Using Serial kdb / kgdb to Debug the Linux Kernel

Doug Anderson, Google

Intro

About Me

- Random kernel Engineer at Google working on Chrome OS.
- I like debugging.
- I like debuggers.
- Not the author nor maintainer of kdb / kgdb, but I fix bugs sometimes.
- I really only have deep experience with Linux on arm32 / arm64.

About You 💖💖💖

- You're a kernel Engineer.
- You sometimes run into crashes / hangs / bugs on devices you're working on.
- You have a serial connection to the device you're working on.
  - There are other ways to talk to kdb / kgdb, but I won't cover those.
- You're here in person (or watching a video), since much of this will be demo.
- ~~You like to go for long romantic walks through the woods at night.~~

Syllabus

- What is kdb / kgdb?
- What kdb / kgdb are best suited for
- Comparison to similar tools
- Getting setup
- Debugging your first problem
- Debugging your second problem
- Next steps

What is kdb / kgdb?

- The docs are the authority. https://www.kernel.org/doc/html/v5.2/dev-tools/kgdb.html
- kdb = The Kernel DeBugger. A simple shell that can do simple peeks/pokes but also has commands that can print kernel state at time of crash.
- kgdb = The Kernel GDB server. Allows a second computer to run GDB and debug the kernel.

Do I want kdb, or kgdb?

- Before my time, I believe you had to pick. Now, you can have both.
- kgdb just lets you use vanilla gdb to debug the kernel. Awesome, but knows nothing about Linux(*).
- kdb knows about Linux but is not a source level (or even assembly level) debugger.
- You can enable kgdb without kdb, but why would you? kdb makes a nice first-level triage and can help with Linux-specifics.

(*) Well, there is "scripts/gdb/linux" to help...

What can I do at crash time with kdb?

- List all processes
- Dump dmesg
- Dump ftrace buffer
- Backtrace any process
- Send magic sysrq
- Peek/poke memory (I've never used this)

Mostly I just run "dumpall" and save it to a text file, then move over to kgdb.

What is kgdb good at?

- You need to have stashed away matching symbol files (vmlinux + modules)
- It's as good at debugging code as gdb is
  - When dealing with optimized code, that sometimes means "not very"
- It is slow, but usable
- You can debug any process in the system, though can't always backtrace past assembly code (which might include interrupts)
- It is far better suited for after-the-crash debugging than single step debugging
  - All CPUs stop and all interrupts are disabled while in the debugger. Not everything handles that so well.
  - Anything that involves periodic comms with the debugger (watchpoints?) is slooooow
  - Stepping / setting / clearing breakpoints just seems buggy

kdb/kgdb vs. JTAG

- Much overlap, especially when you point gdb at your JTAG
- JTAG needs dedicated pins and might be tricky to setup
- JTAG software often needs to be updated for each new core type
- JTAG software / hardware is often expensive
- There is no "kdb" over JTAG
- JTAG communication is usually faster, sometimes has extra buffers

tl;dr: kdb / kgdb can cover ~75% of what people use JTAG for and is free / doesn't require a special setup.

kdb/kgdb vs. reading the kcrash

- Why bother with kgdb when everything you could need is printed to the console (or pstore) on panic?
- Panic prints a lot, but not everything. Maybe you need to see the value of a global variable, or dereference a few pointers.
- Having gdb able to help you make sense of a crash is invaluable.

kdb/kgdb vs. kdump

- In theory you can set things up to dump tons of stuff about the kernel at crash time by kexec-ing a dump kernel.
- I've never done this, so maybe someone will do a presentation next year on it.

Setting Up

Getting setup - need a serial port

- I said this in the beginning. Weren't you listening?
  - I probably distracted you with the long romantic walks through the woods at night
- Serial driver needs polling support since we run with interrupts off.
  - Not too hard to add. poll_get_char() / poll_put_char()

Getting setup - kdmx

- Technically not needed.
- Usually run with kernel console + agetty on serial port and want kgdb to share too. Constantly closing / opening the serial port is a pain.
- kdmx creates two pseudo terminals: one for console+agetty, other for gdb.
- kdmx is more reliable than agent-proxy (a similar tool hosted on the same git server) and doesn't get your IT folks riled up.
- Known issue: every once in a while kdmx gets confused and keeps echoing "-". If that happens, just restart it.

