

基础篇：通过 log 分析 KE

一：KE 概念

Android OS 由 3 层组成，最底层是 kernel，上面是 native bin/lib，最上层是 java 层：



任何软件都有可能发生异常，比如野指针，跑飞、死锁等等。

异常发生在 kernel 层，我们就叫它为 KE (kernel exception)，同理，发生在 native 就是 NE，java 层就是 JE。这篇文章仅关注底层的 KE。

KE 类别

kernel 有 2 中崩溃类别，

oops （类似 **assert**，有机会恢复）

- oops 是美国人比较常用的口语。就是有点意外，吃惊，或突然的意思。内核行为表现为通知感兴趣模块，**打印各种信息，如寄存器值，堆栈信息...**
- 当出现 oops 时，我们就可以根据寄存器等信息调试并解决问题。
- `/proc/sys/kernel/panic_on_oops` 为 1 时导致 panic。我们**默认设置为 1**，即 oops 会发生 panic。

panic

- Panic - 困惑，恐慌，它表示 Linux kernel 遇到了一个不知道该怎么继续的情况。内核行为表现为通知感兴趣模块，**死机或者重启**。
- 在 kernel 代码里，有些代码加了错误检查，发现错误可能直接调用了 `panic()`，并输出信息提供调试。

其实不管分类几种，都表示 kernel 出现故障，需要修复。那如何调试呢？就要看发生异常时留了哪些信息帮我们定位问题了。

常用调试方法

凡是程序就有 bug。bug 总是出现在预料之外的地方。据说世界上第一个 bug 是继电器式计算机中飞进一只蛾子，倒霉的飞蛾夹在继电器之间导致了计算机故障。由于这个小虫子，程序中的错误就被称为 bug。

有 Bug 就需要 Debug，而调试是一种很个性化的工作，十个人可能有十种调试方法。但从手段上来讲，大致可分为两类，在线调试 (Online Debug) 和离线调试 (Offline Debug)。

- **在线调试, Online debug**, 指的是在程序的运行过程中监视程序的行为, 分析是否符合预期。通常会借助一些工具, 如 GDB 和 Trace32 等。有时候也会借助一些硬件设备的协助, 如仿真器/JTAG, 但是准备环境非常困难, 而且用起来也很麻烦, 除非一些 runtime 问题需要外很少使用。
- **离线调试, Offline debug**, 指的是在程序的运行中收集需要的信息, 在 Bug 发生后根据收集到的信息来分析的一种手段。通常也分为两种方式, 一种是 Logging, 一种是 Memory Dump。
 - **Logging**, 日志或者相关信息的收集, 可以比较清晰的看到代码的执行过程, 对于逻辑问题是一种有效的分析手段, 由于其简单易操作, 也是最为重要的一种分析手法。
 - **Memory Dump**, 翻译过来叫做内存转储, 指的是在异常发生的时刻将内存信息全部转储到外部存储器, 即将异常现场信息备份下来以供事后分析。是针对 CPU 执行异常的一种非常有效的分析手段。在 Windows 平台, 程序异常发生之后可以选择启动调试器来马上调试。在 Linux 平台, 程序发生异常之后会转储 core dump, 而此 core dump 可以用调试器 GDB 来进行调试。而内核的异常也可以进行类似的转储。

下面我们由浅入深剖析各种调试方法, 先从 logging 开始吧。

二: kernel 空间布局

kernel space

在分析 KE 前, 你要了解 kernel 内存布局, 才知道*哪些地址用来做什么, 可能会是什么问题。*

在内核空间中存在如下重要的段:

- vmlinux 代码/数据段: 任何程序都有 TEXT(可执行代码), RW(数据段), ZI 段(未初始化数据段), kernel 也有, 对应的是 .text, .data, .bss
- module 区域: kernel 可以支持 ko (模块), 因此需要一段空间用于存储代码和数据段。
- vmalloc 区域: kernel 除了可以申请连续物理地址的内存外, 还可以申请不连续的内存 (虚拟地址是连续的), 可以避免内存碎片化而申请不到内存。
- io map 区域: 留给 io 寄存器映射的区域, 有些版本没有 io map 区域而是直接用 vmalloc 区域了。
- memmap: kernel 是通过 page 结构体描述内存的, 每一个页框都有对应的 page 结构体, 而 memmap 就是 page 结构体数组。

还有其他段小的段没有列出来, 可能根据不同的版本而差别。

目前智能机已进入 64bit, 因此就存在 32bit 布局和 64bit 布局, 下面一一讲解。

ARM64bit kernel 布局

ARM64 可以使用多达 48bit 物理、虚拟地址 (扩充成 64bit, 高位全为 1 或 0)。对 linux kernel 来讲, 目前配置为 39bit 的 kernel 空间。

