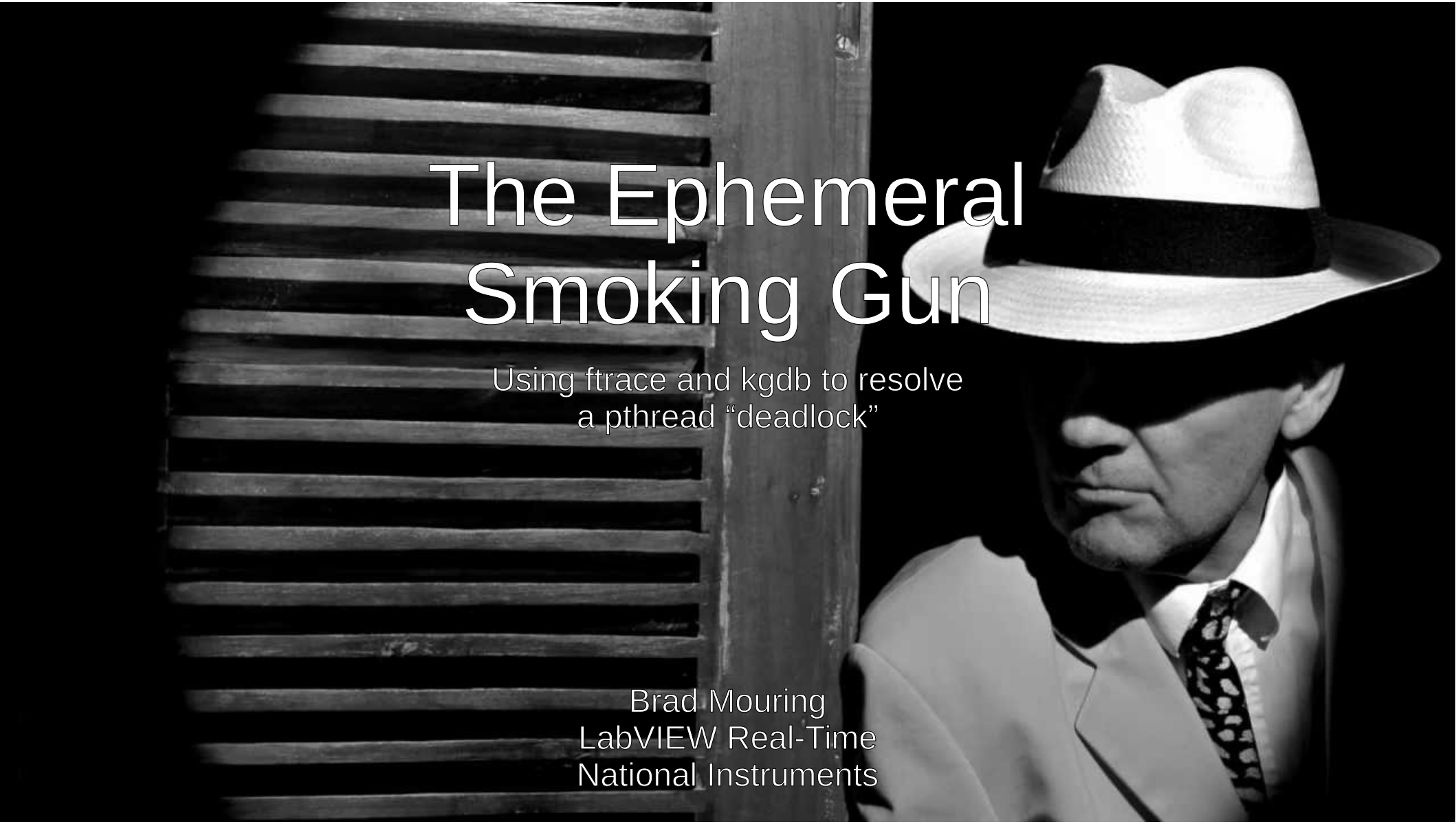


The Ephemeral Smoking Gun

Using ftrace and kgdb to resolve
a pthread “deadlock”

Brad Mouring
LabVIEW Real-Time
National Instruments

The Setup

- Customer application crashed after a few hours

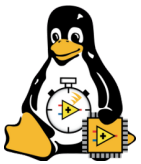
- The clincher: new issue from existing code
  - Run without issue on older, singlemode RTOS

Initial Investigation

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

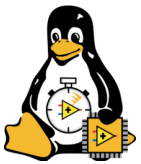
Digging in Further

- Reproducing the issue with console enabled
  - “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



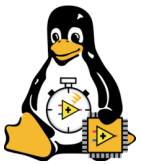
Pthread_mutexes and futexes and contested locks (oh my!)