Hosted at https://git.kernel.org/pub/scm/utils/kernel/kgdb/agent-proxy.git

kdmx is not too hard to set up

```
$ mkdir -p /tmp/kdmx_is_not_hard
$ cd /tmp/kdmx_is_not_hard
$ git clone git://git.kernel.org/pub/scm/utils/kernel/kgdb/agent-proxy.git .
$ cd kdmx
$ make
$ ./kdmx -n -p "/dev/ttyUSB0" -s /tmp/kdmx_ports &
$ cu --nostop -l $(cat /tmp/kdmx_ports_trm)

# When debugging
$ ${CROSS_ARCH}-gdb /path/to/vmlinux \
        -ex "target remote $(cat /tmp/kdmx_ports_gdb)"
```

(could use something besides "cu" if you want)

Getting setup - gdb

- You'll need a cross-compiled version of GDB.
- AKA: if your host is x86_64 and your target is aarch64 then you need gdb that can run in x86_64 but can debug an aarch64 target.
- Presumably comes from the same place your compiler comes from.

gdb is ~~not~~ too hard to set up

- Setting up gdb is way beyond the scope of this talk.
- If you don't have gdb that works, seek professional help.
- If you actually know how to set up gdb yourself, seek professional help.

Getting setup - kernel config

```
$ cat <<EOF >> .config
CONFIG_VT=y
CONFIG_VT_CONSOLE=y
CONFIG_KGDB=y
CONFIG_KGDB_KDB=y
CONFIG_PANIC_TIMEOUT=0
CONFIG_RANDOMIZE_BASE=n
CONFIG_WATCHDOG=n
CONFIG_MAGIC_SYSRQ_DEFAULT_ENABLE=1
EOF
```

```
$ cat <<EOF >> .config
CONFIG_DEBUG_KERNEL=y
CONFIG_DEBUG_INFO=y
CONFIG_DEBUG_INFO_DWARF4=y
CONFIG_FRAME_POINTER=y
CONFIG_GDB_SCRIPTS=y
EOF
```

Getting setup - command line params

- Imagine your serial port is ttyS2, then you need on your kernel command line:
    - kgdboc=ttyS2
- For good measure:
    - console=ttyS2,115200n8 oops=panic panic=0 kgdboc=ttyS2

The "oc" in kgdboc is supposed to be "over console". You can actually get kgdb to run over a port even if it's not the console port, though.

Demo

Dropping into the debugger

- Magic sysrq is the nicest way, but not always simple:
    - External keyboard: Alt-PrintScr-G
    - Command line shell: 'echo g > /proc/sysrq-trigger'
    - Send "BREAK-G" over serial port, but:
        - Break is hard to send over pseudo-terminals. kdmx allows ~B, but might be eaten up by the next level (on a Chromebook, servod eats but provides its own escape sequence)
        - Relies on userspace having an agetty running because otherwise nobody is listening
- Hardcode a breakpoint into your code: kgdb_breakpoint()
- Cause an oops / panic
- Make your own debug trigger by adding kgdb_breakpoint() into an IRQ handler

Debugging your first problem

```
localhost ~ # echo WRITE_KERN > /sys/kernel/debug/provoke-crash/DIRECT
[   35.634506] lkdtm: Performing direct entry WRITE_KERN
[   35.640172] lkdtm: attempting bad 18446744073709551584 byte write at ffffff80105657b8
[   35.648943] Unable to handle kernel write to read-only memory at virtual address ...
...
Entering kdb (current=0xffffffc0de55f040, pid 1470) on processor 4 Oops: (null)
due to oops @ 0xffffff80108bfa48
CPU: 4 PID: 1470 Comm: bash Not tainted 5.3.0-rc2+ #13
Hardware name: Google Kevin (DT)
pstate: 00000005 (nzcv daif -PAN -UAO)
pc : __memcpy+0x48/0x180
lr : lkdtm_WRITE_KERN+0x4c/0x90
...

[4]kdb>
```