由于多达 512GB 的空间, 因此完全可以将整个 RAM 映射进来, 0xFFFFFFFFC00000000 之后就是一一映射了, 就无所谓 high memory 了。

vmalloc 区域功能除了外设寄存器也直接映射到 vmalloc 了, 就没有 32bit 布局里的 IO map space 了。

不同版本的 kernel, 布局稍有差别:

kernel-3.10

Start	End	Size	Use

0000000000000000	0000007fffffff	512GB	user
ffffff8000000000	ffffffbffffeffff	~240GB	vmalloc

ffffffbfffff0000	ffffffbfffffffff	64KB	[guard page]
ffffffbc00000000	ffffffbdfbfeffff	8GB	vmemmap
ffffffbe00000000	ffffffbfbfbfffff	~8GB	[guard]
ffffffbfbfc00000	ffffffbfbdfbfeffff	2MB	early con I/O space
ffffffbfbfe00000	ffffffbfbfbfffff	2MB	PCI I/O space
ffffffbffc000000	ffffffbfffffffff	64MB	modules
ffffffc000000000	ffffffffffffffff	256GB	kernel logical memory map

>= kernel-3.18 && < kernel-4.6

Start	End	Size	Use

0000000000000000	0000007fffffffff	512GB	user
ffffff8000000000	ffffffbdfbfeffff	~247GB	vmalloc
ffffffbdfbf00000	ffffffbdfbfeffff	64KB	[guard page]
ffffffbe00000000	ffffffbfbfbfffff	7GB	vmemmap
ffffffbfc0000000	ffffffbfbdfcffff	957MB	[guard]
ffffffbfbdfd0000	ffffffbfbdfeffff	8KB	fixed mappings
ffffffbfbdff0000	ffffffbfbfbfffff	~2MB	[guard]
ffffffbffc000000	ffffffbfffffffff	64MB	modules
ffffffc000000000	ffffffffffffffff	256GB	kernel logical memory map

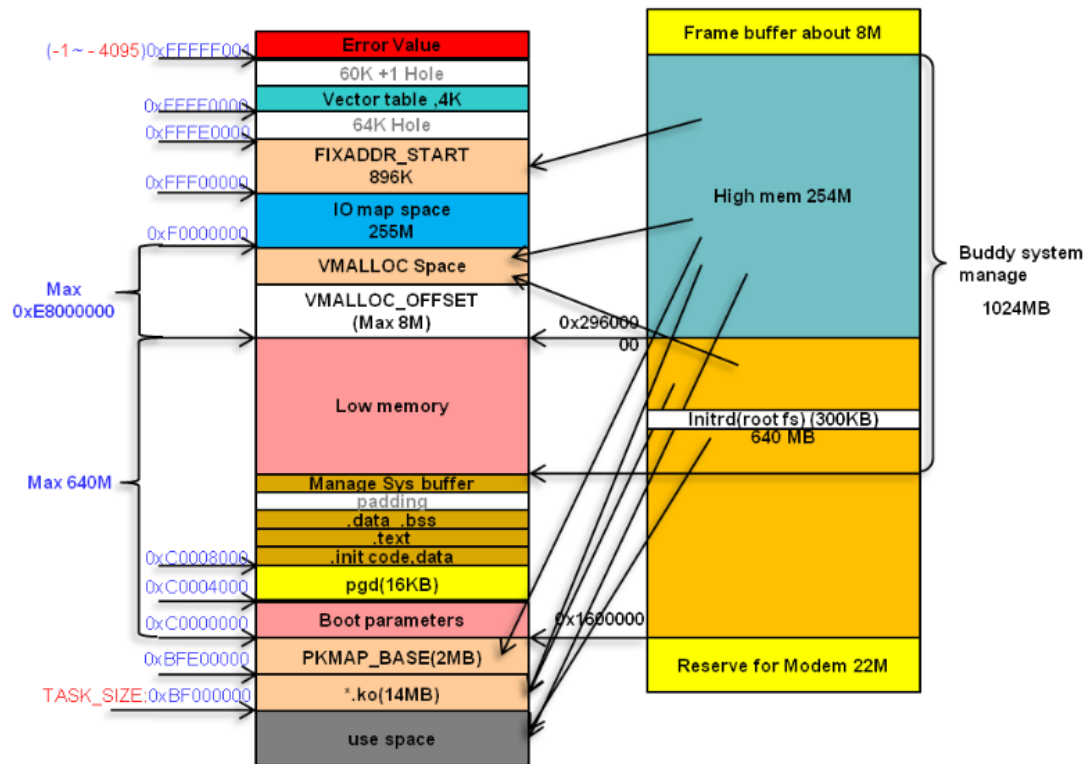
>= kernel-4.6/N0.MP8 kernel-4.4(patch back)