- pthread_mutex configured to be priority-inheriting
- Uncontested lock stays in US (cmpxchg)
- Uses the kernel sys_futux call if contested
  - Creates a futex queue of tasks to wake when the holder releases the lock (FUTEX_WAIT)
  - Sits atop rtmutex code within the kernel
  - On release, previous holder notes that there are waiters, wakes one or more (FUTEX_WAKE)
  - The underlying rt_mutex subsystem provides some nice features (deadlock detection, e.g.)
- Time to don the waders, we're going in...



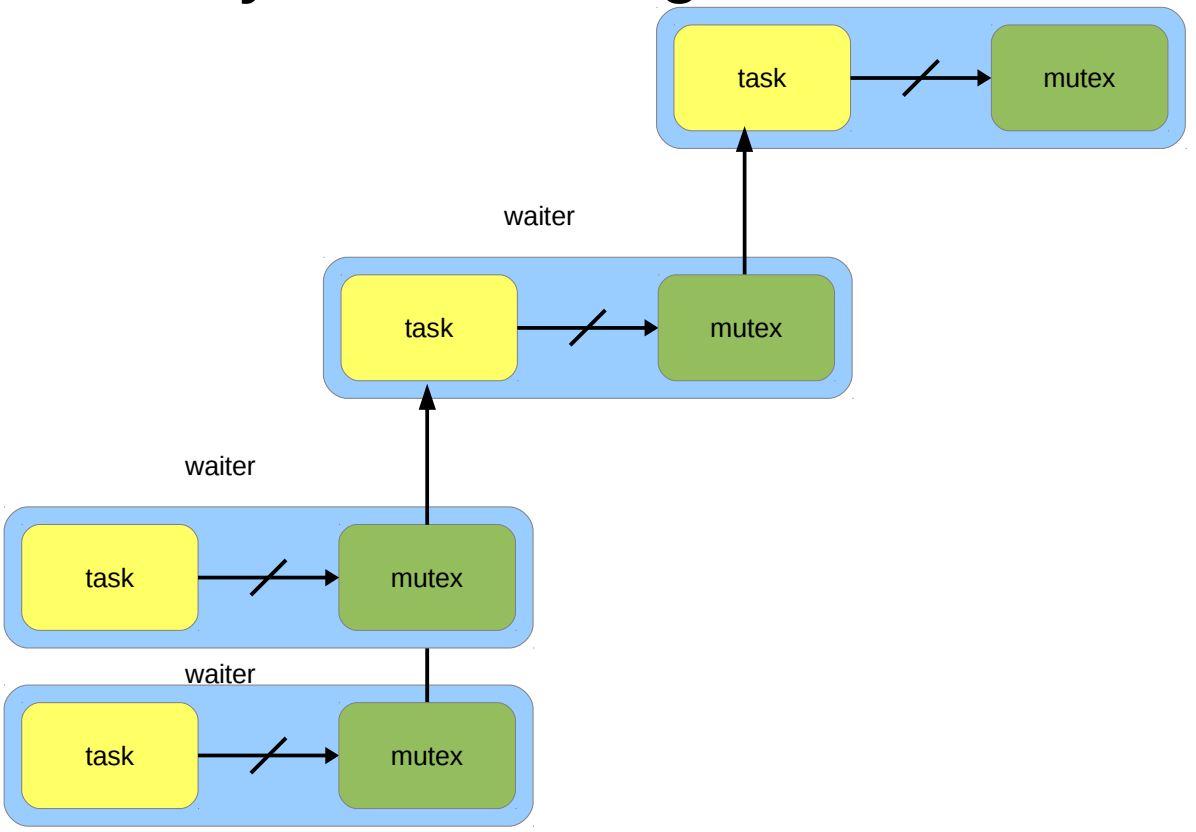
rt_mutexes: Enough knowledge to be dangerous

- Largely from (and apologies for hacking-up) documentation from rostedt
- rt_mutexes designed in -rt (duh), upstreamed
- Implement PI to solve PI (acronyms rock)
- rt_mutex has task who owns, tasks that block
  - (and the locks the blocked tasks own)