Demo: 'bt'

```
[4]kdb> bt
Stack traceback for pid 1470
0xffffffc0de55f040     1470      721  1    4   R  0xffffffc0de55fa30 *bash
Call trace:
 dump_backtrace+0x0/0x138
 show_stack+0x20/0x2c
 kdb_show_stack+0x60/0x84
 ...
 do_mem_abort+0x4c/0xb4
 el1_da+0x20/0x94
 __memcpy+0x48/0x180
 lkdtm_do_action+0x24/0x44
 direct_entry+0x130/0x178
 ...

[4]kdb>
```

Demo: 'dumpall'

```
[4]kdb> dumpall
[dumpall]kdb>   pid R

KDB current process is bash(pid=1470)
[dumpall]kdb>   -dumpcommon

[dumpcommon]kdb>   set BTAPROMPT 0

[dumpcommon]kdb>   set LINES 10000

[dumpcommon]kdb>   -summary

sysname    Linux
release    5.3.0-rc2+
version    #13 SMP PREEMPT Mon Jul 29 14:52:19 PDT 2019
machine    aarch64
nodename   localhost
domainname (none)
date       2019-07-29 21:54:10 tz_minuteswest 0
uptime     00:05
load avg   1.08 0.33 0.11

MemTotal:        3963548 kB
MemFree:         3552620 kB
Buffers:           10788 kB
[dumpcommon]kdb>   -cpu

Currently on cpu 4
Available cpus: 0(I), 1, 2-3(I), 4, 5(I)
[dumpcommon]kdb>   -ps

4 idle processes (state I) and
49 sleeping system daemon (state M) processes suppressed,
use 'ps A' to see all.
Task Addr               Pid   Parent [*] cpu State Thread             Command
0xffffffc0ea4e3040      364        2  1    1   R  0xffffffc0ea4e3a30  loop0
0xffffffc0de55f040     1470      721  1    4   R  0xffffffc0de55fa30 *bash
```

Demo: 'dumpall' (for real)

- So much stuff it can't possibly fit on a slide.
- Some random status that I rarely look at.
- Outputs the end of dmesg (you can get more if you want).
- Lists all processes in a clean-ish format.
- Dumps stacks for all processes, which can be quite useful.

Demo: sr (run sysrq)

```
[4]kdb> sr m
sysrq: Show Memory
Mem-Info:
active_anon:3312 inactive_anon:103 isolated_anon:0
 active_file:6808 inactive_file:36140 isolated_file:0
 unevictable:15000 dirty:2522 writeback:5587 unstable:0
 ...
58097 total pagecache pages
0 pages in swap cache
Swap cache stats: add 0, delete 0, find 0/0
Free swap  = 0kB
Total swap = 0kB
1015040 pages RAM
0 pages HighMem/MovableOnly
24153 pages reserved
4096 pages cma reserved
```

Demo: kgdb

- Can (but usually don't need to) enter kgdb from kdb using "kgdb" command.
- You'll point gdb at the pseudo-tty opened by kdmx.
- Remember you need to have kept your symbols around.