Start	End	Size	Use

0000000000000000	0000007fffffffff	512GB	user
ffffff8000000000	ffffff8008000000	128MB	modules
ffffff8008080000	vmlinux
ffffff8008000000	ffffffbdbbff0000	246GB	vmalloc
ffffffbdbbff0000	ffffffbdc0000000	64KB	[guard page]
ffffffbdc0000000	ffffffbfc0000000	8GB	vmemmap
ffffffbfc0000000	ffffffbf7fd000	~1G	[guard]
ffffffbf7fd000	ffffffbf7fec0000	4108KB	fixed mappings
ffffffbf7fec0000	ffffffbf7fee0000	2MB	[guard]
ffffffbf7fee0000	ffffffbf7fe00000	16MB	PCI I/O
ffffffbf7fe00000	ffffffc000000000	2MB	[guard]
ffffffc000000000	ffffffffffffffff	256GB	kernel logical memory map

ARM32bit kernel 布局

这是一张示意图（有些地址可能会有差异）



整个地址空间是 4G，kernel 被配置为 1G，程序占 3G。

内核代码开始的地址是 0xC0008000，前面放页表（起始地址为 0xC0004000），如果支持模块（*.ko）那么地址在 0xBF000000。

由于 kernel 没办法将所有内存都映射进来，毕竟 kernel 自己只占 1G，如果 RAM 超过 1G，就无法全部映射。怎么办呢？只能先映射一部分了，这部分叫 low memory。其他的就按需映射，VMALLOC 区域就是用于按需映射的。

ARM 的外设寄存器和内存一样，都统一地址编码，因此 0xF0000000 以上的一段空间用于映射外设寄存器，便于操作硬件模块。

0xFFFF0000 是特殊地址，CPU 用于存放异常向量表，kernel 异常绝大部分都是 CPU 异常（MMU 发出的 abort/undef inst. 等异常）。

结语

以上是粗略的说明，还需查看代码获取完整的分析信息（内核在不停演进，有些部分可能还会变化）。

三：了解 printk

kernel log

最初学编程时，大家一定用过 printf()，在 kernel 里有对应的函数，叫 printk()。

最简单的调试方法就是用 printk() 印出你想知道的信息了，而前面章节讲到 oops/panic 时，它们就通过 printk() 将寄存器信息/堆栈信息打印到 kernel log buffer 里。

了解 kernel log 对问题的调试将非常重要，这里有专门的课程介绍，请看：

- MediaTek On-Line> Quick Start> Deep in MTK Turnkey Solution Logging Tools> Kernel log。

可以看到 kernel log 可以通过串口输出，也可以在发生 oops/panic 后将 buffer 保存成文件打包到 db 里，然后拿到串口 log 或 db 对 kernel 进行调试分析了。

通常手机会保留串口测试点，但要抓串口 log 一般都要拆机，比较麻烦。前面讲到可以将 kernel log 保存成文件打包在 db 里，db 是什么东西？

AEE db

db 是叫 AEE（Android Exception Engine，集成在 Mediatek 手机软件里）的模块检查到异常并收集异常信息生成的文件，里面包含调试所需的 log 等关键信息。db 有点像飞机的黑匣子。

对于 KE 来说，db 里包含了如下文件（db 可以通过 GAT 工具解开，请参考附录里的 FAQ）：

__exp_main.txt:	异常类型，调用栈等关键信息。
__exp_detail.txt:	详细异常信息
SYS_ANDROID_LOG:	android main log
SYS_KERNEL_LOG:	kernel log
SYS_LAST_KMSG:	上次重启前的 kernel log
SYS_MINI_RDUMP:	类似 coredump，可以用 gdb/trace32 调试
SYS_REBOOT_REASON:	重启时的硬件记录的信息。
SYS_VERSION_INFO:	kernel 版本，用于和 vmlinux 对比，只有匹配的 vmlinux 才能用于分析这个异常。
SYS_WDT_LOG:	看门狗复位信息
.....	

以上这些文件一般足以调试 KE 了，除非一些特别的问题需要其他信息，比如串口 log 等等。

四：ram console

什么是 ram console？

请参考：

- [MediaTek On-Line](#)> [Quick Start](#)> Deep in MTK Turnkey Solution Logging Tools

系统重启时关键信息

- ram console 除了保持 last kmsg 外，还有重要的系统信息，这些非常有助于我们调试。这些信息保存在 ram console 的头部 ram_console_buffer 里。

```
struct ram_console_buffer
{
    uint32_t sig;

    /* for size comptible */

    uint32_t off_pl;

    uint32_t off_lpl; /* last preloader: struct reboot_reason_pl*/

    uint32_t sz_pl;
```

```

uint32_t off_lk;

uint32_t off_l1k; /* last lk: struct reboot_reason_lk */

uint32_t sz_lk;

uint32_t padding[3];

uint32_t sz_buffer;

uint32_t off_linux; /* struct last_reboot_reason */

uint32_t off_console;


/* console buffer*/

uint32_t log_start;

uint32_t log_size;

uint32_t sz_console;

};