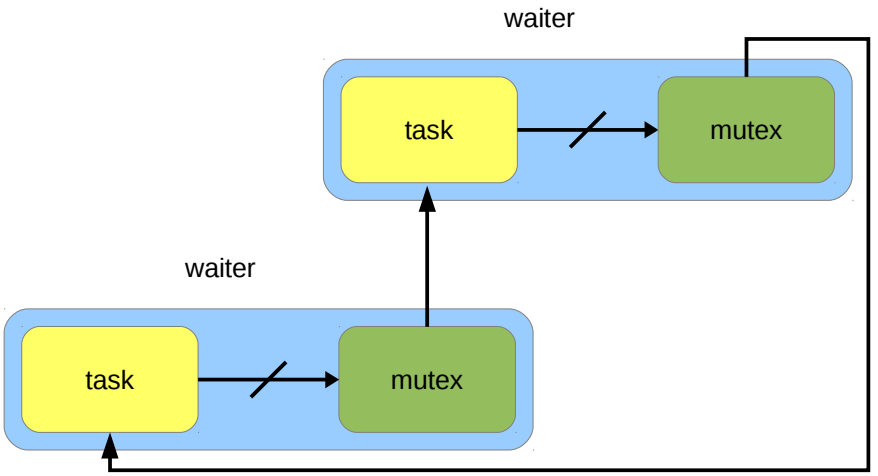
rt_mutexes: Enough knowledge to be dangerous

- These relationships allow for PI
  - Also handy for checking for deadlocks



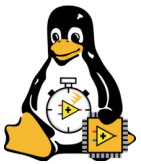
rt_mutexes: Enough knowledge to be dangerous

- These relationships allow for PI
  - Also handy for checking for deadlocks



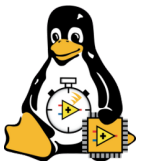
How to debug, and where?

- 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



kgdb: When printk's don't cut the mustard

- Configure the kernel
  - CONFIG_DEBUG_INFO
  - CONFIG_KGDB
  - CONFIG_KGDB_*method_to_connect*
  - CONFIG_KGDB_KDB (optional)



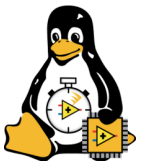
Connecting to a kgdb machine

- 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/

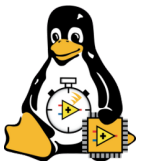
Quick Test

- Connect via gdb host machine
  - Connected to the target being tested
  - Debug vmlinux bin + source
  - gdb that understands the arch of the test machine
- Write a 'g' in /proc/sysrq-trigger to break
- Quick demo



Tips for using kgdb/gdb

- Search for (or write) useful user-defined cmds
  - Sequences you use frequently
- Pop cmds and settings in your ~/.gdbinit
- Graphical frontends are available if you must
- Excellent resources online

