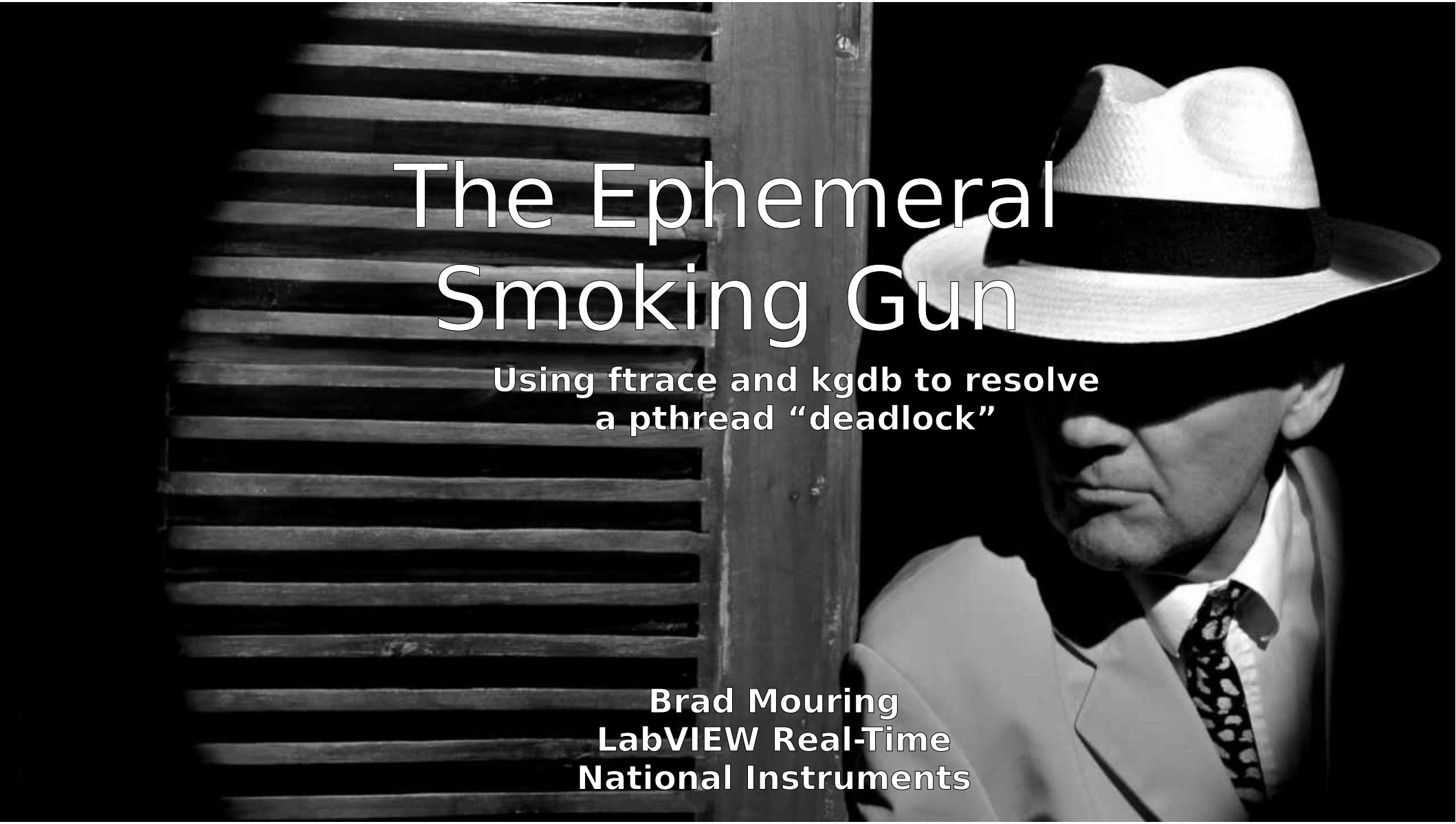


The Ephemeral Smoking Gun

Using ftrace and kgdb to resolve a pthread “deadlock”

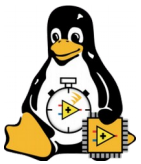
Brad Mouring
LabVIEW Real-Time
National Instruments

The Intro – Who am I

- Work at National Instruments in the RTOS R&D group
  - Multiple product lines use RTOS
- NI Switched to Linux 2-3 yrs ago
  - Single-mode RTOS ⇒ Dual-mode RTOS
  - Functionality and support that comes
  - Mindset within company about FOSS
    - Work with maintainers, minimize out-of-tree

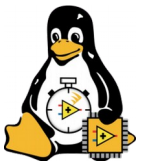
The Setup – Crashing Application

- User-mode application crashed after a few hours of running
- The clincher: new issue from existing code
  - The same application ran continually without issue on older, singlemode RTOS



Initial Investigation

- Configured to provide a core file on crash
- Checking the core file fingered a SIGABRT
  - Normally used for assert() and critical errors
  - Coming from glibc, __pthread_mutex_lock_full()
- To enable: ulimit -c ${blocks}
  - May need to edit /etc/security/limits.conf
  - Can set in the /etc/profile(.d/*)



Digging In Further

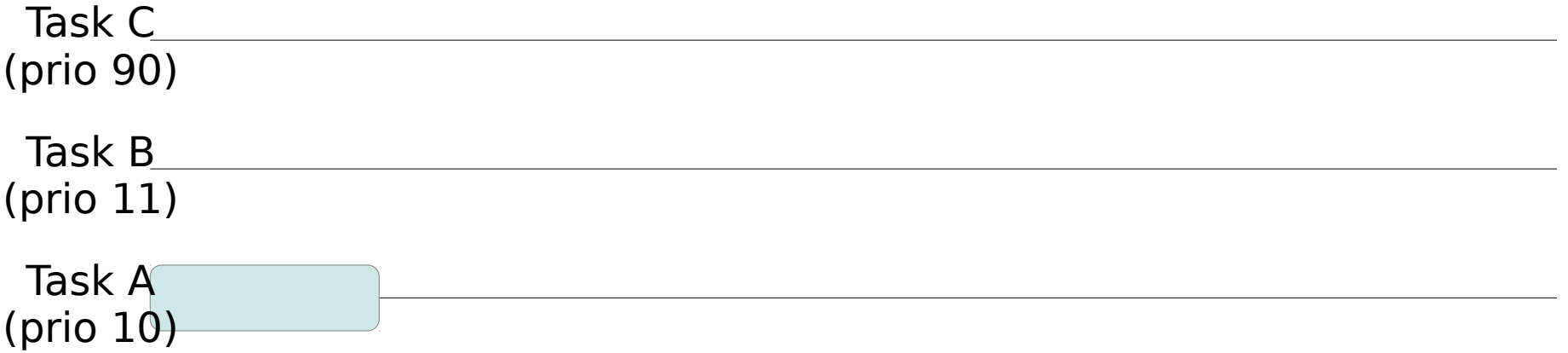
- Reproduced the issue checking stderr
  - “pthread_mutex_lock.c:309: Assertion `...' failed.”
  - Points me to a file and line number
  - Assertion is checking the return from a futex syscall
    - Checking for a reported deadlock on certain lock types

Background: Pthread_mutexes

- Mutexes used to protect a few different application execution system state structures
- The application uses pthread_mutex_t's configured to be priority-inheriting

Background: Priority Inversion

A is running on the processor



Background: Priority Inversion

A is running on the processor
A takes mutex M

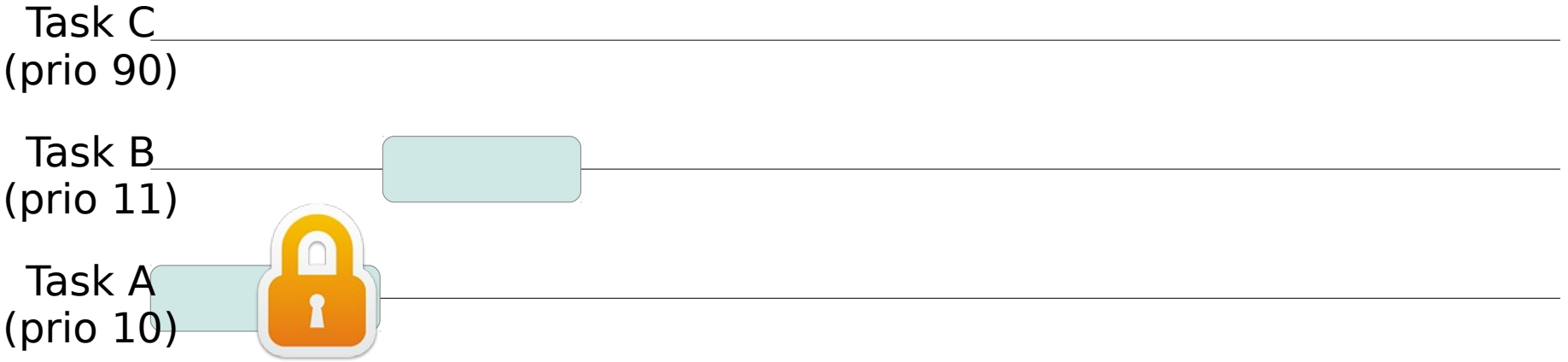
Task C (prio 90)