```

这个结构体里的 off_linux 指向了 struct last_reboot_reason，里面保存了重要的信息：

```

struct last_reboot_reason
{
    uint32_t fiq_step;

    uint32_t exp_type; /* 0xaeedeadX: X=1 (HWT), X=2 (KE), X=3 (nested panic) */

    uint32_t reboot_mode;


    uint32_t last_irq_enter[NR_CPUS];
    uint64_t jiffies_last_irq_enter[NR_CPUS];


    uint32_t last_irq_exit[NR_CPUS];
    uint64_t jiffies_last_irq_exit[NR_CPUS];


    uint64_t jiffies_last_sched[NR_CPUS];
    char last_sched_comm[NR_CPUS][TASK_COMM_LEN];


    uint8_t hotplug_data1[NR_CPUS], uint8_t hotplug_data2;
    uint64_t hotplug_data3;


    uint32_t mcdi_wfi, mcdi_r15, deepidle_data, sodi_data, spm_suspend_data;
    uint64_t cpu_dormant[NR_CPUS];
    uint32_t clk_data[8], suspend_debug_flag;


    uint8_t cpu_dvfs_vproc_big, cpu_dvfs_vproc_little, cpu_dvfs_oppidx, cpu_dvfs_status;


    uint8_t gpu_dvfs_vgpu, gpu_dvfs_oppidx, gpu_dvfs_status;


    uint64_t ptp_cpu_big_volt, ptp_cpu_little_volt, ptp_gpu_volt, ptp_temp;
    uint8_t ptp_status;
}

```

```
uint8_t thermal_temp1, thermal_temp2, thermal_temp3, thermal_temp4, thermal_temp5;

uint8_t thermal_status;


void *kparams;

};
```

以上重要的信息在重启后将被打包到 db 里的 **SYS_REBOOT_REASON** 文件里。对这只文件的各个栏位解读请查看：

- [MediaTek On-Line](#)> [Quick Start](#)> 深入分析看门狗框架> 分析方法> HW reboot> HW reboot 调试信息

五：前期异常处理

CPU 异常捕获

对于野指针、跑飞之类的异常会被 MMU 拦截并报告给 CPU，这一系列都是硬件行为，具体请看：

- [MediaTek On-Line](#)> [Quick Start](#)> 深入分析 Android native exception 框架> 流程-异常处理
- 在上面章节里的内核异常处理流程，有一处不同，走到 arm_notify_die() 后，判断是 kernel mode 就直接调用 die() 了，而不是 force_sig_info()

这类问题比较难定位，也是占 KE 比例的大头，原因通常是内存被踩坏、指针 use after free 等多种因素，在当时可能不会立即出现异常，而是到使用这块内存才有可能崩溃。

分析问题的手段也是多样化，比如用 watch point，MMU protect 或加 debug code 等（请参考附录 FAQ）

软件异常捕获

在 kernel 代码里，一般会通过 BUG(), BUG_ON(), panic() 来拦截超出预期的行为，这是软件主动回报异常的功能。

这些问题分析通常有固定的套路，请参考后面的：《实例篇：案例分析》

BUG()/BUG_ON() 实现

在内核调用可以用来方便标记 bug，提供断言并输出信息。最常用的两个是 BUG() 和 BUG_ON()。当被调用的时候，它们会引发 oops，导致栈的回溯和错误信息的打印。使用方式如下

```
if (condition)

    BUG();

或者：

BUG_ON(condition); //只是在 BUG 基础上多层封存而已；
```

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

32bit kernel:

BUG() 的实现采用了埋入未定义指令（**0xE7F001F2**，记住这个值，log 里看到这个值，你就应该知道是调用了 BUG()/BUG_ON() 了）的方式

```

/*
 * Use a suitable undefined instruction to use for ARM/Thumb2 bug handling.
 * We need to be careful not to conflict with those used by other modules and
 * the register_undef_hook() system.
 */

#define BUG_INSTR_VALUE 0xE7F001F2
#define BUG_INSTR_TYPE ".word "

#define BUG() BUG(__FILE__, __LINE__, BUG_INSTR_VALUE)
#define BUG(file, line, value) BUG(file, line, value)
#define _BUG(_file, _line, _value) do { asm volatile(BUG_INSTR_TYPE #_value); unreachable(); } while (0)

```

64bit kernel:

原生的 kernel, BUG() 是直接调用 panic() 的:

```

/*
 * Don't use BUG() or BUG_ON() unless there's really no way out; one
 * example might be detecting data structure corruption in the middle
 * of an operation that can't be backed out of. If the (sub)system
 * can somehow continue operating, perhaps with reduced functionality,
 * it's probably not BUG-worthy.
 *
 * If you're tempted to BUG(), think again: is completely giving up
 * really the *only* solution? There are usually better options, where
 * users don't need to reboot ASAP and can mostly shut down cleanly.
 */
#ifndef HAVE_ARCH_BUG
#define BUG() do { \
    printk("BUG: failure at %s:%d/%s()!\n", __FILE__, __LINE__, __func__); \
    panic("BUG!"); \
} while (0)
#endif

#ifndef HAVE_ARCH_BUG_ON
#define BUG_ON(condition) do { if (unlikely(condition)) BUG(); } while (0)
#endif

```

不过 Mediatek 修改了 BUG() 的实现, 这样有更多的调试信息输出 (die() 有寄存器等信息输出)

```

/*
 * Don't use BUG() or BUG_ON() unless there's really no way out; one
 * example might be detecting data structure corruption in the middle
 * of an operation that can't be backed out of. If the (sub)system
 * can somehow continue operating, perhaps with reduced functionality,
 * it's probably not BUG-worthy.
 *
 * If you're tempted to BUG(), think again: is completely giving up
 * really the *only* solution? There are usually better options, where
 * users don't need to reboot ASAP and can mostly shut down cleanly.
 */
#ifdef __arch64__
#define BUG() *{(unsigned *)0xdead} = 0x0aee
#define HAVE_ARCH_BUG
#endif
#ifndef HAVE_ARCH_BUG
#define BUG() do { \
    printk("BUG: failure at %s:%d/%s()!\n", __FILE__, __LINE__, __func__); \
    panic("BUG!"); \
} while (0)

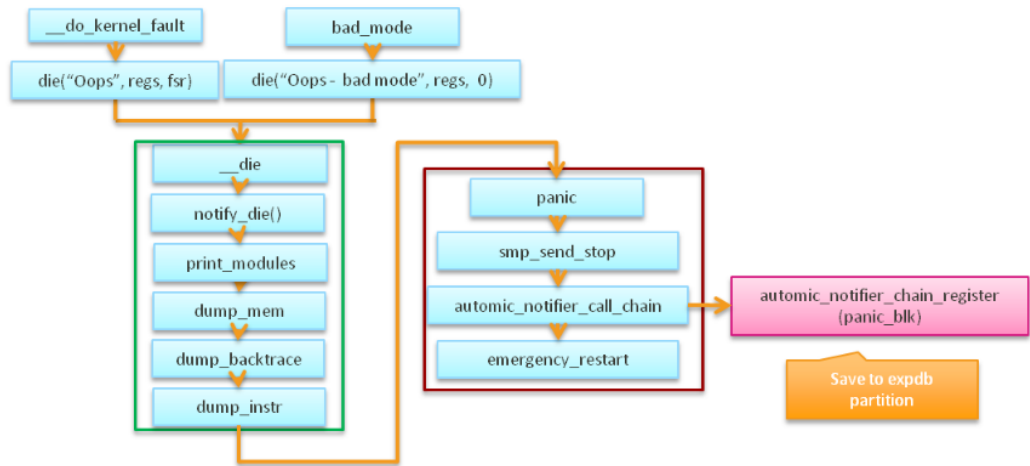
```

当你看到如下 log 时, 就应该知道是 BUG() / BUG_ON() 引起的了!

[147.234926]<0>-(0)[122:kworke/u8:3]Unable to handle kernel paging request at virtual address 0000dead

六: die() 流程

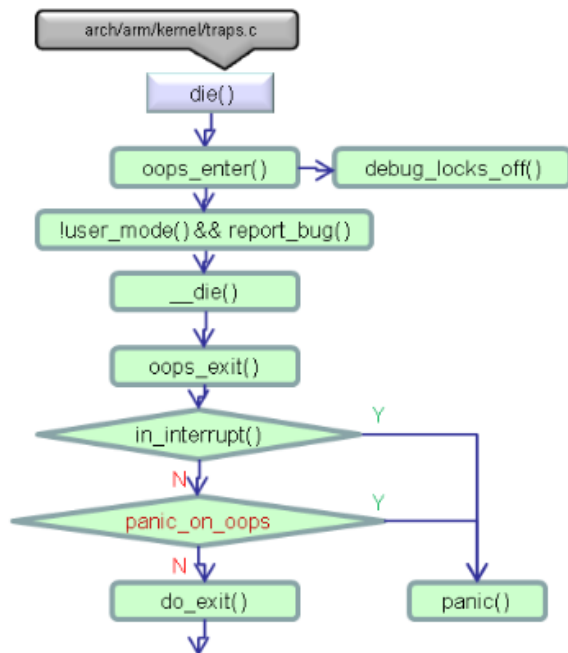
经过前面的流程，走到了 die() 函数，该函数主要输出便于调试的寄存器信息/堆栈信息等重要资料，我们通过 log 分析 KE 就是分析这些资料，因此要知道整个流程。die() => panic() 的大致流程如下：