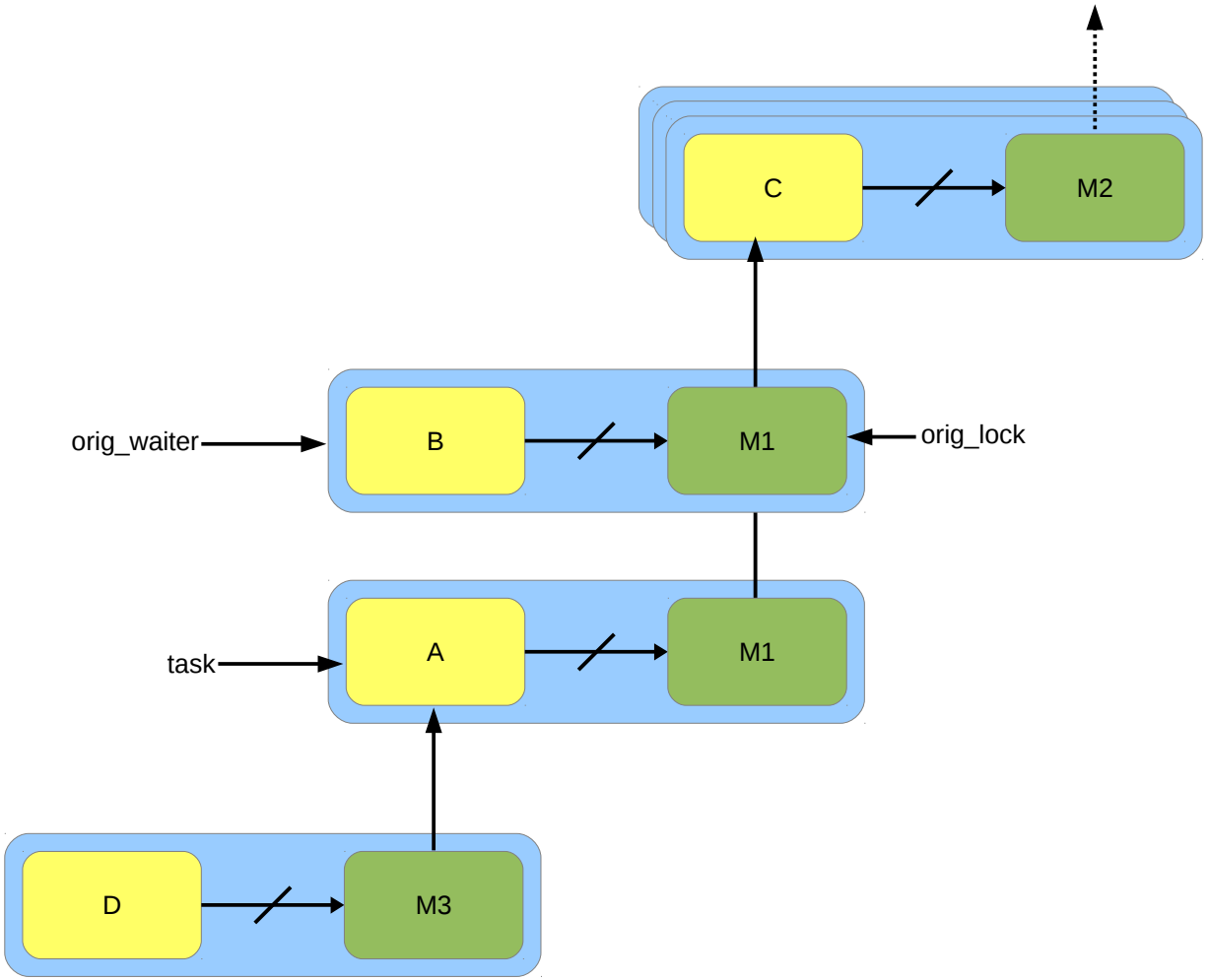


kgdb leads to a dead end

… 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 somehow were supposed to loop back around to the original mutex, as that's the current state of the structures and pointers within the function

State of the Priority Chain at EDEADLK

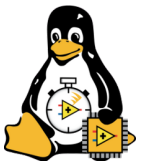


A few clues

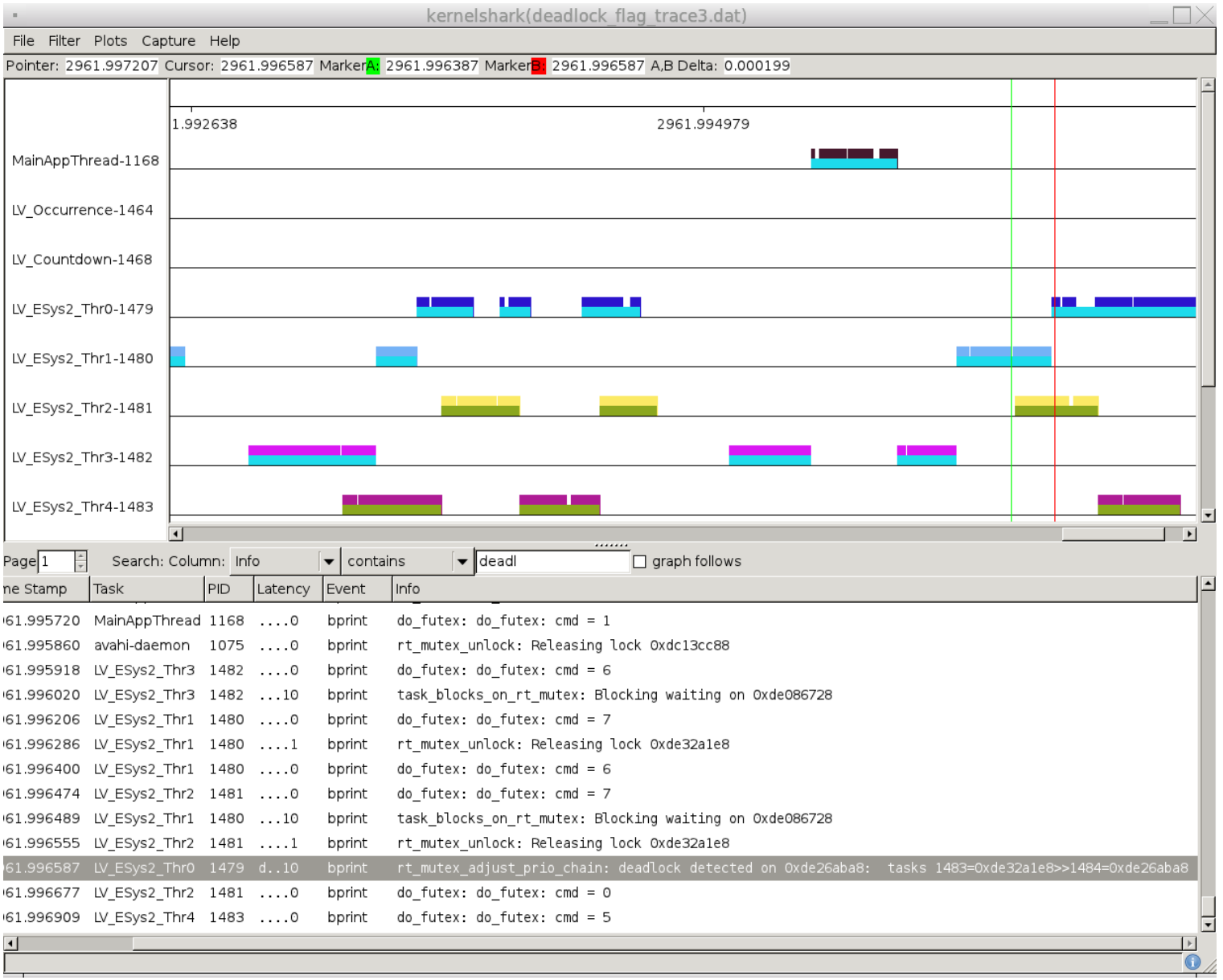
- Mutex M2 recently had an owner but doesn't currently
- There are two tasks (A, B) blocked on mutex M1
- The checks that occur while walking the chain don't see anything odd and complain until a deadlock is detected