Demo: kgdb attaching

```
$ aarch64-cros-linux-gnu-gdb \
  /build/${BOARD}/usr/lib/debug/boot/vmlinux \
  -ex "target remote $(cat /tmp/kdmx_ports_gdb)"
...
...
Reading symbols from /build/kevin/usr/lib/debug/boot/vmlinux...done.
Remote debugging using /dev/pts/89
memcpy () at .../arch/arm64/lib/copy_template.S:94
94              stp1    A_l, A_h, dst, #16

(gdb)
```

Demo: kgdb 'bt'

```
(gdb) bt
#0  memcpy () at .../arch/arm64/lib/copy_template.S:94
#1  0xffffff801056584c in lkdtm_WRITE_KERN ()
    at .../drivers/misc/lkdtm/perms.c:116
#2  0xffffff8010564d14 in lkdtm_do_action (crashtype=0xffffff8010a8aa00 <crashtypes+608>)
    at .../drivers/misc/lkdtm/core.c:221
#3  0xffffff8010564c90 in direct_entry (f=<optimized out>, user_buf=<optimized out>,
                                        count=11, off=0xffffff8011d9bdf0)
    at .../drivers/misc/lkdtm/core.c:382
...
#15 0xffffff80100830f8 in el0_sync () at .../arch/arm64/kernel/entry.S:779
Backtrace stopped: Cannot access memory at address 0xffffff8011de40d8
```

(Backtrace stopped message is normal)

Demo: kgdb 'disass /s'

```
(gdb) disass /s
Dump of assembler code for function memcpy:
.../arch/arm64/lib/copy_template.S:
42              mov     dst, dstin
   0xffffff80108bfb40 <+0>:     mov     x6, x0
...
94              stp1    A_l, A_h, dst, #16
=> 0xffffff80108bfbb4 <+116>:   stp     x7, x8, [x6], #16


(gdb) print /x $x6
$1 = 0xffffff8010565890


(gdb) info symbol $x6
do_overwritten in section .text
```

Demo: need to know assembly???

- Can get by without knowing assembly, but helps to not be too afraid of it since you end up there sometimes.
- Good to know basics, like "`stp x7, x8, [x6], #16`" writes registers x7/x8 to (roughly) the memory location pointed to by x6. Can always search the web!
- Sometimes assembly can help you figure out the value of a variable when gdb claims "<optimized out>".

Demo: kgdb 'info reg'

```
(gdb) info reg
x0             0xffffff8010565890       -549481719664
x1             0xffffff80105658c0       -549481719616
x2             0xffffffffffffffe0       -32
x3             0x20     32
x4             0x0      0
x5             0x0      0
x6             0xffffff8010565890       -549481719664
...
sp             0xffffff8011d9bc40       0xffffff8011d9bc40
pc             0xffffff80108bfbb4       0xffffff80108bfbb4 <memcpy+116>
cpsr           0x60000005       [ SP EL=2 C Z ]
fpsr           0x0      0
fpcr           0x0      0
```

Demo: kgdb back to C (1)

```
(gdb) frame 1
#1  0xffffff801056584c in lkdtm_WRITE_KERN () at .../drivers/misc/lkdtm/perms.c:116
116                memcpy(ptr, (unsigned char *)do_nothing, size);

(gdb) list
111
112                size = (unsigned long)do_overwritten - (unsigned long)do_nothing;
113                ptr = (unsigned char *)do_overwritten;
114
115                pr_info("attempting bad %zu byte write at %px\n", size, ptr);
116                memcpy(ptr, (unsigned char *)do_nothing, size);
117                flush_icache_range((unsigned long)ptr, (unsigned long)(ptr + size));
118
119                do_overwritten();
120        }
```

Demo: kgdb back to C (2)

```
(gdb) print ptr
$2 = (unsigned char *) 0xffffff8010565890 <do_overwritten> "\375{\277\251\375\003"

(gdb) print size
$3 = 18446744073709551584

(gdb) print do_overwritten - do_nothing
$4 = -32

(gdb) print (unsigned long)(do_overwritten - do_nothing)
$13 = 18446744073709551584
```

(I wonder if that huge number was intentional)

Demo: kgdb = pretty handy (1)

```
(gdb) frame 2
#2  0xffffff8010564d14 in lkdtm_do_action (crashtype=0xffffff8010a8aa00 <crashtypes+608>)
    at .../drivers/misc/lkdtm/core.c:221
221             crashtype->func();

(gdb) print crashtype
$5 = (const struct crashtype *) 0xffffff8010a8aa00 <crashtypes+608>

(gdb) print *crashtype
$6 = {name = 0xffffff8010bfe2d9 "WRITE_KERN", func = 0xffffff8010565800 <lkdtm_WRITE_KERN>}
```