在学习这些流程时，建议结合代码和 KE 的 log 一起看，你就知道 log 里那些信息在代码哪处打印出来的了。

die() 总流程

先从 die() 入手，看下 die() 总流程：



走到debug_locks_off()就有log输出了，如下：

```

ic inline int __debug_locks_off(void)
{
    int ret;
    ret = xchg(&debug_locks, 0);
    if (ret) {
        printk("[KERN Warning] Some Kernel ERROR or WARN occur and Force debug_lock off!\n");
        printk("[KERN Warning] check below backtrace first:\n");
        dump_stack();
    }
}

```

```

/u:1][KERN Warning] Some Kernel ERROR or WARN occur and Force debug_lock off!
/u:1][KERN Warning] check below backtrace first:
/u:1]Backtrace:
/u:1][<c0011e50>] (dump_backtrace+0x0/0x10c) from [<c0589ba0>] (dump_stack+0x18/0x1c)
/u:1] r6:ddb26000 r5:c06d5050 r4:00000000 r3:271ae91c
/u:1][<c0589b88>] (dump_stack+0x

```

如果这个异常是代码里调用 BUG()/BUG_ON() 引起，那么有额外 log 说明：

```

enum bug_trap_type report_bug(unsigned long bugaddr, struct pt_regs *regs)
{
    if (!is_valid_bugaddr(bugaddr))
        return BUG_TRAP_TYPE_NONE;
    printk(KERN_DEFAULT "-----[ cut here ]-----\n");
    printk(KERN_CRIT "Kernel BUG at %p [verbose debug info unavailable]\n", (void *)bugaddr);
}

```

如果是bug会
额外打印

如果这个异常是代码里调用 BUG()/BUG_ON() 引起，那么有额外 log 说明：

```

enum bug_trap_type report_bug(unsigned long bugaddr, struct pt_regs *regs)
{
    if (!is_valid_bugaddr(bugaddr))
        return BUG_TRAP_TYPE_NONE;
    printk(KERN_DEFAULT "-----[ cut here ]-----\n");
    printk(KERN_CRIT "Kernel BUG at %p [verbose debug info unavailable]\n", (void *)bugaddr);
}

```

如果是bug会
额外打印

输出的 log 大致如下：

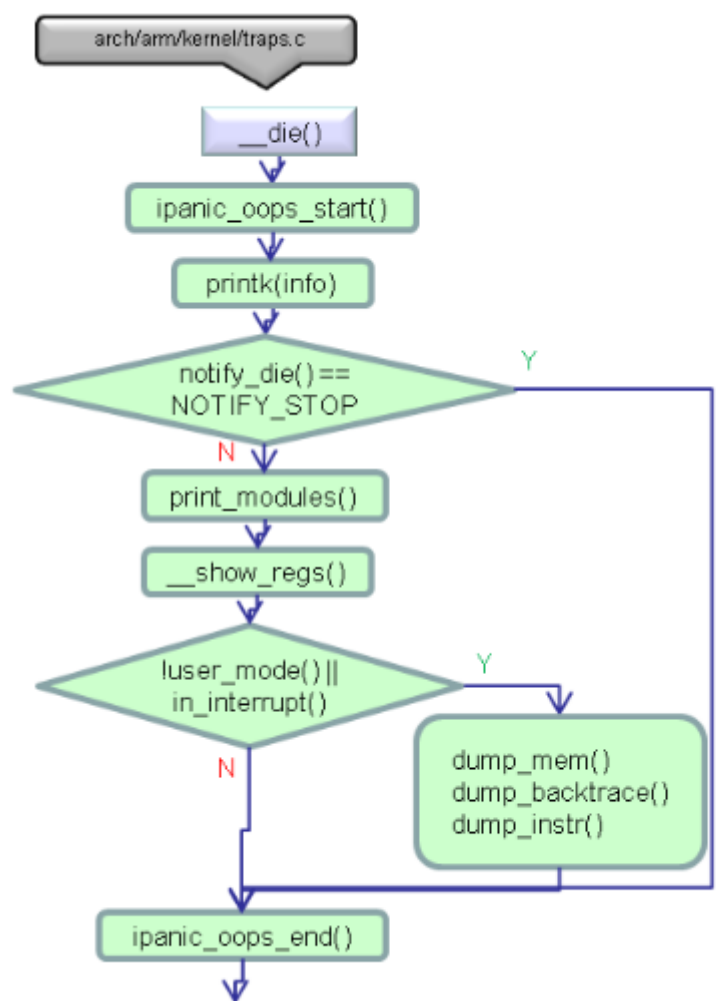
```

(O)-----[ cut here ]-----
(O)Kernel BUG at c03db5bc [verbose debug info unavailable]
(O)Internal error: Oops - BUG: 0 [#1] PREEMPT ARM
(O)Modules linked in: mtk_fm_drv bf13b000 (P) mtk_wmt_wifi bf137000 mtk_stp_bt bf12a000 mtk_stp_gps bf11d000 mtk_stp_uart bf114000 (P)