Task B (prio 11)

Task A (prio 10)

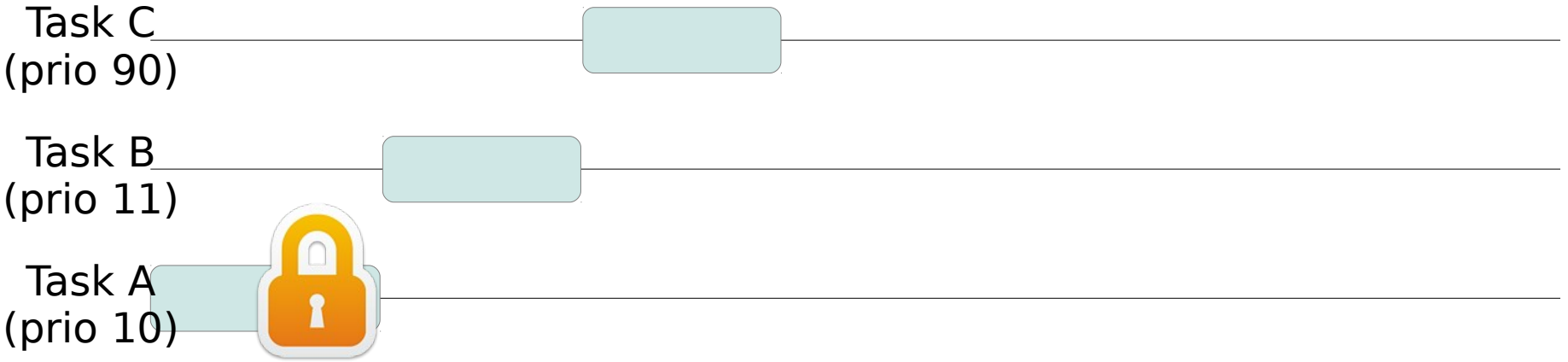
Background: Priority Inversion

A is running on the processor
A takes mutex M
B becomes runnable, is scheduled in



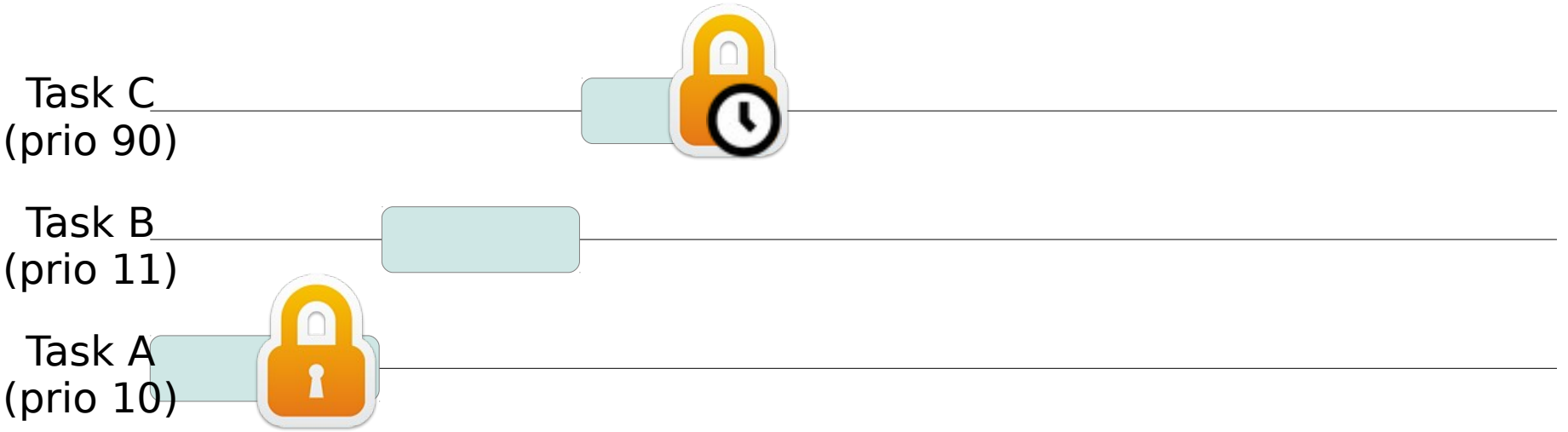
Background: Priority Inversion

A is running on the processor
A takes mutex M
B becomes runnable, is scheduled in
C becomes runnable, is scheduled in



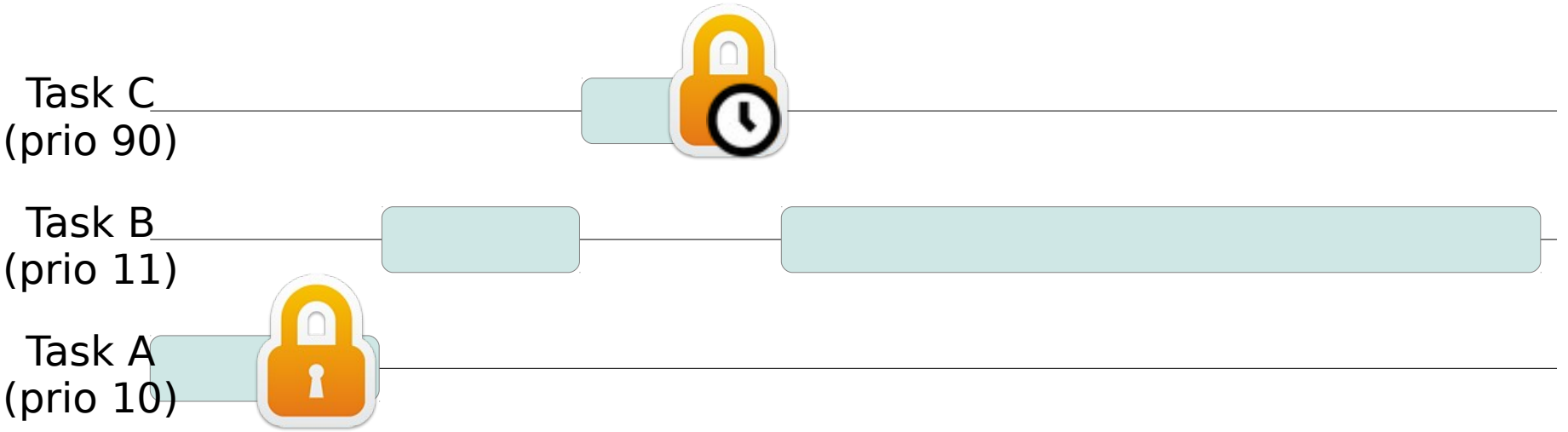
Background: Priority Inversion

A is running on the processor
A takes mutex M
B becomes runnable, is scheduled in
C becomes runnable, is scheduled in
C blocks on mutex M



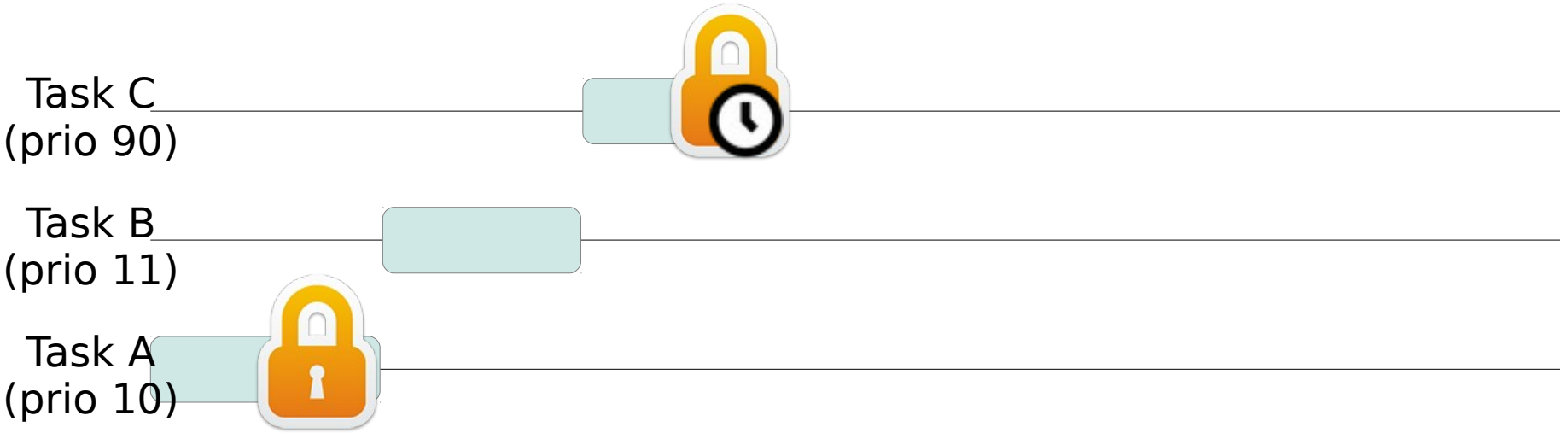
Background: Priority Inversion

A is running on the processor
A takes mutex M
B becomes runnable, is scheduled in
C becomes runnable, is scheduled in
C blocks on mutex M
B is scheduled in (prio 11), blocking C (prio 90) from running!



