Linux Kernel Training. Lecture 5

Introduction to kernel debugging

Oleksandr Redchuk
<oleksandr.redchuk@gmail.com>

April 6, 2020. GlobalLogic

Kernel debugging config options

- There are number of features in Linux related to debugging
- Most of them are consolidated under “Kernel Hacking” Kconfig menu
  - CONFIG_DEBUG_KERNEL — master switch
  - CONFIG_DEBUG_SLAB Each byte of allocated memory is set to special values, guard are added before and after allocated memory object.
  - CONFIG_INIT_DEBUG Enables checks for code that attempts to access initialization-time memory after initialization is complete
  - CONFIG_DEBUG_LIST Adds sanity check into list functions

And many others…

Kernel will be slower but be able to catch and report errors.

Kernel debugging config options (cont.)

```
#ifdef CONFIG_DEBUG_LIST
extern bool __list_del_entry_valid(struct list_head *entry);
#else
static inline bool __list_del_entry_valid(struct list_head *entry)
{
    return true;
}
#endif

static inline void __list_del_entry(struct list_head *entry)
{
    if (!__list_del_entry_valid(entry))
           return;

    __list_del(entry->prev, entry->next);
}
```

Controlling printk() behaviour

- Here is how message loglevel is defined in the Linux Kernel

0 (KERN_EMERG)	system is unusable
1 (KERN_ALERT)	action must be taken immediately
2 (KERN_CRIT)	critical conditions
3 (KERN_ERR)	error conditions
4 (KERN_WARNING)	warning conditions
5 (KERN_NOTICE)	normal but significant condition
6 (KERN_INFO)	informational
7 (KERN_DEBUG)	debug-level messages

- All Kernel Messages with a loglevel smaller than the console loglevel will be printed to the console. One can change the limit using:

```
# echo 4 > /proc/sys/kernel/printk
```

Controlling printk() behaviour (cont.)

- One interesting thing we should note explicitly `pr_fmt()` preprocessor macro in `<linux/printk.h>`

```
#ifndef pr_fmt
#define pr_fmt(fmt) fmt
#endif
```

- Your code should define custom `pr_fmt()` preprocessor macro that provides string if custom prefix for `pr_debug()` is desired

```
#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
```

- That would produce following messages with `pr_debug()` in kernel ring buffer

```
module_name_without_ko: fmt_message_goes_here
```

- One can even use

```
#define pr_fmt(fmt) KBUILD_MODNAME ":%s:%d: " fmt, __func__, __LINE__
```

Controlling printk() behaviour (cont.)

- There are number of preprocessor macro in pr_*() family that can be used instead of printk() in most cases. Main benefit from using of such functions is much simpler and cleaner code. They defined in `<linux/printk.h>` too, for example:

```
#define pr_crit(fmt, ...) \
    printk(KERN_CRIT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_err(fmt, ...) \
    printk(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warning(fmt, ...) \
    printk(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warn pr_warning
#define pr_info(fmt, ...) \
    printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
```

Controlling printk() behaviour (cont.)

- There are also macros that print message only once. They use printk_once() macro in their implementation

```
#define printk_once(fmt, ...)                              \
({                                                         \
        static bool __print_once __read_mostly;            \
                                                           \
        if (!__print_once) {                               \
                __print_once = true;                       \
                printk(fmt, ##__VA_ARGS__);                \
        }                                                  \
})
```

- and there are helpers, for example

```
#define pr_err_once(fmt, ...)                    \
    printk_once(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warn_once(fmt, ...)                   \
    printk_once(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)
```

Controlling printk() behaviour (cont.)

- printk itself is fast. It just prints into string and places the string in circular buffer. Message printing might be quite slow especially for serial console.
  - printk to buffer: a few microseconds (but this time can be sufficient if there are a lot of messages)
  - printk to serial console: a few **milli**seconds
- There is also ratelimited version of printk. Use it in code paths where message might be generated too often (e.g. packet coming from network and we print message to ring buffer).

```
#define printk_ratelimited(fmt, ...)                                  \
({                                                                    \
        static DEFINE_RATELIMIT_STATE(_rs,                            \
                                      DEFAULT_RATELIMIT_INTERVAL,     \
                                      DEFAULT_RATELIMIT_BURST);       \
                                                                      \
        if (__ratelimit(&_rs))                                        \
                printk(fmt, ##__VA_ARGS__);                           \
})
```

Controlling printk() behaviour (cont.)