```

__die() 流程

绝大部分的关键信息是由__die() 函数输出的，流程如下：



异常类型信息

开始印出异常类型等信息，看一份 kernel log 有没有 oops，直接搜索关键字 Internal error 就可以了：

```
static int __die(const char *str, int err, struct thread_info *thread, struct pt_regs *regs)
{
    struct task_struct *tsk = thread->task;
    static int die_counter;
    int ret;

    ipanic_oops_start(); /* MTK */
    printk(KERN_EMERG "Internal error: %s: %x [%d] PREEMPT SMP ARM" "\n", str, err, ++die_counter);
}
```

输出的信息大致如下：

```
(0)-----[ cut here ]-----
(0)Kernel BUG at c03db5bc [verbose debug info unavailable]
(0)Internal error: Oops - BUG: 0 [#1] PREEMPT ARM
(0)Modules linked in: mtk_fm_drv bf13b000 (+) mtk_wmt_wifi bf137000 mtk_stp_bt bf12a000 mtk_stp_gps bf11d000 mtk_stp_uart bf114000 (P)
```

module 信息

接下来是 module 信息，不过我们不建议使用 module，这边也不打算介绍了。

CPU 寄存器信息

然后是重要的CPU寄存器信息(32bit的代码, 64bit类同):

```
void __show_regs(struct pt_regs *regs)
{
    unsigned long flags;
    char buf[64];

    printk("CPU: %d %s (%s %s)\n", raw_smp_processor_id(), print_tai
        (int)strosn(init_utsname()->version, " "), init_utsname()->versio
    print_symbol("PC is at %s\n", regs->ARM_pc);
    print_symbol("LR is at %s\n", regs->ARM_lr);
    printk("pc : [<%08lx>] lr : [<%08lx>] psr: %08lx\n"
        "sp : %08lx ip : %08lx fp : %08lx\n", regs->ARM_pc, regs->ARM
    printk("r10: %08lx r9 : %08lx r8 : %08lx\n", regs->ARM_r10, regs->ARM
    printk("r7 : %08lx r6 : %08lx r5 : %08lx r4 : %08lx\n", regs->ARM_r
    printk("r3 : %08lx r2 : %08lx r1 : %08lx r0 : %08lx\n", regs->ARM_r
    flags = regs->ARM_cpsr;
    buf[0] = flags & PSR_N_BIT ? 'N' : 'n';
    buf[1] = flags & PSR_Z_BIT ? 'Z' : 'z';
    buf[2] = flags & PSR_C_BIT ? 'C' : 'c';
    buf[3] = flags & PSR_V_BIT ? 'V' : 'v';
    buf[4] = '\0';
    printk("Flags: %s IRQs %s FIQs %s Mode %s ISA %s Segment %s\n",
        buf, interrupts_enabled(regs) ? "n" : "if",
        fast_interrupts_enabled(regs) ? "n" : "if",
        processor_modes[processor_mode(regs)],
        isa_modes[isa_mode(regs)],
        get_fs() == KERNEL_DS ? "kernel" : "user");
}
```

输出的信息大致如下:

在哪个CPU上

版本 (3.4.0 #1)

```
(0)CPU: 0 Tainted: P (3.4.0 #1)
(0)PC is at dumchar_open+0x1c0/0x20c
(0)LR is at console_unlock+0x184/0x354
(0)pc : [<c03db5bc>] lr : [<c005b964>] psr: 60000013
(0)sp : ddafd78 ip : c078c008 fp : ddafd78
(0)r10: deb3085c r9 : df1ee5e0 r8 : c06c7d08
(0)r7 : dda8ef40 r6 : df1ee584 r5 : ddad3b40 r4 : 00000000
(0)r3 : 56ad7a47 r2 : 56ad7a47 r1 : c078c008 r0 : 00000051
(0)Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user
(0)Control: 10c53c7d Table: 1e504059 D&C: 00000015
ARM/thumb/ThumbEE/Jazelle

从这里基本上可以看出死在哪个函数



kernel or user



CP15 控制寄存器1  
bit0: MMU开关  
bit2: cache开关  
bit3: write buffer开关  
bit7: 大小端配置  
bit12: I cache开关  
bit13: 高低异常向量配置  
bit23: 新的v6 MMU架构



MMU转换表