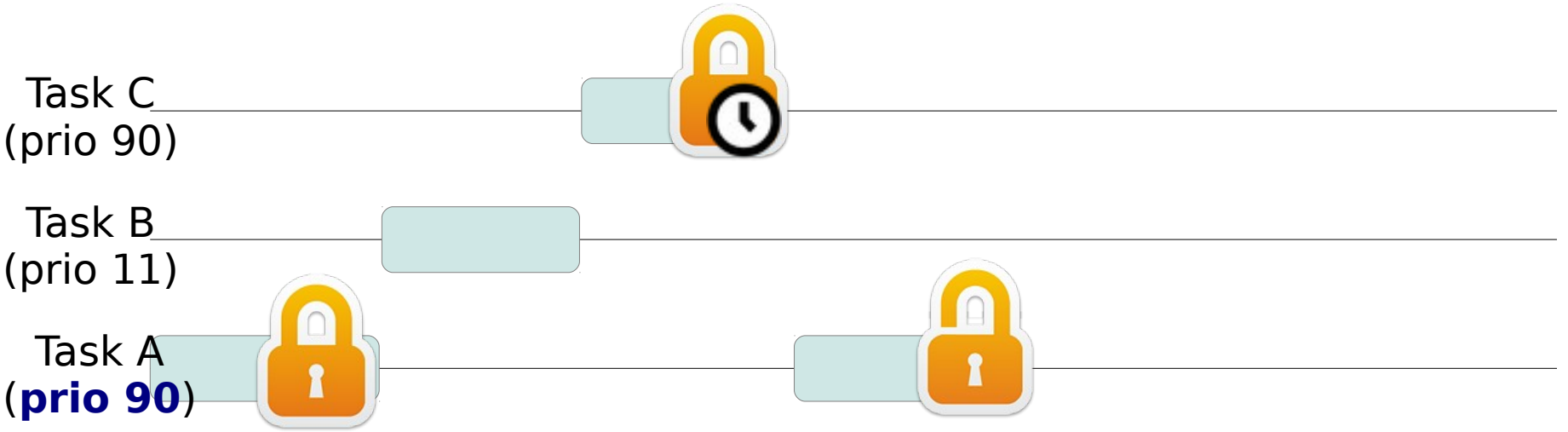
Background: Priority Inheritance A Solution

A is running on the processor
A takes mutex M
B becomes runnable, is scheduled in
C becomes runnable, is scheduled in
C blocks on mutex M



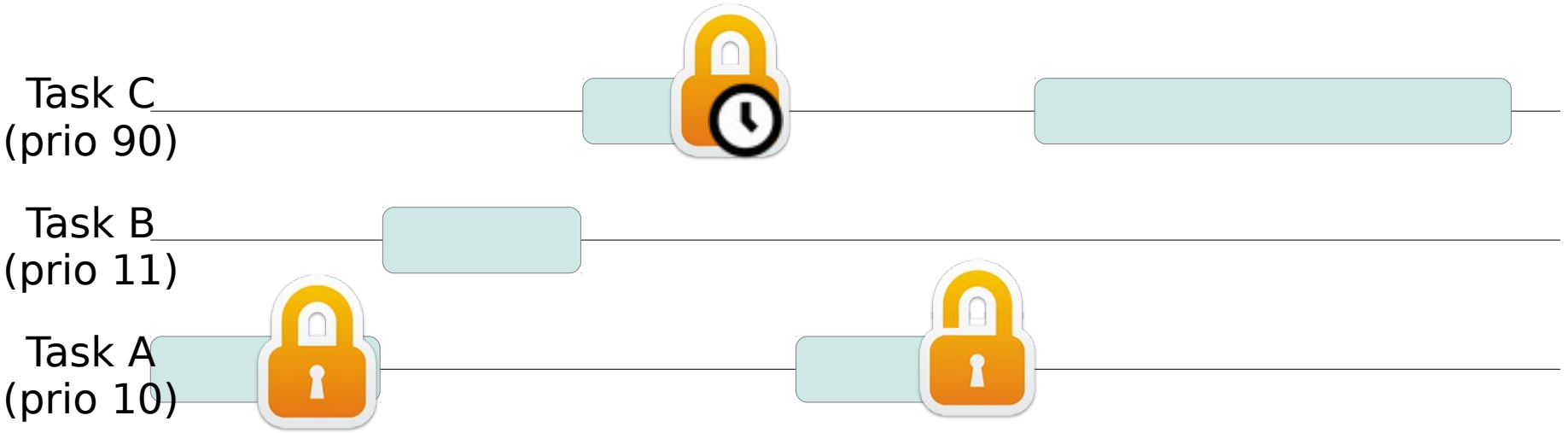
Background: Priority Inheritance A Solution

A is running on the processor
A takes mutex M
B becomes runnable, is scheduled in
C becomes runnable, is scheduled in
C blocks on mutex M
A receives C's priority, finishes with mutex M, releases M



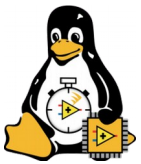
Background: Priority Inheritance A Solution

A is running on the processor
A takes mutex M
B becomes runnable, is scheduled in
C becomes runnable, is scheduled in
C blocks on mutex M
A receives C's priority, finishes with mutex M, releases M
A receives its previous priority, C is scheduled in



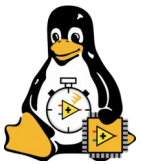
Background: Pthread_mutexes and Futexes

- pthread_mutex use futexes when contended
- Uncontested lock stays in userspace (cmpxchg)
- Uses the kernel sys_futex call if contested
  - Creates a queue of tasks to wake when the holder releases the lock (FUTEX_LOCK_PI)
  - Sits atop rtmutex code within the kernel
  - On release, previous holder notes that there are waiters, wakes one or more (FUTEX_UNLOCK_PI)
  - The underlying rt_mutex subsystem provides some nice features (deadlock detection, e.g.)