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 (used for other purposes)
- Insert some strategic trace_printk()s
- Add SIGABRT handler to app to stop tracing
- Reproduce the issue, use trace-cmd extract

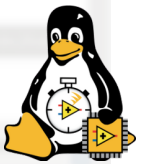


kernelshark comes into the picture

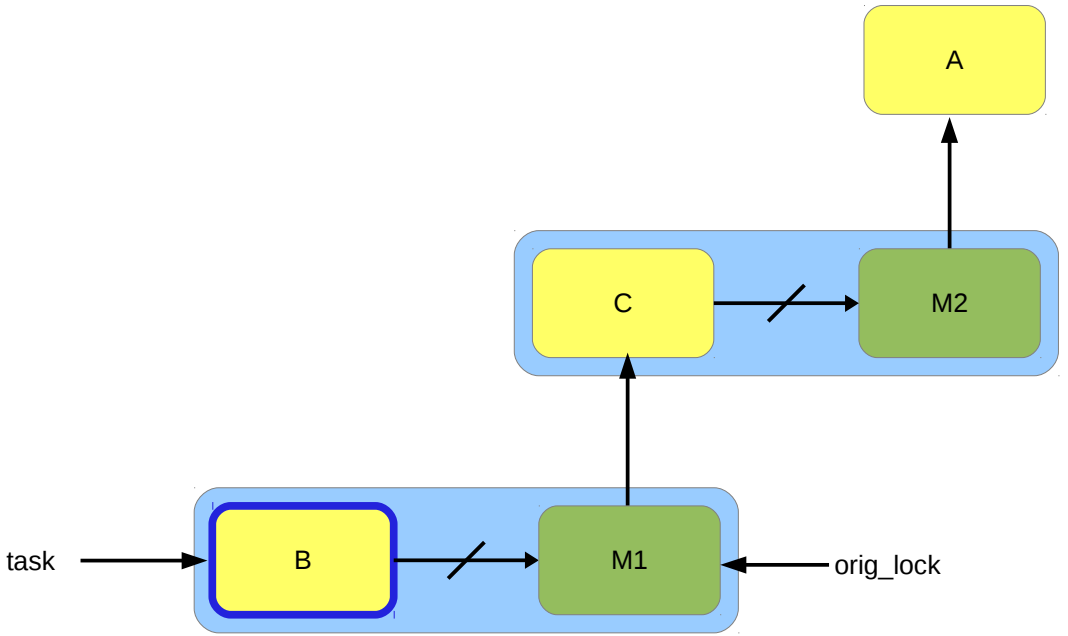


kernelshark comes into the picture

- 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 begin to form a narrative on what is happening