- Also, there are corresponding helpers

```
#define pr_err_ratelimited(fmt, ...)                                \
    printk_ratelimited(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
...
```

- For device drivers there is a special set of printk() functions
- They are prefixed with dev_*()
- They defined in `<linux/device.h>`
- Use them in your device driver
- Same applies to network subsystem where net_ratelimited_*() functions should be used
- They defined in <linux/net.h>
- Use them in your network device driver

Controlling printk() behaviour (cont.)

- There is also interface in proc filesystem to control printk() behaviour
  - One can delay each message printing by specified number of milliseconds via sysctl -w kernel.printk_delay = <msec> knob
  - One can control message rate limiting by means of sysctl -w kernel.printk_ratelimit = <jiffies> and sysctl -w kernel.printk_ratelimit_burst = <number>
- Also console log level can be specified
  - Via Linux command line parameter loglevel=
  - Via sysctl -w kernel.printk = c d m f

(c) console_loglevel	messages with higher priority will be printed
(d) default_message_loglevel	messages without explicit priority assigned this one
(m) minimum_console_loglevel	minimum (highest) that console_loglevel can be set
(f) default_console_loglevel	default value of console_loglevel

  - Via dmesg(1) -n or --console-level parameter

Debugfs: A memory mapped file system

- It is used by kernel developers to put/get information from kernel code to user space
- There is no rules that force any restriction on format or contents of data in debugfs: developers are free to put information that they wish in format whey think is best
- As such none of interfaces in debugfs should be considered as API and to be used in user space for that purpose. The only purpose of interfaces in debugfs is to provide information from kernel to user space useful for debugging kernel interfaces
- Assuming above debugfs root available only to superuser by default

Debugfs: A memory mapped file system (cont.)

- It is mounted by super user from user space via following command (see /etc/init.d/rcS in our busybox directories)

```
mount -t debugfs none /sys/kernel/debug
```

- However there is options uid, gid and mode to make debugfs root available to non super user. Also it is possible to mount it to location other than `/sys/kernel/debug`
- There is another restriction with using debugfs from your kernel code:
  - it’s API available to GPL-only code, which means that modules with proprietary or “dual” licensing can not use debugfs API
- All debugfs API is available via inclusion of `<linux/debugfs.h>` in your code

Debugfs: A memory mapped file system (cont.)

- There are number of places in Linux Kernel where debugfs is used, as well as in many external third-party modules
- We will not cover debugfs API for kernel code right now, as it requires bit understanding of Linux Virtual File System (VFS) interface
- However we will use debugfs interfaces provided by various subsystems and features in Linux Kernel.
- One of such wonderful feature is dynamic debugging facility that permits for low overhead debug messages insertion in source code. It uses debugfs to control debug messages generation scopes, severity levels and more.
- You may refer to Documentation/filesystems/debugfs.txt for more information.
- We will not cover debugfs API for kernel code right now, as it requires bit understanding of Linux Virtual File System (VFS) interface.
- However we will use debugfs interfaces provided by various subsystems and features in Linux Kernel.

dyndbg: Dynamic debugging

- It is quite common to have various messages in user space source code that are print to either console or to dedicated log file(s)
- Developing code for kernel does not differ in this approach from user space too much. However it is not specific to write debug messages to file(s)
- On the other hand there is need to control when to output this kind of messages from code
- Also printing huge amount of messages may affect system performance
- Assuming all the above flexible mechanism to generate and control generation is added to Linux Kernel. That mechanism is called dynamic debugging (dyndbg for short)

dyndbg: Dynamic debugging (cont.)

- Previously printk() is used to implement debug message output
- And each piece of code in-tree and out-of-tree kernel code invent their own wheel to do that
- With old, still supported, but not preferred approach code did something like

```
#ifdef DEBUG
#define DPRINTF(fmt, args...)                                \
        do {                                                 \
                if (printk_ratelimit())                      \
                printk(KERN_DEBUG fmt, ## args);             \
        } while (0);                                         \
#endif /* DEBUG */
```

- With new all this stuff replaced with pr_debug() and family functions

dyndbg: Dynamic debugging (cont.)