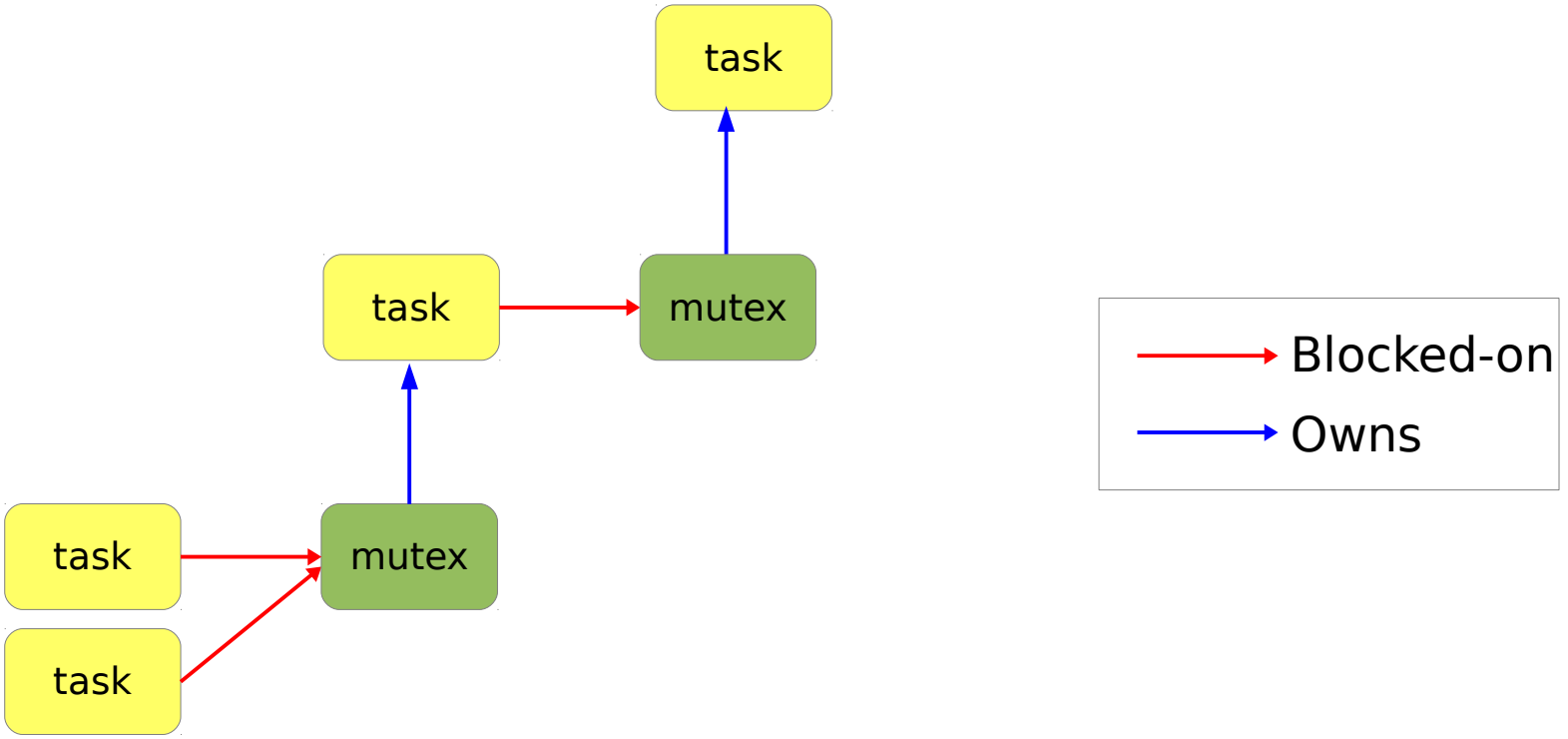
Background: RT Mutexes

- RT Mutexes were designed the linux-rt tree
  - Used to silently replace normal spinlocks
- Sold to mainline as a solution for prio inversion through futexes
- Prio inheritance is attained through tracking:
  - The tasks blocked on a mutex (sorted by prio)
  - The task that owns a mutex



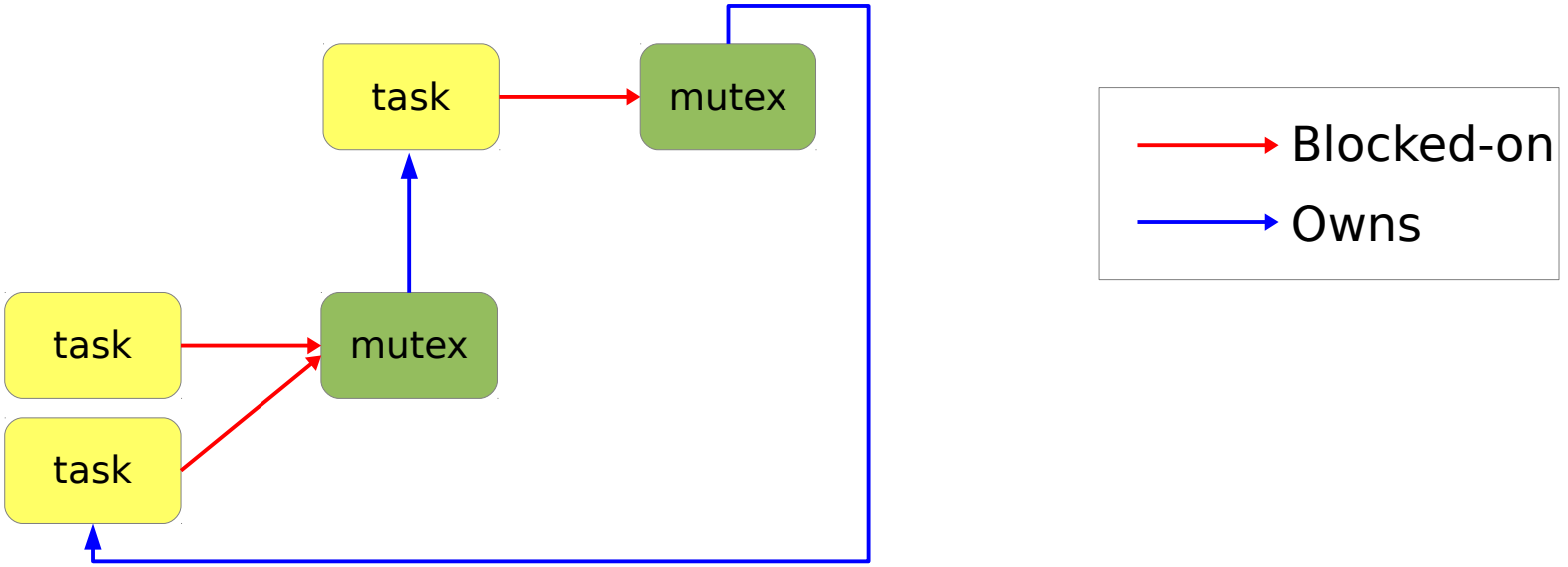
Background: RT Mutexes Visually

- These relationships allow for prio inheritance



Background: RT Mutexes Visually

- These relationships allow for prio inheritance
  - Also handy for checking for deadlocks



How to debug, and where?

- EDEADLK returned in a few locations, including a few in futex/mutex/rtmutex code
- Place a kgdb_breakpoint at these sites
- Build a kernel with kgdb enabled
- Reproduce the issue
- Troubleshoot from there

Background: How to enable KGDB

- Configure the kernel
  - CONFIG_DEBUG_INFO
  - CONFIG_KGDB
  - CONFIG_KGDB_*method_to_connect*
    - *e.g. CONFIG_KGDB_SERIAL_CONSOLE*
  - CONFIG_KGDB_KDB (optional)

Background: Connecting to a KGDB target

- You have a few options
  - Serial port (null-modem connection)
  - Over Ethernet (kgdboe) with out-of-tree source[1]
- Set module params on boot, on module load, or thereafter through sysfs
  - Port and baud

[1]http://sysprogs.com/VisualKernel/kgdboe/

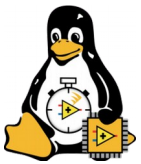
Tips for using kgdb/gdb

- Find (or write) useful user-defined cmds
  - Sequences you use frequently
- Pop cmds and settings in your ~/.gdbinit
- Graphical frontends are available
- Excellent resources online
  - https://sourceware.org/gdb/current/onlinedocs/gdb/