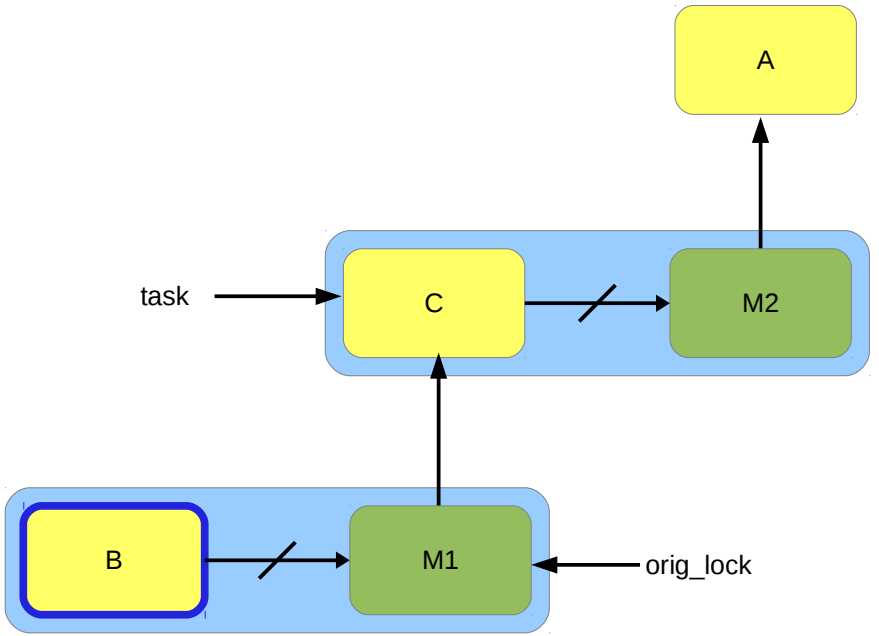


Re-ftrace-ing my steps



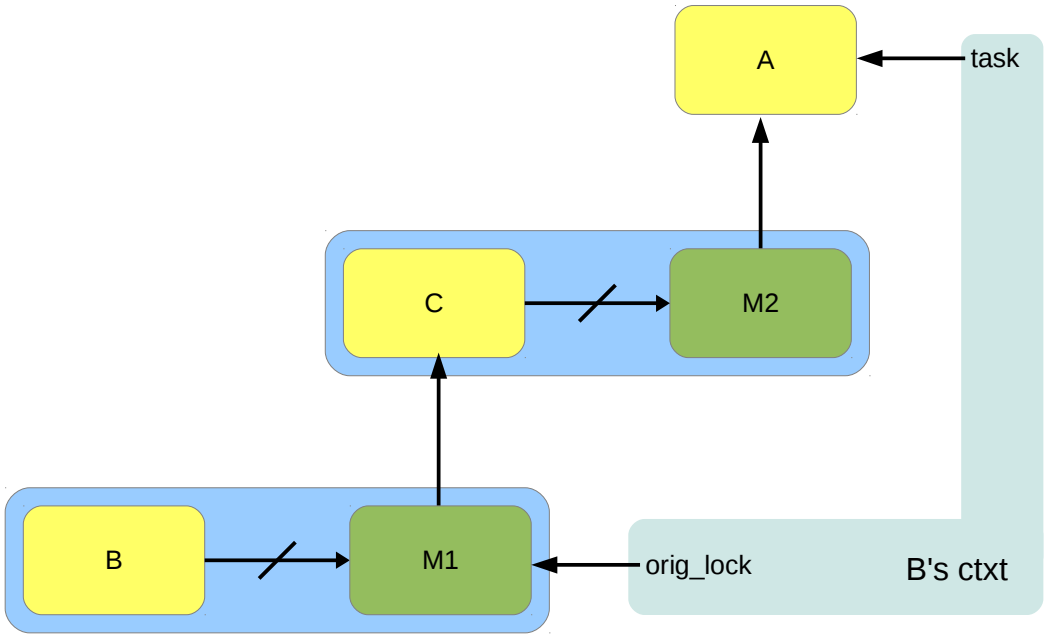
B blocks on M1
M1 is held by C
C is blocked on M2
M2 is held by A

Re-ftrace-ing my steps



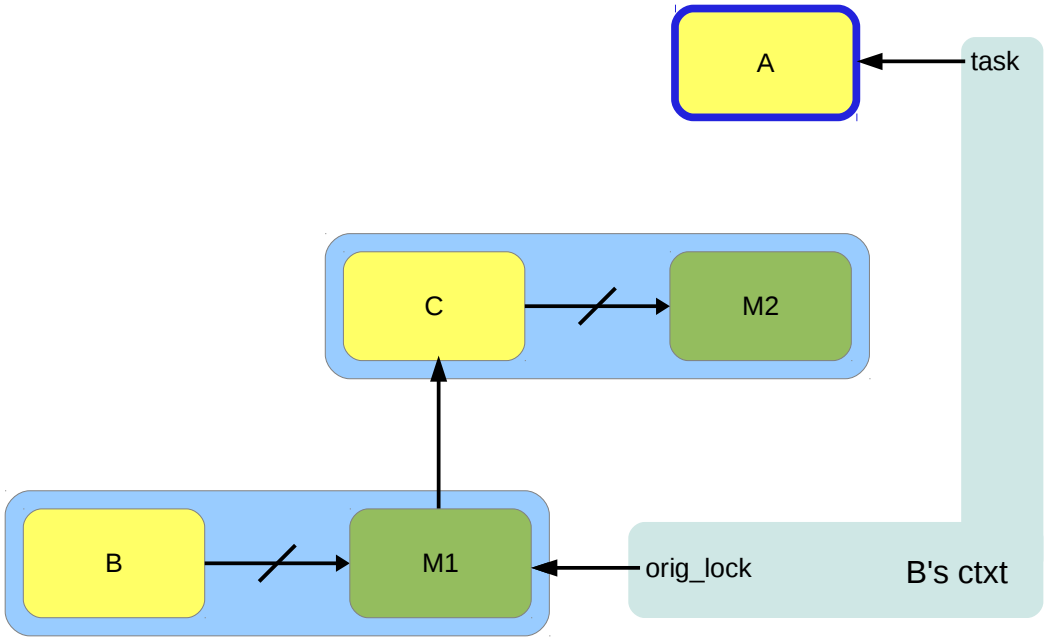
B blocks on M1
M1 is held by C
C is blocked on M2
M2 is held by A
B begins walking the prio chain