- Here is `pr_debug()` definition in `<linux/printk.h>`

```
/* If you are writing a driver, please use dev_dbg instead */
#if defined(CONFIG_DYNAMIC_DEBUG)
/* dynamic_pr_debug() uses pr_fmt() internally so we don't need it here */
#define pr_debug(fmt, ...) \
    dynamic_pr_debug(fmt, ##__VA_ARGS__)
#elif defined(DEBUG)
#define pr_debug(fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#else
#define pr_debug(fmt, ...) \
    no_printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#endif
```

dyndbg: Dynamic debugging (cont.)

- There is also pr_devel() preprocessor macro that evaluates to printk() if DEBUG is on and to no_printk() when not defined

```
/* pr_devel() should produce zero code unless DEBUG is defined */
#ifdef DEBUG
#define pr_devel(fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#else
#define pr_devel(fmt, ...) \
    no_printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#endif
```

dyndbg: Dynamic debugging (cont.)

- and no_printk() is just a dummy helper inline function to let gcc to check fmt arguments. It is used when CONFIG_PRINTK isn't active

```
/*
 * Dummy printk for disabled debugging statements to use whilst maintaining
 * gcc's format and side-effect checking.
 */
static inline __printf(1, 2)
int no_printk(const char *fmt, ...)
{
        return 0;
}
```

- It is defined in `<linux/printk.h>` too

dyndbg: Dynamic debugging (cont.)

- Dynamic debug is compiled when CONFIG_DYNAMIC_DEBUG is active
- By default no pr_debug() messages would be printed
- To control what messages to print at early boot stages where debugfs isn't initialized one can pass parameters via Linux command line from bootloader

```
# dyndbg parameter is used to activate pr_debug() in non-module code
# module.dyndbg format is used to activate pr_debug() in modules
# (including built-in)

… dyndbg=QUERY module_name.dyndbg=QUERY ...
```

- When debugfs is available one can control where pr_debug() is active by reading/writing to /sys/kernel/debug/dynamic_debug/control.

dyndbg: Dynamic debugging (cont.)

- There is special format of QUERY messages going to either command line parameter or to dynamic_debug/control file

```
$ sysfs='/sys/kernel/debug'
$ echo "command; command; command; ..." >$sysfs/dynamic_debug/control

# command ::= match-spec* flags-spec
#
# match-spec ::= 'func' string |
#                'file' string |
#                'module' string |
#                'format' string |
#                'line' line-range
#
# line-range ::= lineno | '-'lineno | lineno'-' | lineno'-'lineno
#
# lineno ::= unsigned-int
```

dyndbg: Dynamic debugging (cont.)

- The flags specification comprises a change operation followed by one or more flag characters. The change operation is one of the characters:

-	remove the given flags
+	add the given flags
=	set the flags to the given flags

- The flags are:

p	enables the pr_debug() callsite.
f	Include the function name in the printed message
l	Include line number in the printed message
m	Include module name in the printed message
t	Include thread ID in messages not generated from interrupt context
_	No flags are set. (Or'd with others on input)

- Refer to Documentation/dynamic-debug-howto.txt as more complete source of information about dynamic debugging feature

dyndbg: Dynamic debugging (cont.)

- Let’s replace pr_info() by pr_debug() in hello_exit()
- Do not enable debug messages:

```
~ # insmod hello.ko count=2
[   20.750420] hello: loading out-of-tree module taints kernel.
[   20.757132] hello_init: Hello, world!
[   20.759883] hello_init: Hello, world!
~ # cat /sys/kernel/debug/dynamic_debug/control  | grep hello.c
..../hello.c:85 [hello]hello_exit =_ "%s: %lld\012"
~ # rmmod hello
~ # dmesg | tail -2
[   20.757132] hello_init: Hello, world!
[   20.759883] hello_init: Hello, world!
~ #
```

dyndbg: Dynamic debugging (cont.)