Demo: kgdb = pretty handy (2)

```
(gdb) frame 5
#5  0xffffff801026d288 in __vfs_write (file=0xffffffc0eacf3340, p=0x13ac588
      "WRITE_KERN\nrcolors\n", count=11, pos=0xffffff8011d9bdf0) at .../fs/read_write.c:494
494                     return file->f_op->write(file, p, count, pos);

(gdb) print *file
$7 = {f_u = {fu_llist = {next = 0x0}, fu_rcuhead = {next = 0x0, func = 0x0}}, f_path = {
    mnt = 0xffffffc0f104fce0, dentry = 0xffffffc0ef13b1a0},
  f_inode = 0xffffffc0ef13c008, f_op = 0xffffffc0dd407c80, f_lock = {{rlock = {
        raw_lock = {{val = {counter = 0}, {locked = 0 '\000', pending = 0 '\000'}, {
              locked_pending = 0, tail = 0}}}, magic = 3735899821,
        owner_cpu = 4294967295, owner = 0xffffffffffffffff}}},
  f_write_hint = WRITE_LIFE_NOT_SET, f_count = {counter = 1}, f_flags = 131073,
  ...
```

Demo: kgdb = pretty handy (3)

```
(gdb) set print pretty on

(gdb) set pagination off

(gdb) print *file
$8 = {
  f_u = {
    fu_llist = {
      next = 0x0
    },
    fu_rcuhead = {
      next = 0x0,
      func = 0x0
    }
  },
  ...
```

Demo: kdb commands through kgdb

```
(gdb) monitor lsmod
Module                 Size  modstruct     Used by
btusb                 40960  0xffffff8008bdb140    0  (Live) 0xffffff8008bd3000 [ ]
btrtl                 16384  0xffffff8008b3f040    1  (Live) 0xffffff8008b3d000 [ btrtl ]
btbcm                 16384  0xffffff8008bc4040    1  (Live) 0xffffff8008bc2000 [ btbcm ]
btintel               20480  0xffffff8008b0f140    1  (Live) 0xffffff8008b0c000 [ btintel ]
...

(gdb) monitor dumpall
[dumpall]kdb>   pid R

KDB current process is swapper/0(pid=0)
[dumpall]kdb>   -dumpcommon

...
```

Demo: lx- scripts (1)

- There are python scripts that work with gdb to parse / interpret kernel global data structures.
- Bundled with kernel sources: "scripts/gdb". Tied to kernel version (since globals / structures could change over time).
- Put "vmlinux-gdb.py" and "scripts" next to your vmlinux.
- Add "add-auto-load-safe-path" to "~/.gdbinit"

Demo: lx- scripts (2)

```
$ cd /build/kevin/usr/lib/debug/boot

$ find .
.
./scripts
./scripts/gdb
./scripts/gdb/linux
...
./scripts/gdb/linux/__init__.py
./vmlinux
./vmlinux-gdb.py

$ cat ~/.gdbinit
add-auto-load-safe-path /build/kevin/usr/lib/debug/boot/
```

Demo: lx- scripts (3)

```
lx-clk-summary
lx-cmdline
lx-configdump
lx-cpus
lx-device-list-bus
lx-device-list-class
lx-device-list-tree
lx-dmesg
lx-fdtdump
lx-genpd-summary
lx-iomem
lx-ioports
lx-list-check
lx-lsmod
lx-mounts
lx-ps
lx-symbols
lx-timerlist
lx-version
```

```
(gdb) lx-clk-summary
                                 enable  prepare  protect
   clock                          count    count    count        rate
------------------------------------------------------------------------
 xin32k                               0        0        0       32768
 xin24m                              20       21        0    24000000
    clk_timer11                       0        0        0    24000000
```