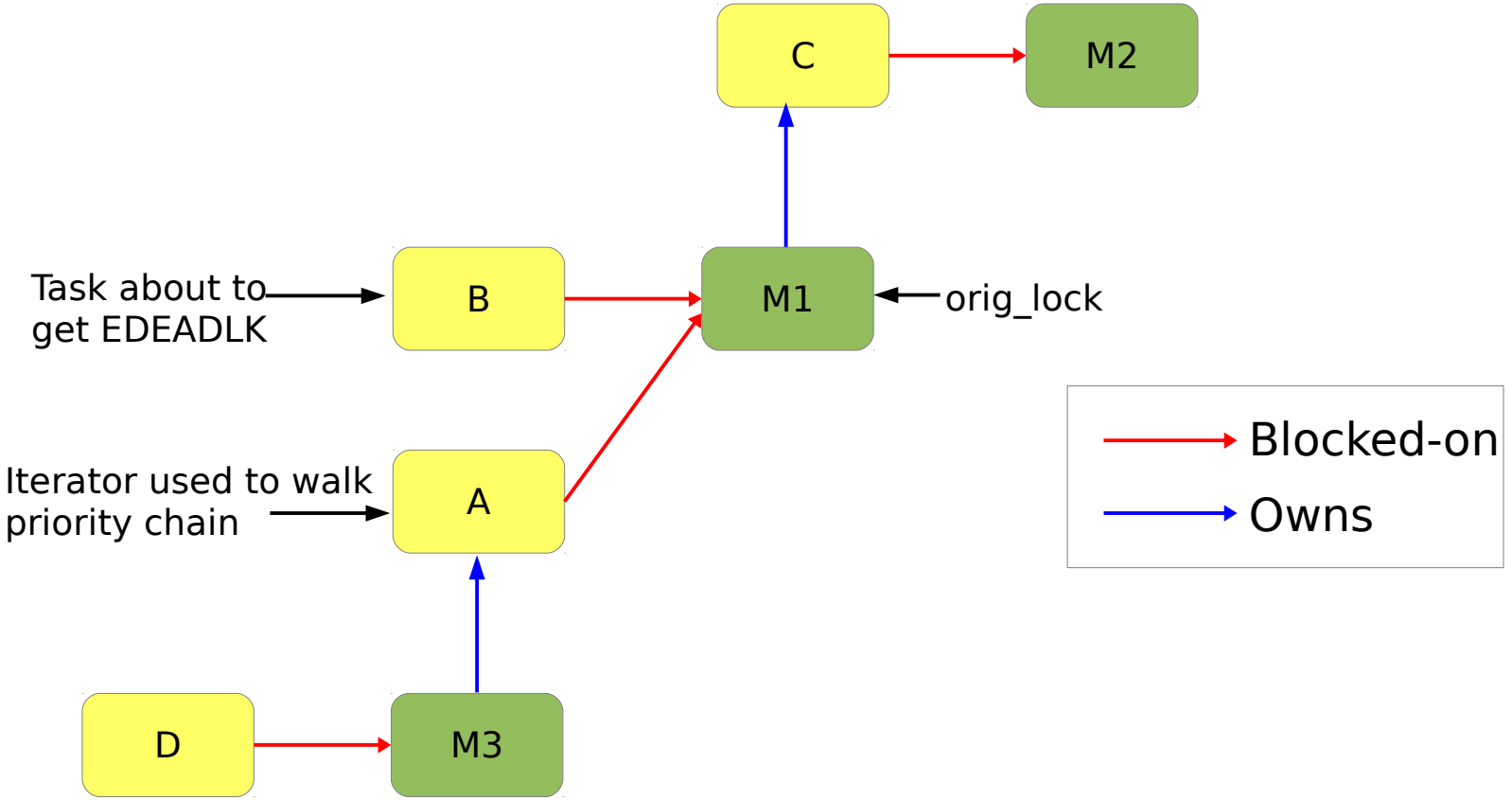
KGDB leads to a dead end

... and that's not necessarily a bad thing.

- EDEADLK came from rtmutex priority chain walking code (rt_mutex_adjust_prio_chain)
  - The priochain walking code seemed to think that we had a loop in the chain
  - Walking the chain manually in gdb from the original mutex, we reach a mutex who has no owner
  - We were supposed to loop back around to the original mutex, as that's the current state of the pointers within the chain walking function



State of the Priority Chain at EDEADLK

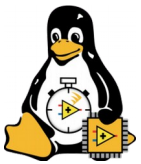


A Few Clues From the Scene of the Crime

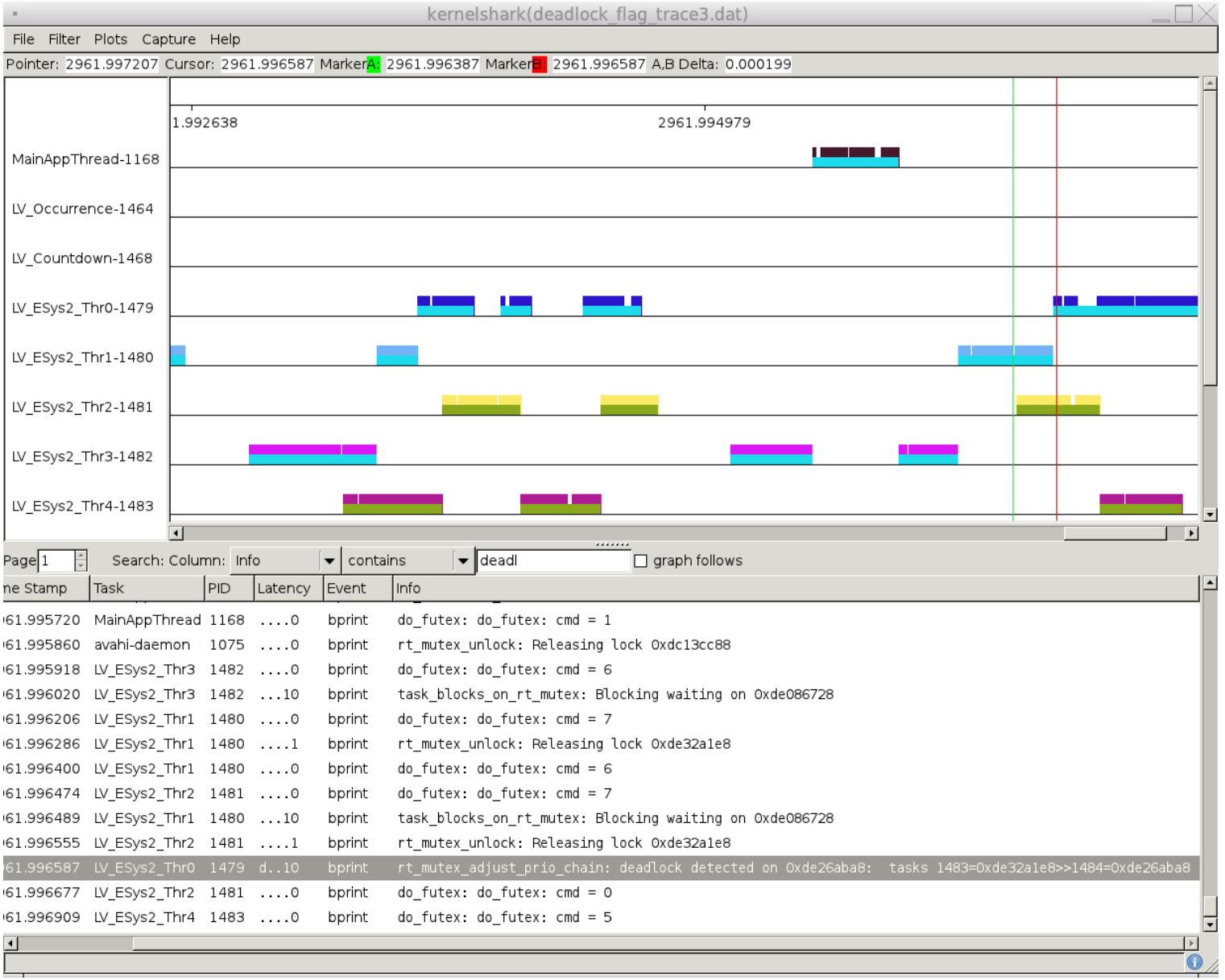
- Mutex M2 recently had an owner but doesn't currently
- There are two tasks (A, B) blocked on mutex M1
- The checks that occur while walking the chain don't see anything odd and complain until a deadlock is detected

Re-ftrace-ing my steps

- A picture of what's going on leading up to the detected deadlock may shed some light into what's going on
- Ftrace and a set of tracers were already enabled on our kernel
- Insert some strategic trace_printk()s
- Add SIGABRT handler to app to stop tracing
- Reproduce the issue, use trace-cmd extract