- Enable debug messages:

```
~ # insmod hello.ko count=2
[   54.008406] hello_init: Hello, world!
[   54.011169] hello_init: Hello, world!
~ # echo 'file hello.c line 85 +p' > /sys/kernel/debug/dynamic_debug/control
~ # cat /sys/kernel/debug/dynamic_debug/control  | grep hello.c
..../hello.c:85 [hello]hello_exit =p "%s: %lld\012"
~ # rmmod hello
~ # dmesg | tail -3
[   54.011169] hello_init: Hello, world!
[   78.001123] hello_exit: 54005378298
[   78.001138] hello_exit: 54008150089
~ #
```

Hunting the bugs at compile time

- It is always good to catch some odd changes to the code at compile time when it is known that only subset of options is valid and adding something other than is supported without changing implementation is known to be buggy.
- There is a number of build time checks performed by preprocessor, compiler and semantics parsers like sparse, however neither of them can catch case described above.
- For that purpose Linux has number of preprocessor macros, that evaluate (if possible, otherwise it is illegal to use them) condition at build time and raise build error.

Hunting the bugs at compile time (cont.)

- These routines are defined in `<linux/bug.h>`
  - **BUILD_BUG**() - stop build unconditionally.
  - **BUILD_BUG_ON**(condition) - stop build when @condition evaluates to *true*. Note that @condition must be compile time evaluable (e.g. integer constant, known pointer value, and any expression with values known at compile time).
  - **BUILD_BUG_ON_ZERO**(condition) - stop build when @condition is *true*. Can be used as structure member initializer and returns (size_t) 0 if @condition is *false*.
  - **BUILD_BUG_ON_NULL**(condition) - stop build when @condition is *true*, Can be used as structure member initializer and returns NULL if @condition is *false*.
  - **BUILD_BUG_ON_NOT_POWER_OF_2**(expr) - stop build if @expr isn’t power of two.

Hunting the bugs at compile time (cont.)

- Let’s look at their functionality by examples

```
static inline void dst_hold(struct dst_entry *dst)
{
    /*
    * If your kernel compilation stops here, please check
    * __pad_to_align_refcnt declaration in struct dst_entry
    */
    BUILD_BUG_ON(offsetof(struct dst_entry, __refcnt) & 63);
    atomic_inc(&dst->__refcnt);
}
```

- Here is compilation would stop if offset of __refcnt field in struct dst_entry isn’t aligned to 64 bytes (i.e. bits 0-5, 2^6 - 1 == 63 should be zero).

Hunting the bugs at compile time (cont.)

- There are other hints on bug hunting at compile time. With some of them you should be already familiar from previous lecture.
  - Do not ignore compiler/preprocessor warnings: they pointing you to potential problems within your code
  - Use extra warning levels by passing W=1..3 option to make
  - Use sparse(1). This gives you even more warnings you can't expect from compiler/preprocessor
  - Use static analysis and report generation options like `make check_stack`
  - Take look at → Kernel hacking → Compile-time checks and compiler options in during kernel configuration
  - Build source using different gcc versions or even use different compiler (e.g. clang, icc) they might give you more points for debug.

OOPS and Panic

- **OOPS** it is runtime, **non-fatal** condition happening in response to some unspecified/unhandled behaviour in functionality or triggered by external events during the system operation (e.g. hotplug, process kill due to OOM). System can either recover from such condition without any data/functionality loss and continue to run, even with limited functionality.
- **Panic** is a **fatal** condition, where system can not continue operations without significant functionality and/or data loss nor it can recover from such conditional safely. Often after system recovers from certain non-fatal OOPS it might trigger panic later as normal system functionality is compromised, some vital data structures might be corrupted.

OOPS and Panic (cont.)

What Linux actually does on OOPS?

- It is architecture dependent
- In general there are traps configured for exceptions like page fault, general protection, invalid opcode, divide by zero, alignment error (for platforms with strict alignment rules like some SPARC), etc.
- Then, depending on trap handlers implementation it calls `notify_die()` to call registered with `register_die_notifier()` notifiers (callback functions) that inform it's subscribers about exception (one of such subscribers is kgdb)
- Finally, if exception related to user mode, it is translated to signal (e.g. SIGSEGV). Overwise it triggers architecture dependent die() call which makes informative reports we see on OOPS.

OOPS and Panic (cont.)