Re-ftrace-ing my steps



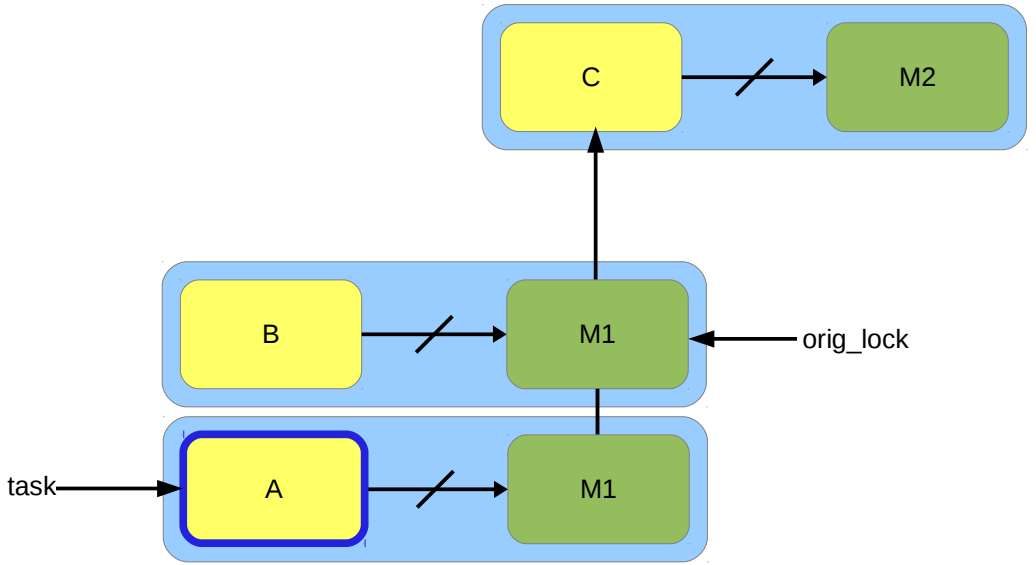
B blocks on M1
M1 is held by C
C is blocked on M2
M2 is held by A
B begins walking the prio chain...PREEMPT!