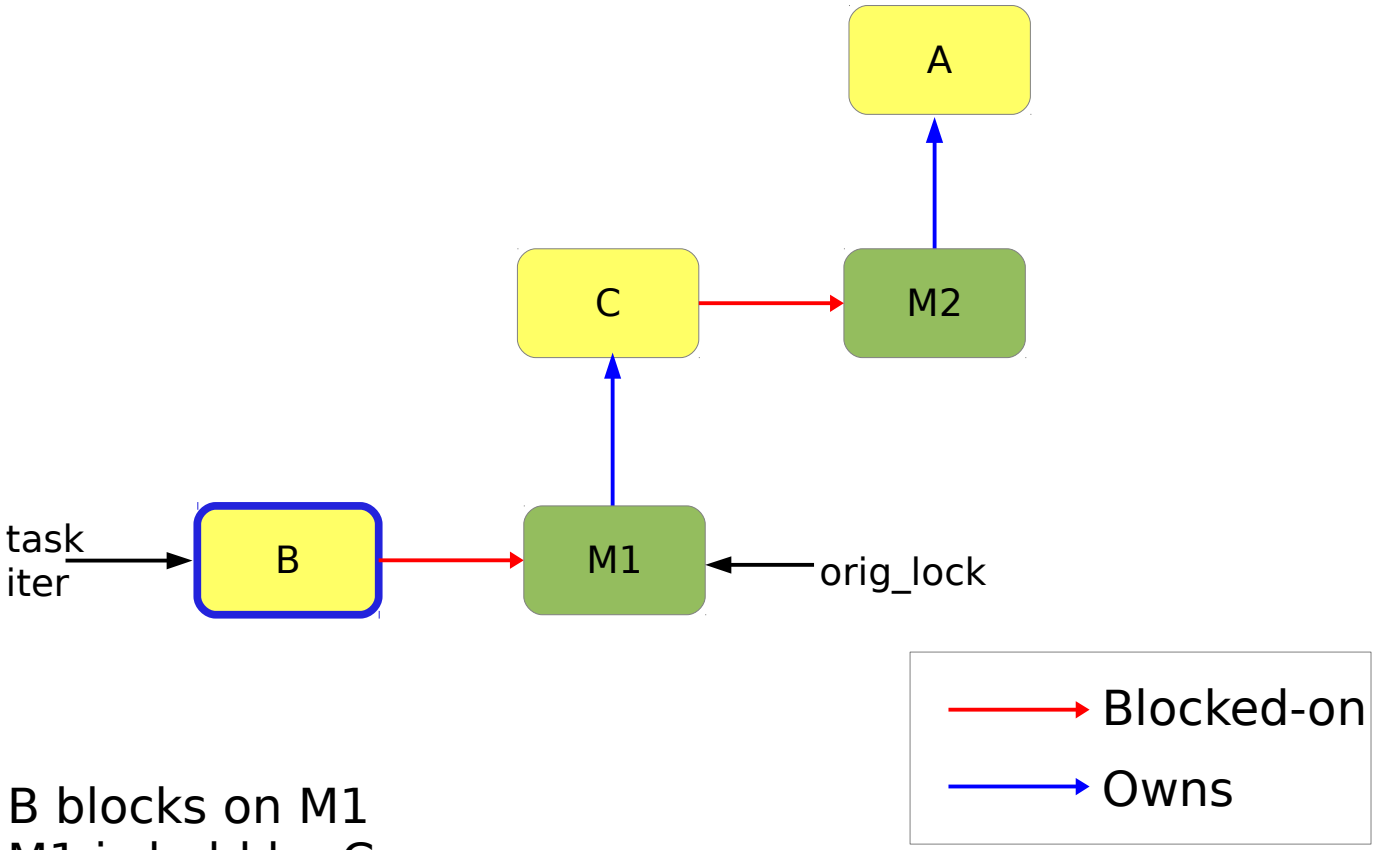
kernelshark comes into the picture



kernelshark comes into the picture

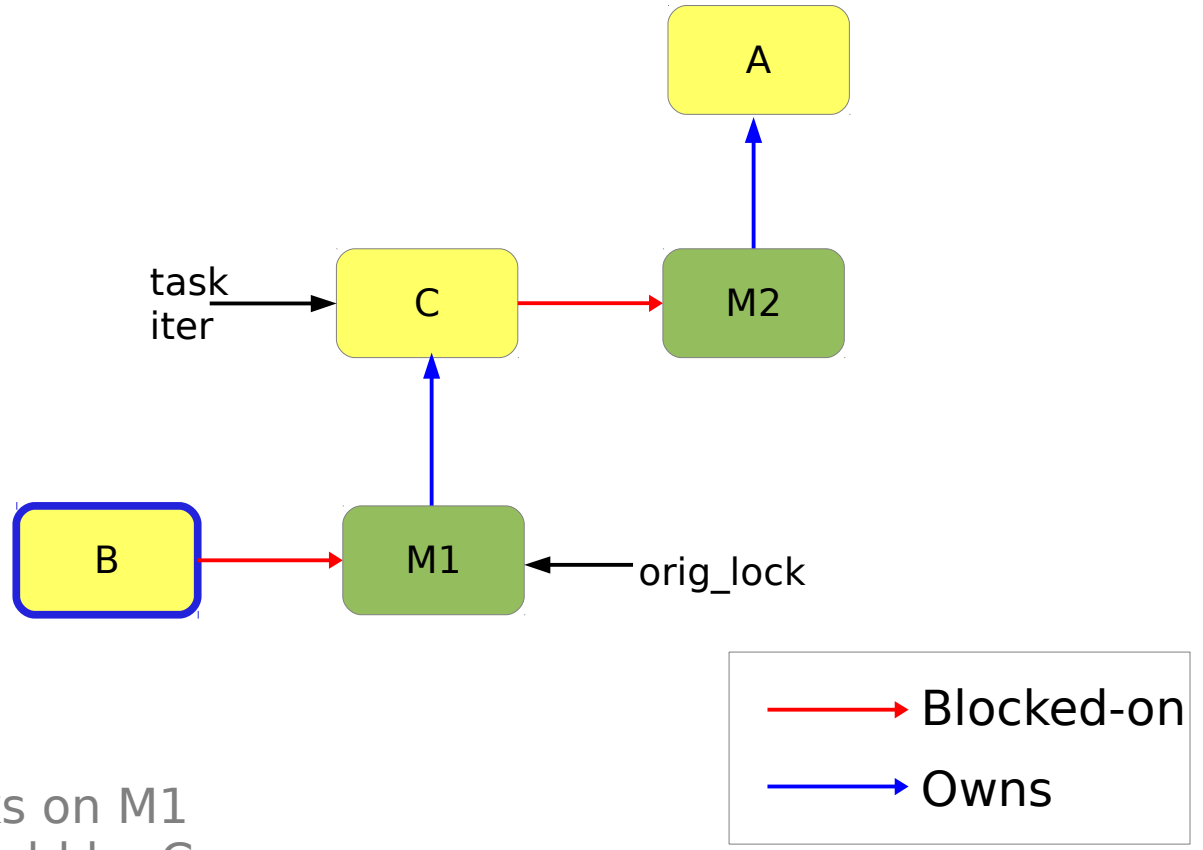
- Pulling the dump into kernelshark to take a closer look, we notice a few interesting points
  - Task 'B' (received EDEADLK) scheduled out between attempting to take mutex and reporting EDEADLK
  - Quite a bit of mutex activity while B is out
- We can begin to form a narrative on what is happening leading up to the reported deadlock

Re-ftrace-ing my steps



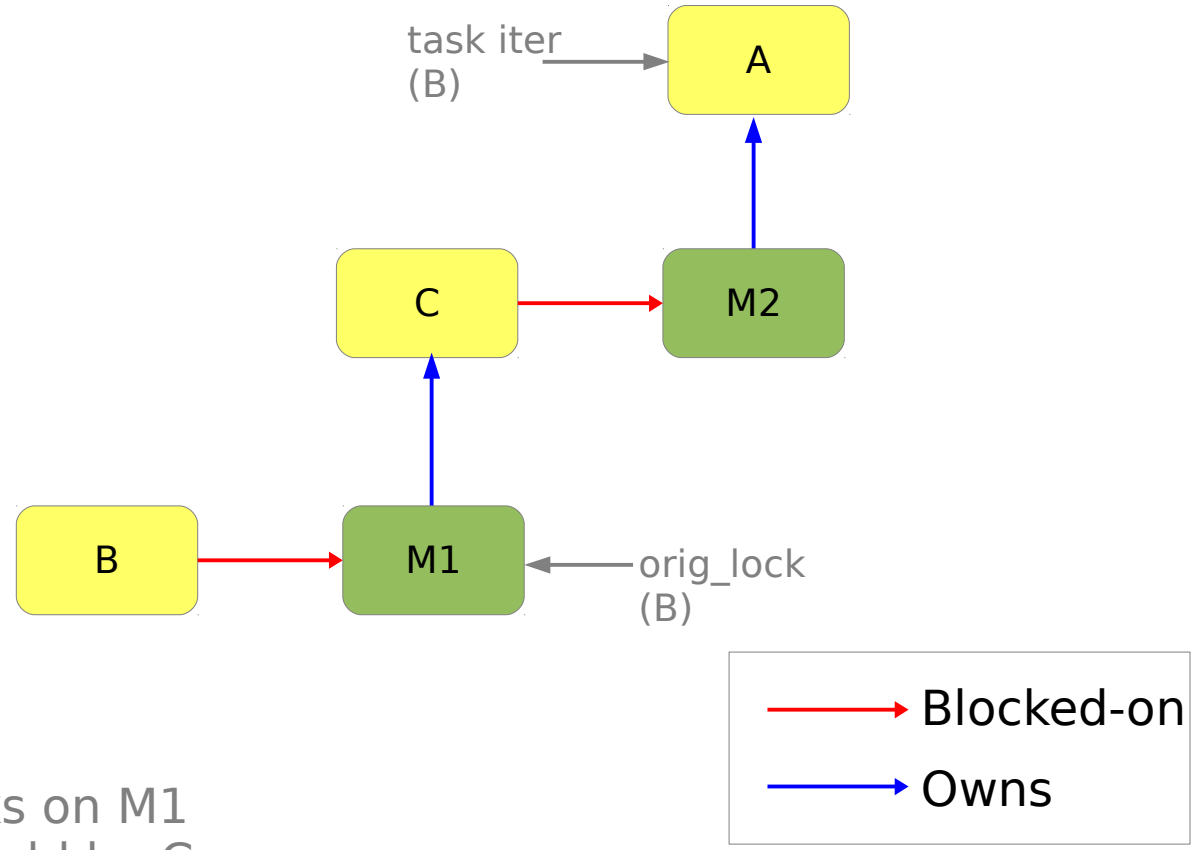
B blocks on M1
M1 is held by C
C is blocked on M2
M2 is held by A