Demo: Debugging a 2nd crash (1)

```
# echo SOFTLOCKUP > /sys/kernel/debug/provoke-crash/DIRECT
[   45.069040] lkdtm: Performing direct entry SOFTLOCKUP
<BREAK>g
[   46.921886] sysrq: DEBUG

Entering kdb (current=0xffffff801101a9c0, pid 0) on processor 0 due to Keyboard Entry
[0]kdb>
```

Can we find the processes what is locked up? Yes (assuming you have a kdb where "btc" works -- https://lore.kernel.org/patchwork/patch/1108504/)

Demo: Debugging a 2nd crash (2)

- If we truly don't know why something is stuck, can just do "dumpall" and look through all the stacks.
- If you think something is running, try "btc".
- Can also try "ps <state>". See kdb_task_state_string() for <state>.
- Can also try "sr w" to show blocked tasks.
- If you have a PID, you can use "btp" to backtrace a PID.
- In this case, "btc" works.

Demo: Debugging a 2nd crash (3)

```
[0]kdb> btc
...
Stack traceback for pid 1478
0xffffffc0e0acb040     1478      728  1    1   R  0xffffffc0e0acba30  bash
Call trace:
 lkdtm_SOFTLOCKUP+0x1c/0x24
 lkdtm_do_action+0x24/0x44
 direct_entry+0x130/0x178
 full_proxy_write+0x60/0xb4
 __vfs_write+0x54/0x18c
 vfs_write+0xcc/0x174
 ksys_write+0x7c/0xe4
 __arm64_sys_write+0x20/0x2c
 el0_svc_common+0x9c/0x14c
 el0_svc_compat_handler+0x28/0x34
 el0_svc_compat+0x8/0x10
```

Demo: "info thread" in kgdb (1)

- Tasks in Linux are represented as "threads" in kgdb.
- You can see a list of the mapping with "info thread".
- Can be used to point gdb at other tasks, either running or sleeping.

Demo: "info thread" in kgdb (2)

```
(gdb) set pagination off

(gdb) info thread
  Id   Target Id         Frame
* 1    Thread 4294967294 (shadowCPU0) arch_kgdb_breakpoint () at
                                      .../arch/arm64/include/asm/kgdb.h:21
  ...
  169  Thread 1478 (bash) cpu_relax () at .../arch/arm64/include/asm/processor.h:248

(gdb) thread 169
[Switching to thread 169 (Thread 1478)]
#0  cpu_relax () at .../arch/arm64/include/asm/processor.h:248
248             asm volatile("yield" ::: "memory");
```

Demo: kgdb falls on its face (1)

- kgdb (on arm64) can't trace past an exception handler because they're not tagged properly.
- Try the above (soft lockup) without manually breaking into the debugger--let the soft lockup handler detect it.
- Compare kdb (kernel back trace) with kgdb's backtrace.

Demo: kgdb falls on its face (2)

```
(gdb) bt
#0  arch_kgdb_breakpoint () at .../v4.19/arch/arm64/include/asm/kgdb.h:21
#1  kgdb_breakpoint () at .../v4.19/kernel/debug/debug_core.c:1135
...
#17 0xffffff8010081164 in handle_domain_irq (domain=0x1, hwirq=<optimized out>,
regs=0xffffff80140ebb20)
    at .../v4.19/include/linux/irqdesc.h:174
#18 gic_handle_irq (regs=0xffffff80140ebb20) at .../v4.19/drivers/irqchip/irq-gic-v3.c:511
#19 0xffffff8010082cb8 in el1_irq () at .../v4.19/arch/arm64/kernel/entry.S:670
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
```

Probably could be fixed with the proper CFI annotations. Patches welcome!

Demo: Breakpoints (1)

- In general kgdb is better for debugging crashes, but breakpoints do still work and you can still continue after you drop into the debugger.
- When I tested, I was sometimes unable to delete breakpoints (?).