Re-ftrace-ing my steps



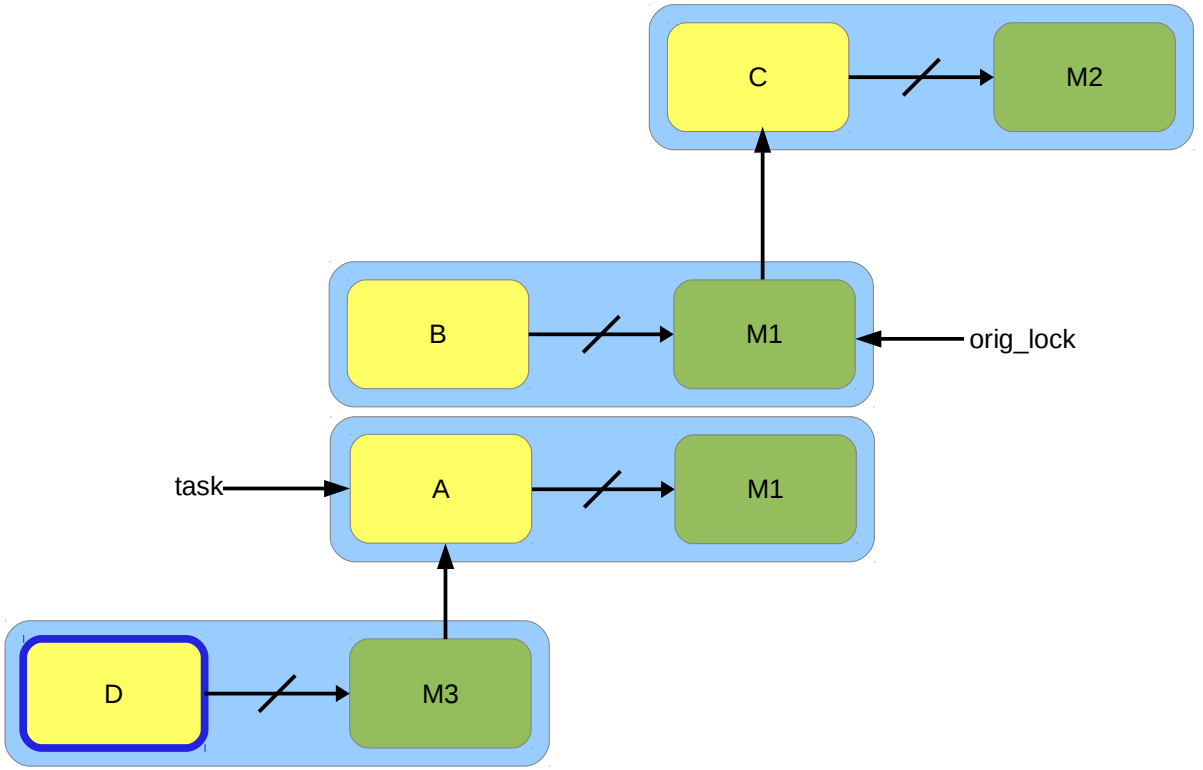
A is scheduled in, releases M2

Re-ftrace-ing my steps



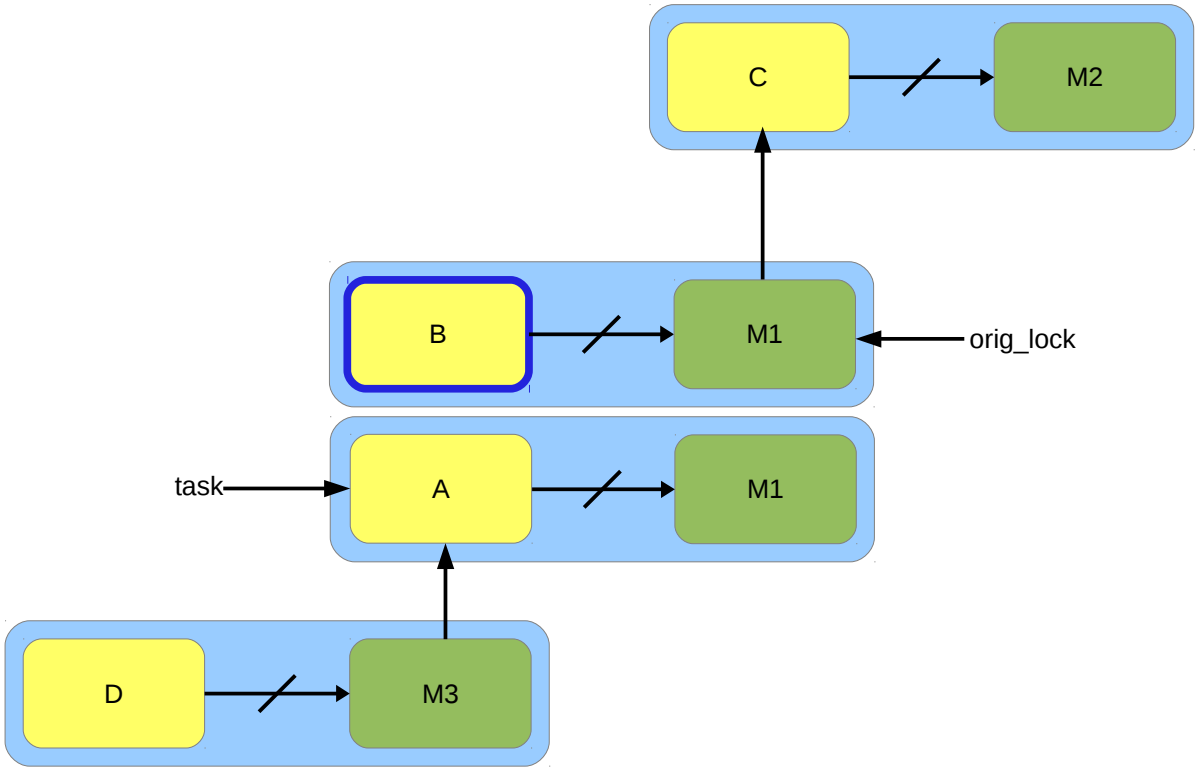
A is scheduled in, releases M2
A takes (uncontended) M3 in userspace
A blocks on M1

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

Putting it all together

- The situation we find the prio chain in just so happens to pass all of the checks put in place to verify that we still have a sane prio chain and that the chain hasn't changed to the point where we stop
- The mutex currently investigated is the same as the original mutex that blocked B (M1), this is seen as a deadlock and reported thusly

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 his own fix
- Moral: issue got fixed, learned about working with the mailing lists

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 debugging
  - kdump kernel(s)
  - KDB
  - Vendor tools

Questions?
Comments?
Thanks!