Why and how to control behaviour on OOPS?
OOPS may cause system to panic immediately, rather than continue

- To catch problem at first place (e.g. paging fault at NULL pointer dereference)
- Eases debugging process
- By using oops=panic on Linux cmdline from bootloader one can instruct kernel to treat OOPS as fatal error and thus panic.
- It is also possible to control value of this parameter at runtime via sysctl interface in proc filesystem.

```
~# sysctl -w kernel.panic_on_oops = 1
```

On the same side as OOPS/Panic: warnings, traces

- There is another class conditions, which are similar to OOPS in sense they are not fatal nor even may indicate any recovery made/needed by/from the system nor user to address them.
- They typically might indicate some misconfiguration, misunderstanding in use of some kernel APIs/interfaces, etc.
- They have different from OOPS/Panic nature and usually added to code with means like WARN(), WARN_ON(), WARN_ONCE(), WARN_ON_ONCE() and dump_stack() in places where attention is needed due to potential problems using approach.

On the same side as OOPS/Panic: warnings, traces

Runtime warning routines

- They are defined as generic in `<asm-generic/bug.h>` as preprocessor macros and might be overwritten by arch specific code (WARN_ON() currently).
  - **WARN**(condition, format...) - trigger warning message when conditional evaluates to true. Use printk() for message to print @format with optional arguments message
  - **WARN_ON**(condition) - trigger warning message when conditional evaluates to true
  - **WARN_ONCE**(condition, format...) - like WARN(), but print warning message only once
  - **WARN_ON_ONCE**(condition) - like WARN_ON(), but print warning message only once
- They might be used in conditional statements because they return either 0 when warning isn't triggered or 1 when it is. They might just check for conditional if CONFIG_BUG isn't enabled.

```
if (WARN_ON_ONCE((rt->dst.flags & DST_NOCACHE) &&
                 !atomic_read(&rt->dst.__refcnt)))
        return -EINVAL;
```

Stack trace dump routines

- It might be required to dump stack trace at the same error code path. For example, when implementing custom BUG()/WARN() functionality that does not depend on CONFIG_BUG or have to add custom title before trace message. It might depend on some other conditions like CONFIG_FOO for current driver/module or always present to let user runtime checks.
- One of the good examples of such use of dump_stack() is

```
#define ASSERT_RTNL() do {                                   \
    if (unlikely(!rtnl_is_locked())) {                       \
           printk(KERN_ERR "RTNL: assertion failed at %s (%d)\n", \
                   __FILE__,  __LINE__);                     \
           dump_stack();                                     \
    }                                                        \
} while(0)
```

Trigger OOPS/Panic from kernel code

- here are two preprocessor defines in `<asm-generic/bug.h>`, that use `panic()` in their implementation:
  - BUG() trigger panic unconditionally
  - BUG_ON(condition) trigger panic when @condition evaluates to *true*

```
#define BUG() do {                                                    \
    printk("BUG: failure at %s:%d/%s()!\n", __FILE__, __LINE__, __func__); \
    panic("BUG!");                                                    \
} while (0)

#define BUG_ON(condition) do { if (unlikely(condition)) BUG(); } while (0)
```

- Note that arch code might provide overrides for both BUG() and BUG_ON() and thus it is not necessary that panic() is called on BUG(): it might be just OOPS.
- Also from user space

```
~# echo ‘c’ >/proc/sysrq-trigger
```

OOPS message format

- In general it is architecture dependent, but have some generic structure
- It might contain following data for tracing to problem origin
    - Register contents
    - Stack back traces
    - Strings describing hardware where message is triggered
    - String describing type of the problem (e.g. page fault, division by zero, etc)
    - List of modules linked to the kernel
    - Position of Instruction Pointer (IP) and symbolic name of function owning it
    - Various values from Linux Kernel specific data structures (e.g. page directory entry (pde), page table entry (pte), etc).

OOPS message format (cont.)

Let’s generate it

```
struct time_data {
    ktime_t start_time;          /* 0 */
    struct list_head list;       /* 8 */
};

__attribute__((__noinline__))
static void add_to_list(struct list_head *node)
{
    BUG_ON(!node);               /* It does not trigger oops because node is not NULL! */
    list_add_tail(node, &time_list);
}

static int __init hello_init(void)
{
    struct time_data *time_data = 0; // kmalloc(sizeof(*time_data), GFP_KERNEL);

    add_to_list(&time_data->list);
    /* … */
    return 0;
};
```

OOPS message format (cont.)