Demo: Breakpoints (2)

```
# echo g > /proc/sysrq-trigger
```

```
(gdb) br pci_try_reset_function
Breakpoint 1 at 0xffffff801044557c: file .../drivers/pci/pci.c, line 5003.
(gdb) c
Continuing.
```

```
# echo 1 > /sys/kernel/debug/mwifiex/mlan0/reset
```

```
Thread 188 hit Breakpoint 1, pci_try_reset_function (dev=0xffffffc0ef4b2880)
    at .../drivers/pci/pci.c:5003
5003    {
(gdb)
```

Demo: modules - the manual way (1)

```
(gdb) bt
#0  pci_try_reset_function (dev=0xffffffc0ef4b2880) at .../drivers/pci/pci.c:5003
#1  0xffffff8008af6674 in ?? ()
#2  0x00000000000001c8 in ?? ()

(gdb) monitor lsmod
Module                Size  modstruct     Used by
...
mwifiex_pcie         32768  0xffffff8008afc340    0  (Live) 0xffffff8008af6000 [ ]
mwifiex             245760  0xffffff8008aecc80    1  (Live) 0xffffff8008ab9000 [ mwifiex ]
cfg80211            598016  0xffffff8008aa8dc0    1  (Live) 0xffffff8008a26000 [ cfg80211 ]
```

Demo: modules - the manual way (2)

```
(gdb) add-symbol-file .../wireless/marvell/mwifiex/mwifiex.ko.debug 0xffffff8008ab9000
(gdb) add-symbol-file .../wireless/marvell/mwifiex/mwifiex_pcie.ko.debug 0xffffff8008af6000

(gdb) bt
#0  pci_try_reset_function (dev=0xffffffc0ef4b2880) at .../drivers/pci/pci.c:50
#1  0xffffff8008af6674 in mwifiex_pcie_card_reset_work (adapter=<optimized out>)
    at .../drivers/net/wireless/marvell/mwifiex/pcie.c:2807
#2  mwifiex_pcie_work (work=<optimized out>)
    at .../drivers/net/wireless/marvell/mwifiex/pcie.c:2820
#3  0xffffff8010101b08 in process_one_work (worker=0xffffffc0ec2ded80,
                                            work=0xffffffc0e131cce8)
    at .../kernel/workqueue.c:2269
#4  0xffffff8010102038 in worker_thread (__worker=0xffffffc0ec2ded80)
    at .../kernel/workqueue.c:2415
#5  0xffffff8010106bd8 in kthread (_create=0xffffffc0da167780) at ...
#6  0xffffff80100856ac in ret_from_fork () at .../arch/arm64/kernel/entry.S:116
```

Demo: modules - lx-symbols

```
(gdb) lx-symbols /build/kevin/usr/lib/debug
loading vmlinux
scanning for modules in /build/kevin/usr/lib/debug
scanning for modules in /outside/home/dianders/b/tip/src/third_party/kernel/v4.19
...
loading @0xffffff8008af6000: .../marvell/mwifiex/mwifiex_pcie.ko.debug
loading @0xffffff8008ab9000: .../marvell/mwifiex/mwifiex.ko.debug
```

NOTE: having this work with Chrome OS split debug (.ko.debug) requires a patch for now.

Demo: can't stop unstoppable cpus (1)

- On most architectures (like arm64), kgdb stops CPUs by sending them an IPI.
- If a CPU is looping with interrupts disabled then you're out of luck.
- Maybe in the future more architectures will solve this (FIQ on arm64?)

