Kernel Debugging

Advanced Embedded Linux Development

with **Dan Walkes**

University of Colorado **Boulder**

Learning objectives:

Strategies for debug of kernel code and your driver

Strategies for using printk

Oops messages

Kernel Debugging Techniques

- Not easily executed in a debugger
  - Possible to analyze with gdb and /proc/kcore
  - Not possible to halt, set, breakpoints, modify memory
  - Not the typical preferred method to debug
- Not easily traced
- Often difficult to reproduce bugs, especially timing related
- Often bugs crash the system and destroy evidence.

How do we debug the kernel?

- Start by enabling “kernel hacking” options in your kernel menuconfig (not typically on due to performance)
- CONFIG_DEBUG_KERNEL and friends
- CONFIG_DEBUG_SPINLOCK - lock debugging
- CONFIG_DEBUG_INFO - needed for kgdb
- CONFIG_MAGIC_SYSRQ - will discuss later
- Your book has a more complete list
- Also check the “kernel hacking” section of the kernel menuconfig.

Kernel Debug Printing

- Printing is most common debug method
- printk
- Notice no comma
- EMERG, ALERT, CRIT, ERR, WARNING, NOTICE, INFO, DEBUG are supported levels (0-7)
- klogd and syslogd daemons typically handle these
  - Will go to /var/log/messages, other file, or console terminal based on your syslog config.
- dmesg prints kernel message from /proc/kmsg
- Why not use printf? No glibc!

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

Kernel Debug Printing

- /proc/sys/kernel/printk can be used to control prints redirected to the console

```
ecen5013@ecen5013-VirtualBox:~/ldd3/scull$ cat /proc/sys/kernel/printk
4       4       1       7
```

  - current = 4 WARNING and lower
  - default = 4 WARNING and lower
  - minimum = 1 ALERT and EMERG
  - boot time = 7 DEBUG
- printk is safe to use anywhere (including interrupt handlers)
  - Writes output to a circular buffer

Linux Device Drivers 3rd Edition Chapter 4

Kernel Debug Printing

- Debugging prints, useful for testing but should not be included in production.
  - One option: Add your own macro for printk like PDEBUG

```
#undef PDEBUG             /* undef it, just in case */
#ifdef SCULL_DEBUG
#  ifdef __KERNEL__
     /* This one if debugging is on, and kernel space */
#    define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ## args)
#  else
     /* This one for user space */
#    define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#  endif
#else
#  define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */
```

Linux Device Drivers 3rd Edition Chapter 4

strace

- Use strace to trace system calls and interactions between user space program and the driver

```
ecen5013@ecen5013-VirtualBox:~/ldd3/scull$ strace cat /dev/scull0
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=10281936, ...}) = 0
mmap(NULL, 10281936, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fc695d20000
close(3)                                = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 7), ...}) = 0
openat(AT_FDCWD, "/dev/scull0", O_RDONLY) = 3
fstat(3, {st_mode=S_IFCHR|0664, st_rdev=makedev(240, 0), ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc696ccc000
read(3, "hello_world\n", 131072)        = 12
write(1, "hello_world\n", 12hello_world
)       = 12
read(3, "", 131072)                     = 0
munmap(0x7fc696ccc000, 139264)          = 0
close(3)                                = 0
```

File descriptor = 3 for /dev/scull0

Read 12 bytes from file descriptor 3
Write 12 bytes to stdout (fd 1)

Kernel Dynamic Debug

- Debugging prints, useful for testing but should not be included in production.
  - Option 2: (Not discussed in the book): Use dynamic debug (pr_debug)

```
mount -t debugfs none /sys/kernel/debug
echo "file stm32-adc.c +p" > /sys/kernel/debug/dynamic_debug/control

echo "file stm32-adc.c line 1438 +p" > /sys/kernel/debug/dynamic_debug/control
```

- What is the benefit of this approach over the one mentioned previously?
  - Don’t need to recompile the driver
  - Don’t need special debug builds

https://www.kernel.org/doc/html/latest/admin-guide/dynamic-debug-howto.html
https://wiki.st.com/stm32mpu/wiki/How_to_use_the_kernel_dynamic_debug

Print Ratelimiting

- printk_ratelimited (replaces printk_ratelimit mentioned in Linux Device Drivers Book)
  - Prevents your failure prints from flooding the kernel logs

```
printk_ratelimited(KERN_WARNING
                   "error: reading the clock failed (%d)\n",
                   error);
```

https://elixir.bootlin.com/linux/v5.2.1/source/arch/powerpc/kernel/rtas-rtc.c#L35

System Faults

- System faults - due to a bug in a specific driver
- May result in “panic” - kernel stops executing
- May leave system in a generally unusable state
  - May not be enough to unload/reload your driver
  - May need to reboot the system

Oops Messages

- Example of oops is a null pointer dereference of use of incorrect pointer value.
  - Also known as page faults
- May leave system in a generally unusable state
  - May not be enough to unload/reload your driver
  - May need to reboot the system

Oops Example

```
ssize_t faulty_write (struct file *filp, const char __user *buf, size_t count,
        loff_t *pos)
{

    /* make a simple fault by dereferencing a NULL pointer */
    *(int *)0 = 0;
    return 0;
}
```

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:    0
EIP:    0060:[<d083a064>]    Not tainted
EFLAGS: 00010246   (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000   ebx: 00000000   ecx: 00000000   edx: 00000000
esi: cf8b2460   edi: cf8b2480   ebp: 00000005   esp: c31c5f74
ds: 007b   es: 007b   ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
       ffffffff 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
       00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
 [<c0150558>] vfs_write+0xb8/0x130
 [<c0150682>] sys_write+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb
```

What happened?

Where in code did it occur?

- Module faulty
- 4 bytes into the function faulty_write, which is 0x10 bytes long

Oops Example

- objdump shows you assembly content associated with an object file (in this case a .ko)
  - objdump -S intermixes source with assembly (requires debug info for kernel module builds)
  - buildroot/output/host/bin/aarc64-linux-objdump is our cross objdump utility
- <path to>objdump -S <path to>/module.ko

```
Disassembly of section .text:

0000000000000000 <faulty_write>:
   0:   d2800001        mov     x1, #0x0
   4:   d2800000        mov     x0, #0x0
   8:   b900003f        str     wzr, [x1]
   c:   d65f03c0        ret
```