Changes in Makefile

```
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
obj-m   := hello.o
ccflags-y += -g                               # add debugging info
else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
    $(MAKE) -C $(KDIR) M=$$PWD
    cp hello.ko hello.ko.unstripped
    $(CROSS_COMPILE)strip -g hello.ko         # strip only debugging info

clean:
    $(MAKE) -C $(KDIR) M=$$PWD clean

%.s %.i: %.c                                  # just use make hello.s instead of objdump
    $(MAKE) -C $(KDIR) M=$$PWD $@

endif
```

OOPS message format (cont.)

```
~ # insmod hello.ko
[   13.704625] hello: loading out-of-tree module taints kernel.
[   13.711132] Unable to handle kernel NULL pointer dereference at virtual address 00000008
[   13.719693] pgd = d6388774
[   13.722518] [00000008] *pgd=9a2d9831, *pte=00000000, *ppte=00000000
[   13.729100] Internal error: Oops: 817 [#1] SMP ARM
[   13.734105] Modules linked in: hello(O+)
[   13.738208] CPU: 0 PID: 78 Comm: insmod Tainted: G           O      4.19.114 #5
[   13.745834] Hardware name: Generic AM33XX (Flattened Device Tree)
[   13.752206] PC is at add_to_list+0x1c/0x2c [hello]
[   13.757207] LR is at hello_init+0xc/0x1000 [hello]
[   13.762205] pc : [<bf00001c>]    lr : [<bf00500c>]    psr: 200f0013
[   13.768743] sp : da255dc8  ip : da237ac0  fp : 00000000
[   13.774192] r10: bf002040  r9 : c1704c48  r8 : 00000000
[   13.779644] r7 : bf005000  r6 : ffffe000  r5 : c1704c48  r4 : c1888140
[   13.786455] r3 : bf002000  r2 : bf002000  r1 : 00005782  r0 : 00000008
[   13.793267] Flags: nzCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment none
[   13.800711] Control: 10c5387d  Table: 9a258019  DAC: 00000051
[   13.806707] Process insmod (pid: 78, stack limit = 0x7fb7c0e3)
[   14.022583] Exception stack(0xda255fa8 to 0xda255ff0)
[   13.817342] 5dc0:                   c1888140 c0302d70 db16a400 00000000 00210d00 da255ddc
[   13.825880] 5de0: c1704c48 da2dc840 8040003f bf002088 bf002088 51b4471b ffe00000 da2dc8c0
```

OOPS message format (cont.)

```
[   13.812792] Stack: (0xda255dc8 to 0xda256000)
[   13.817342] 5dc0:                   c1888140 c0302d70 db16a400 00000000 00210d00 da255ddc
[   13.825880] 5de0: c1704c48 da2dc840 8040003f bf002088 bf002088 51b4471b ffe00000 da2dc8c0
[   13.834417] 5e00: 8040003e e1261000 da2dc8c0 51b4471b e1261000 da2dc840 bf002040 51b4471b
...
[   13.953953] 5fc0: b6f0738c b6f26950 000013a4 00000080 00000001 bec54e8c 001086c4 000f411e
[   13.962491] 5fe0: bec54b48 bec54b38 0003b270 b6de01b0 600f0010 0011b1d8 00000000 00000000
[   13.971046] [<bf00001c>] (add_to_list [hello]) from [<bf00500c>] (hello_init+0xc/0x1000
[hello])
[   13.980233] [<bf00500c>] (hello_init [hello]) from[<c0302d70>] (do_one_initcall+0x54/0x208)
[   13.989060] [<c0302d70>] (do_one_initcall) from [<c03d6160>] (do_init_module+0x64/0x214)
[   13.997513] [<c03d6160>] (do_init_module) from [<c03d8654>] (load_module+0x22dc/0x2628)
[   14.005871] [<c03d8654>] (load_module) from [<c03d8b10>] (sys_init_module+0x170/0x1c4)
[   14.014139] [<c03d8b10>] (sys_init_module) from [<c0301000>] (ret_fast_syscall+0x0/0x54)
[   14.022583] Exception stack(0xda255fa8 to 0xda255ff0)
[   14.027856] 5fa0:                   b6f0738c b6f26950 0011b1d8 000013a4 001086c4 00000000
[   14.036395] 5fc0: b6f0738c b6f26950 000013a4 00000080 00000001 bec54e8c 001086c4 000f411e
[   14.044930] 5fe0: bec54b48 bec54b38 0003b270 b6de01b0
[   14.050203] Code: e3023000 e34b3f00 e5932004 e5830004 (e5803000)
[   14.056636] ---[ end trace 9ddb61d277c49039 ]---
Segmentation fault
```

OOPS message format (cont.)