Re-ftrace-ing my steps



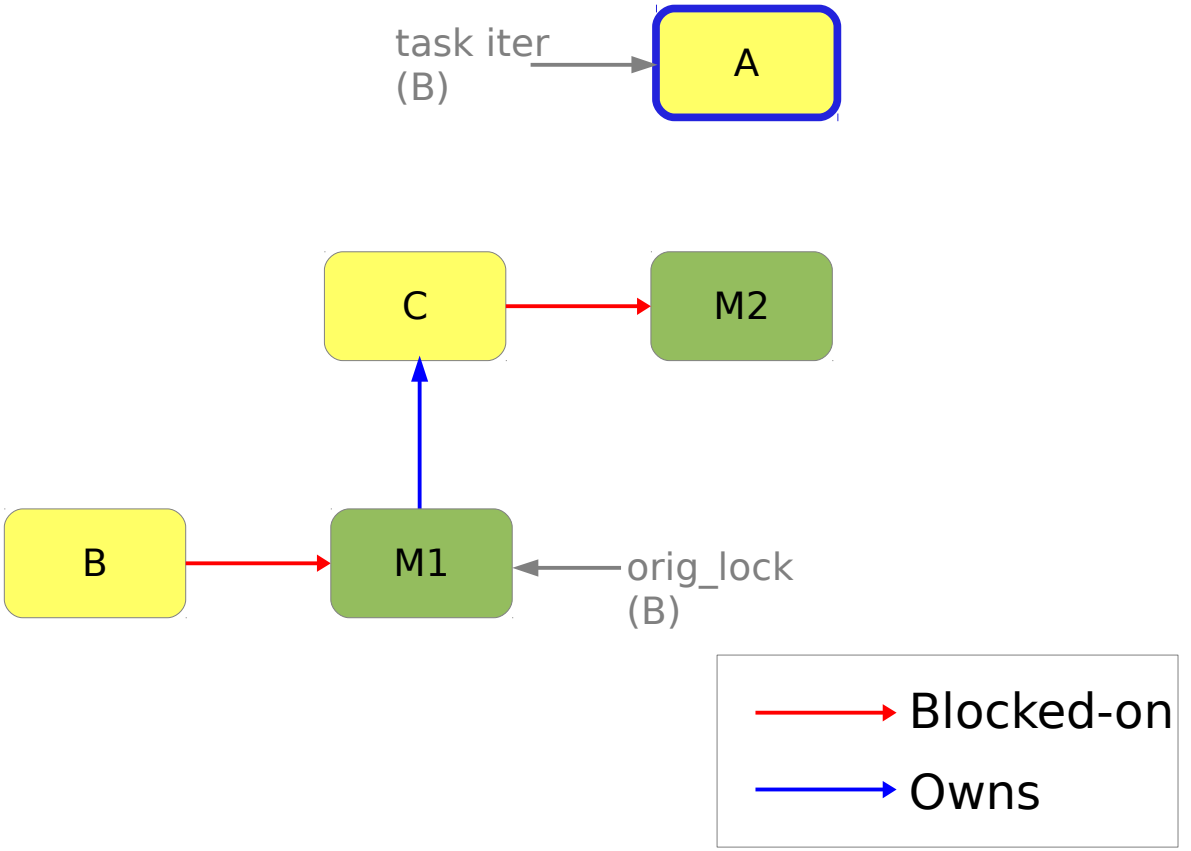
B blocks on M1
M1 is held by C
C is blocked on M2
M2 is held by A
B begins walking the prio chain

Re-ftrace-ing my steps



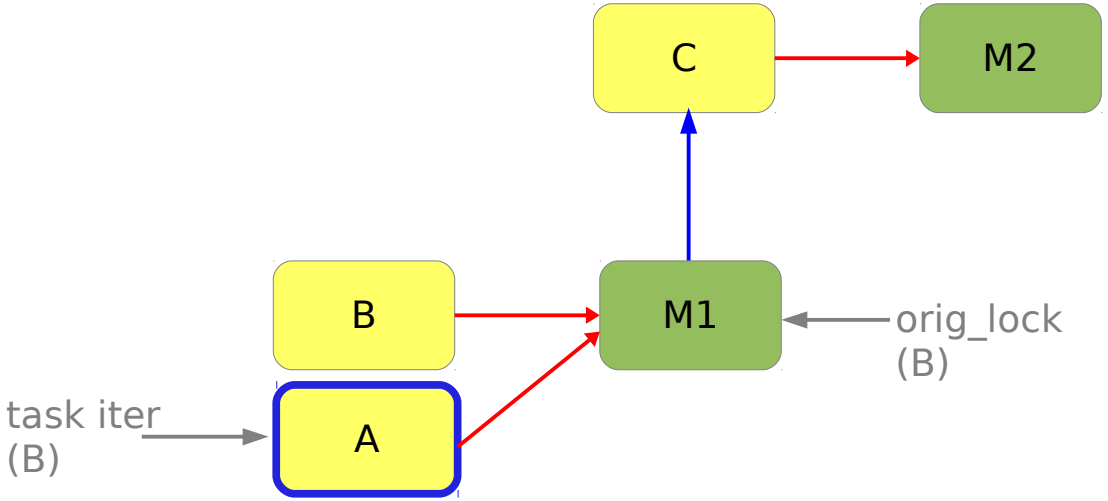
B blocks on M1
M1 is held by C
C is blocked on M2
M2 is held by A
B begins walking the prio chain...PREEMPT!