Demo: can't stop unstoppable cpus (2)

```
# echo HARDLOCKUP > /sys/kernel/debug/provoke-crash/DIRECT
[   43.981017] lkdtm: Performing direct entry HARDLOCKUP
<BREAK>g
[   45.672377] sysrq: DEBUG
[   46.698158] KGDB: Timed out waiting for secondary CPUs.

Entering kdb (current=0xffffff801101a9c0, pid 0) on processor 0 due to Keyboard Entry
[0]kdb> btc
btc: cpu status: Currently on cpu 0
Available cpus: 0, 1-3(I), 4(D), 5(I)
...
WARNING: no task for cpu 4
...
```

Demo: tricks for optimized code (1)

- Set breakpoint at cros_ec_xfer_high_pri() and stop
- See that in frame 5 (cros_ec_console_log_work()) param is "<optimized out>"

```
Thread 184 hit Breakpoint 1, cros_ec_xfer_high_pri (ec_dev=0xffffffc0f1088900,
    ec_msg=0xffffff8011cabd10, fn=0xffffff80106e59ec <do_cros_ec_pkt_xfer_spi>)
    at .../drivers/platform/chrome/cros_ec_spi.c:648
648     {
(gdb) bt
#0  cros_ec_xfer_high_pri (ec_dev=0xffffffc0f1088900, ec_msg=0xffffff8011cabd10,
...
#5  0xffffff80106e7d1c in cros_ec_console_log_work (__work=<optimized out>)
    at .../drivers/platform/chrome/cros_ec_debugfs.c:76
```

Demo: tricks for optimized code (2)

- Look elsewhere, like 1 frame up!

```
(gdb) frame 6
#6  0xffffff8010101a98 in process_one_work (worker=0xffffffc0d9bd0080,
    work=0xffffffc0f16de588)
    at /mnt/host/source/src/third_party/kernel/v4.19/kernel/workqueue.c:2269
2269            worker->current_func(work);

(gdb) print work
$1 = (struct work_struct *) 0xffffffc0f16de588
```

Demo: tricks for optimized code (3)

- Sometimes have to work harder

```
#5  0xffffff80106e7d1c in cros_ec_console_log_work (__work=<optimized out>)
    at .../drivers/platform/chrome/cros_ec_debugfs.c:76
76              ret = cros_ec_cmd_xfer_status(ec->ec_dev, &snapshot_msg);
(gdb) list cros_ec_console_log_work
...
57      static void cros_ec_console_log_work(struct work_struct *__work)
58      {
59              struct cros_ec_debugfs *debug_info =
60                      container_of(to_delayed_work(__work),
61                                   struct cros_ec_debugfs,
62                                   log_poll_work);
(gdb) print debug_info
$4 = <optimized out>
```

Demo: tricks for optimized code (4)

```
(gdb) print &((struct cros_ec_debugfs *)0)->log_poll_work->work
$6 = (struct work_struct *) 0x88

(gdb) frame 6
#6  0xffffff8010101a98 in process_one_work (worker=0xffffffc0d9bd0080,
    work=0xffffffc0f16de588)
    at /mnt/host/source/src/third_party/kernel/v4.19/kernel/workqueue.c:2269
2269            worker->current_func(work);

(gdb) print *(struct cros_ec_debugfs *)((u64)work - 0x88)
$7 = {ec = 0xffffffc0ee496080, dir = 0xffffffc0ef2308a8, log_buffer = {
    buf = 0xffffffc0ee578080 "[1517.597 AP wants warm reset]\r\nRTC: 0x5d44b563
(1564783971.00 s)\r\n[1517.597 chipset_reset(0)]\r\n[1517.607 ERR-GTH]\r\n[1518.760 event
set 0x08000000]\r\nC0 st2\r\nfusb302_tcpm_select_rp_value: 62 vs 61, 19 "..., head = 579,
```

Demo: tricks for optimized code (5)

- Sometimes might need to look at assembly
- ARM64 calling convention:
  - R0 - R7 are parameters
  - R0 - R18 aren't preserved across function calls
  - R19 - R28 are preserved, so you can rely upon them when debugging

Demo: tricks for optimized code (6)

- Recompile with less optimization
- Sometimes you can get by with a #pragma

Wrapping Up

Conclusion + next steps

- Running with kdb / kgdb enabled as you're developing can be a real timesaver.
- Not everything always works perfectly, but there's still a lot there.
- It's not as hard as you thought to get setup.