Let's find it (objdump): `PC is at add_to_list+0x1c/0x2c`

```
$ ${CROSS_COMPILE}objdump -dS hello.ko.unstripped | less
00000000 <add_to_list>:
__attribute__((__noinline__)) static void add_to_list(struct list_head *node)
{
          BUG_ON(!node);
   0:    e3500000              cmp       r0, #0
   4:    1a000000              bne       c <add_to_list+0xc>
   8:    e7f001f2              .word    0xe7f001f2
...
static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
          __list_add(new, head->prev, head);
   c:    e3003000              movw      r3, #0
  10:    e3403000              movt      r3, #0
  14:    e5932004              ldr       r2, [r3, #4]
          next->prev = new;
  18:    e5830004              str       r0, [r3, #4]
          new->next = next;
  1c:    e5803000              str       r3, [r0]
          new->prev = prev;
  20:    e5802004              str       r2, [r0, #4]
```

Delta is 0x1c

OOPS message format (cont.)

Let’s find it (gdb): `PC is at add_to_list+0x1c/0x2c`

Notice that we added 4 (the length of the command at offset 0x1c) to upper bound of a range to disassemble because gdb uses [from, to) range notation (to is the first address that will not be disassembled).

```
$ ${CROSS_COMPILE}gdb -q hello.o
(gdb) disas add_to_list+0x0c,add_to_list+0x1c+4
Dump of assembler code from 0x18 to 0x2c:
   0x00000018 <add_to_list+12>:    movw    r3, #0
   0x0000001c <add_to_list+16>:    movt    r3, #0
   0x00000020 <add_to_list+20>:    ldr    r2, [r3, #4]
   0x00000024 <add_to_list+24>:    str    r0, [r3, #4]
   0x00000028 <add_to_list+28>:    str    r3, [r0]
End of assembler dump.
(gdb) quit
$
```

OOPS message format (cont.)

THUMB2 mode (CONFIG_THUMB2_KERNEL)

```
~ # insmod hello.ko
[   35.117134] hello: loading out-of-tree module taints kernel.
[   35.123766] Unable to handle kernel NULL pointer dereference at virtual address 00000008
[   35.132332] pgd = 83ba745f
[   35.135165] [00000008] *pgd=9a294831, *pte=00000000, *ppte=00000000
[   35.141743] Internal error: Oops: 817 [#1] SMP THUMB2
[   35.147034] Modules linked in: hello(O+)
[   35.151152] CPU: 0 PID: 81 Comm: insmod Tainted: G         O        4.19.114 #6
[   35.158807] Hardware name: Generic AM33XX (Flattened Device Tree)
[   35.165201] PC is at hello_init+0x21/0xfff [hello]
[   35.170228] LR is at do_one_initcall+0x3f/0x16c
[   35.174971] pc : [<bf805022>]          lr : [<c0302ac3>]    psr: 200f0033
[   35.181533] sp : da2d3db8  ip : da27b240  fp : c1404c48
[   35.187003] r10: da27bdc8  r9 : bf802040  r8 : 00000000
[   35.192474] r7 : bf805001  r6 : ffffe000  r5 : 00000008  r4 : 00000000
[   35.199309] r3 : bf802000  r2 : bf802240  r1 : bf802000  r0 : bf801050
...
...
[    35.463474] Code: 0050 605d f6cb 7080 (60a3) 60e1
```

References

- Linux Device Drivers, Third Edition. Chapter 4: Debugging Techniques
- Debugging by printing (elinux.org)
- Documentation/admin-guide/dynamic-debug-howto.rst
- Documentation/filesystems/debugfs.txt
- Documentation/filesystems/proc.txt
- Documentation/filesystems/sysfs.txt
- Documentation/admin-guide/sysfs-rules.rst