Re-ftrace-ing my steps



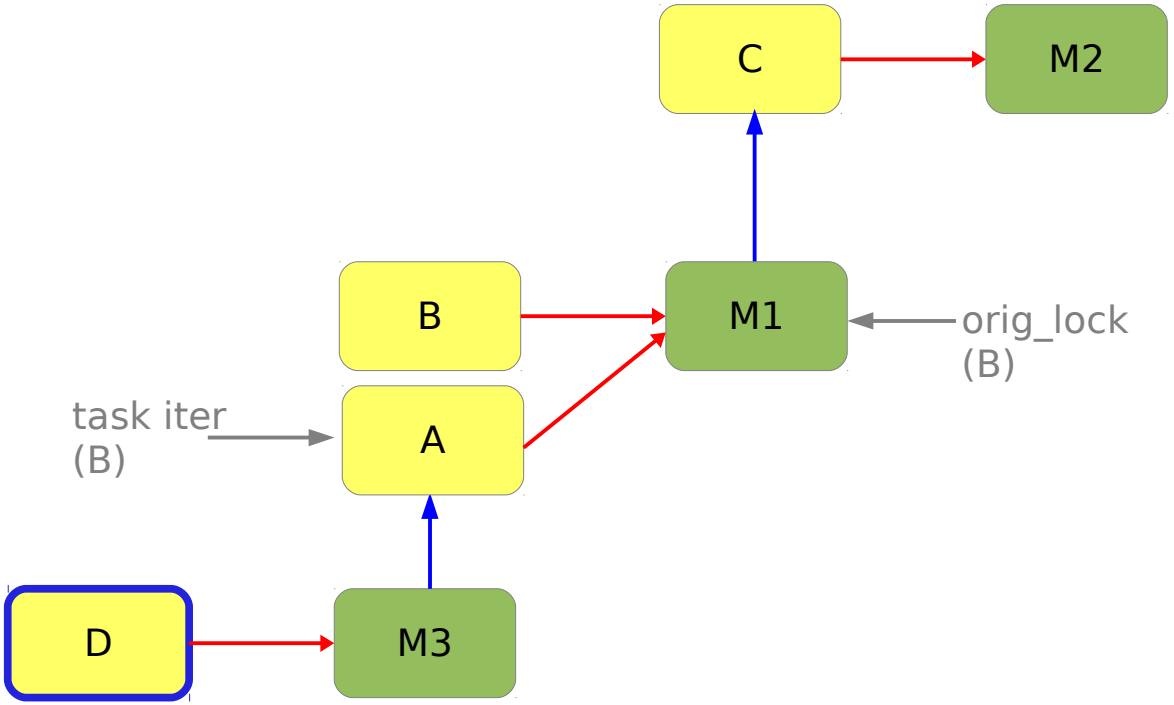
A is scheduled in, releases M2

Re-ftrace-ing my steps



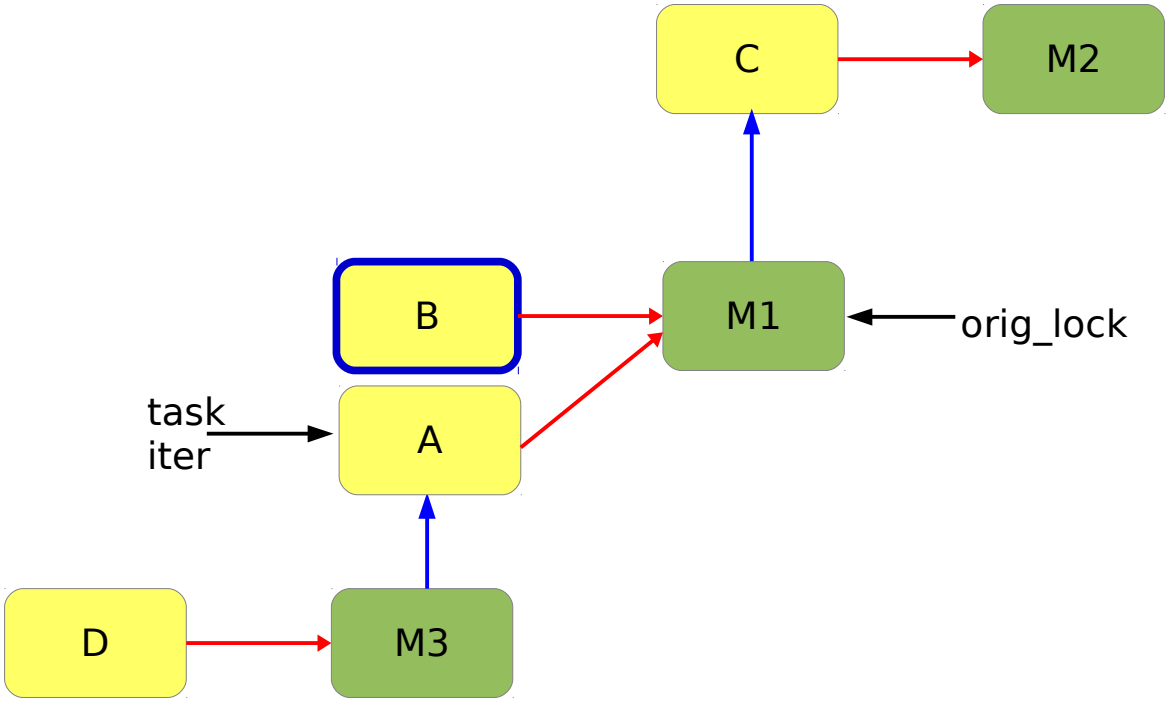
A is scheduled in, releases M2
A takes (uncontended) M3 in userspace
A blocks on M1

Re-ftrace-ing my steps



D is scheduled in, blocks on M3 (creates rtmutex)

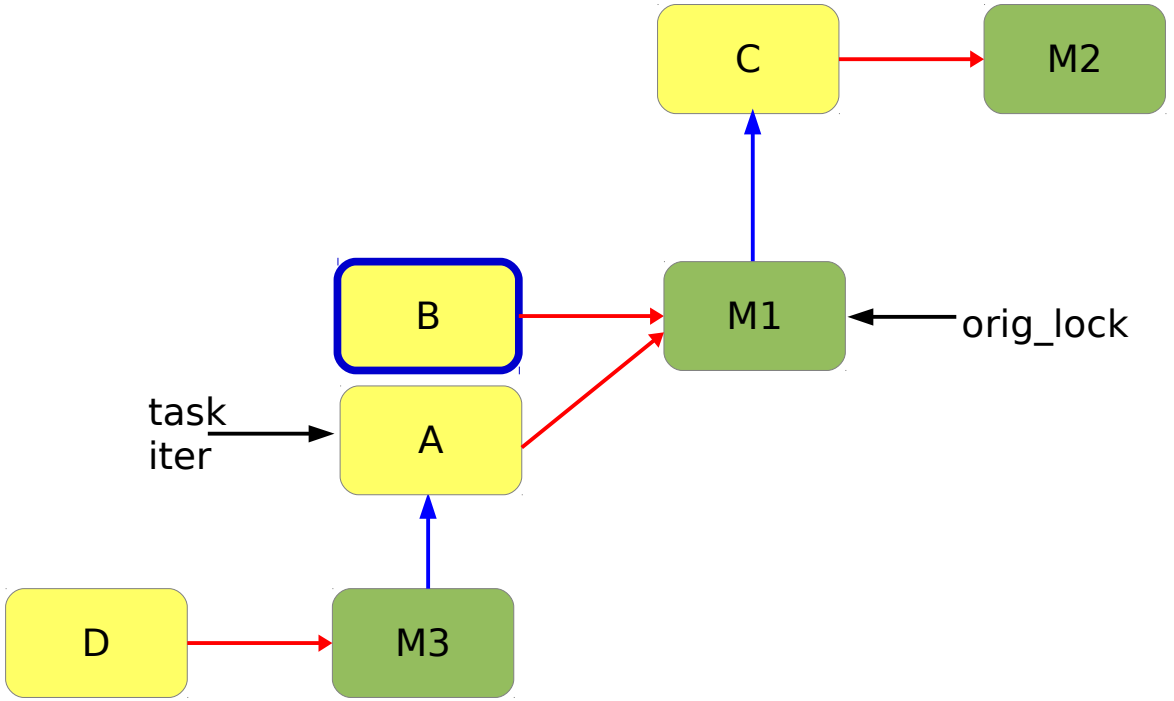
Re-ftrace-ing my steps



B is scheduled back in, continues its walk of the prio chain

Re-ftrace-ing my steps

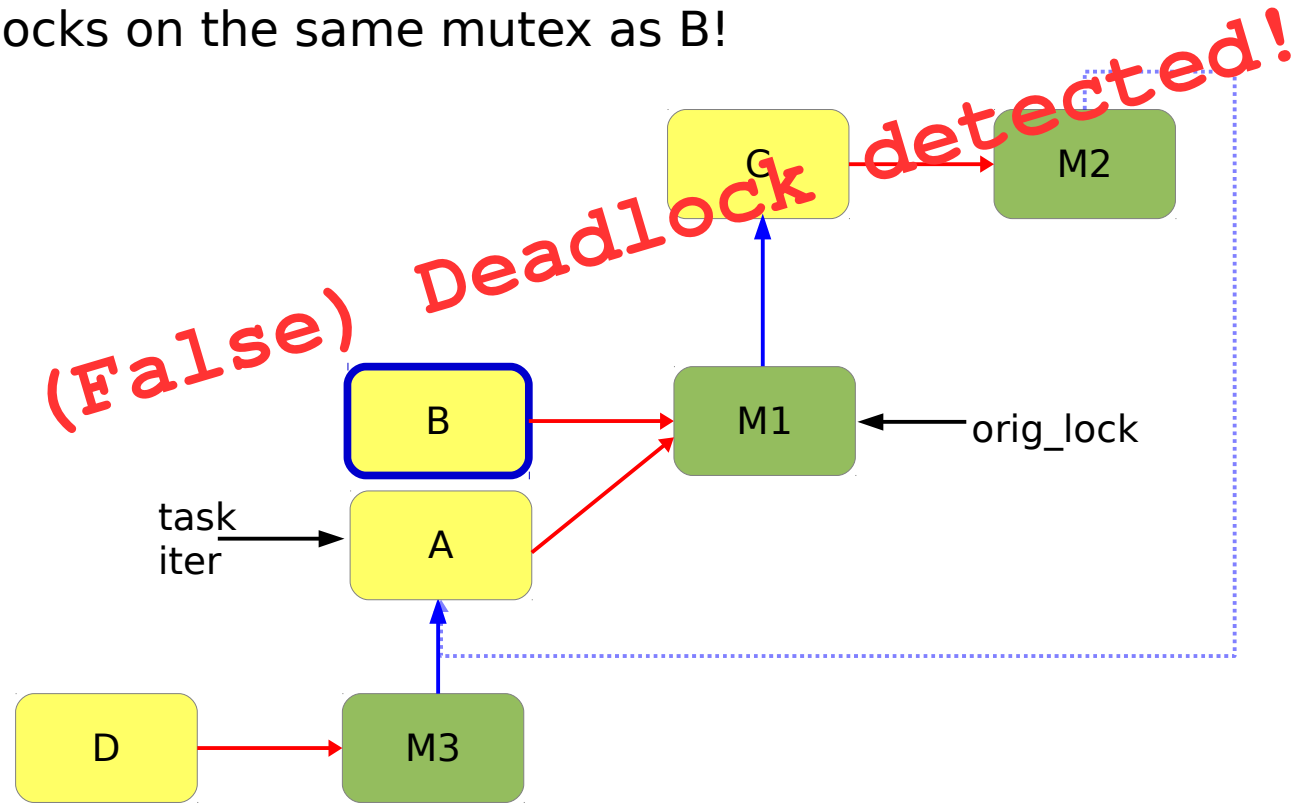
A blocks on the same mutex as B!



B is scheduled back in, continues its walk of the prio chain

Re-ftrace-ing my steps

A blocks on the same mutex as B!



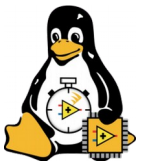
B is scheduled back in, continues its walk of the prio chain

Takin' it to the Streets

- Came to the linux-rt-users mailing list
- Had findings writeup, preliminary patch
- tglx saw the issue at hand, didn't like my patch, proposed a different fix
- Result: issue got fixed, learned about working with the mailing lists
- Moral: Don't be afraid to engage the community, but be ready for feedback

Conclusions

- There are some great tools (and online documentation) to solve kernel issues
- I've only covered two, there are many more
  - Lockdep checking
  - RCU diagnostics
  - kdump kernel(s)
  - KDB
  - Vendor tools



Questions?
Comments?
Thanks!