```

寄存器附近的内存

有助于我们分析问题的内存信息, 问题很可能就出在里面。

```

static void show_extra_register_data(struct pt_regs *regs, int nbytes)
{
    mm_segment_t fs;

    fs = get_fs();
    set_fs(KERNEL_DS);
    show_data(regs->ARM_pc - nbytes, nbytes * 2, "PC");
    show_data(regs->ARM_lr - nbytes, nbytes * 2, "LR");
    show_data(regs->ARM_sp - nbytes, nbytes * 2, "SP");
    show_data(regs->ARM_ip - nbytes, nbytes * 2, "IP");
    show_data(regs->ARM_fp - nbytes, nbytes * 2, "FP");
    show_data(regs->ARM_r0 - nbytes, nbytes * 2, "R0");
    show_data(regs->ARM_r1 - nbytes, nbytes * 2, "R1");
    show_data(regs->ARM_r2 - nbytes, nbytes * 2, "R2");
    show_data(regs->ARM_r3 - nbytes, nbytes * 2, "R3");
    show_data(regs->ARM_r4 - nbytes, nbytes * 2, "R4");
    show_data(regs->ARM_r5 - nbytes, nbytes * 2, "R5");
    show_data(regs->ARM_r6 - nbytes, nbytes * 2, "R6");
    show_data(regs->ARM_r7 - nbytes, nbytes * 2, "R7");
    show_data(regs->ARM_r8 - nbytes, nbytes * 2, "R8");
    show_data(regs->ARM_r9 - nbytes, nbytes * 2, "R9");
    show_data(regs->ARM_r10 - nbytes, nbytes * 2, "R10");
}

```

arch/arm/kernel/process.c

128字节

只打印kernel范围内的地址的数据，所以有可能只有一部分寄存器打印出来

输出的信息大致如下：

输出的信息大致如下：

```

(0) PC: 0xc03db53c:
(0) b53c ebf4df09 e5878000 e3e00004 eaffffca e1a03008 e1a01008 e1a02008 e59f0094
(0) b55c eb0662d3 e59f0090 eb0662d1 e3a00000 e1a01000 ebf0d9be e3e03000 e3e00015
(0) b57c e5873000 eaffffbc e595200c e1a03000 e58d0000 e1a01006 e59f0060 e592201c
(0) b59c eb0662c3 e59f0050 eb0662c1 e1a00004 e1a01004 ebf0d9ae e59f0044 eb0662bc
(0) b5bc e7f001f2 e59f003c eb0662b9 e3e0000b eaffffa9 c07a8c68 c0a41750 c06fcbb8
(0) b5dc c06fc8d8 c06e76a8 c06db09c c06fc95c c06fcaa8 c06fc974 c06fc9f8 c06fca68
(0) b5fc c06fc9f8 c06fc9d4 e1a0c00d e92dd8f0 e24cb004 e2504000 e1a06002
(0) b61c e1a07003 e59b500c 0a000020 e3560000 e2d71000 ba000029 e3550c02 8a000023
(0)
(0) LR: 0xc005b8e4:

```

调用栈

有时间可以直接从调用栈看出来，由此可见调用栈是多么重要。

```

static void dump_backtrace(struct pt_regs *regs, struct task_struct *tsk)
{
    unsigned int fp, mode;
    int ok = 1;

    void dump_backtrace_entry(unsigned long where /* 函数头地址 */, unsigned long from /* 返回地址LR */, unsigned long frame /* FP */)
    {
        printk("[<08lx>] [%pS] from [<08lx>] [%pS]\n", where, (void *)where, from, (void *)from);
        if (in_exception_text(where))
            dump_mem("", "Exception stack", frame + 4, frame + 4 + sizeof(struct pt_regs));
    }
}

```

arch/arm/kernel/traps.c

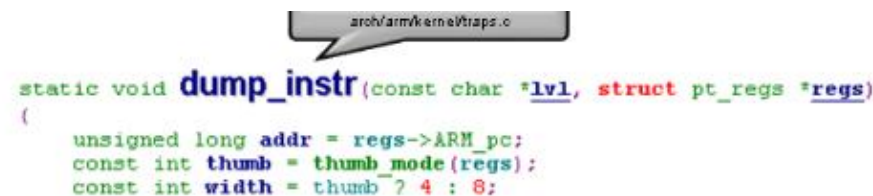
栈帧由该函数打印

输出的信息大致如下：



PC 附近指令

可以看到 PC 附近的指令：



输出的信息大致如下：



分析 log

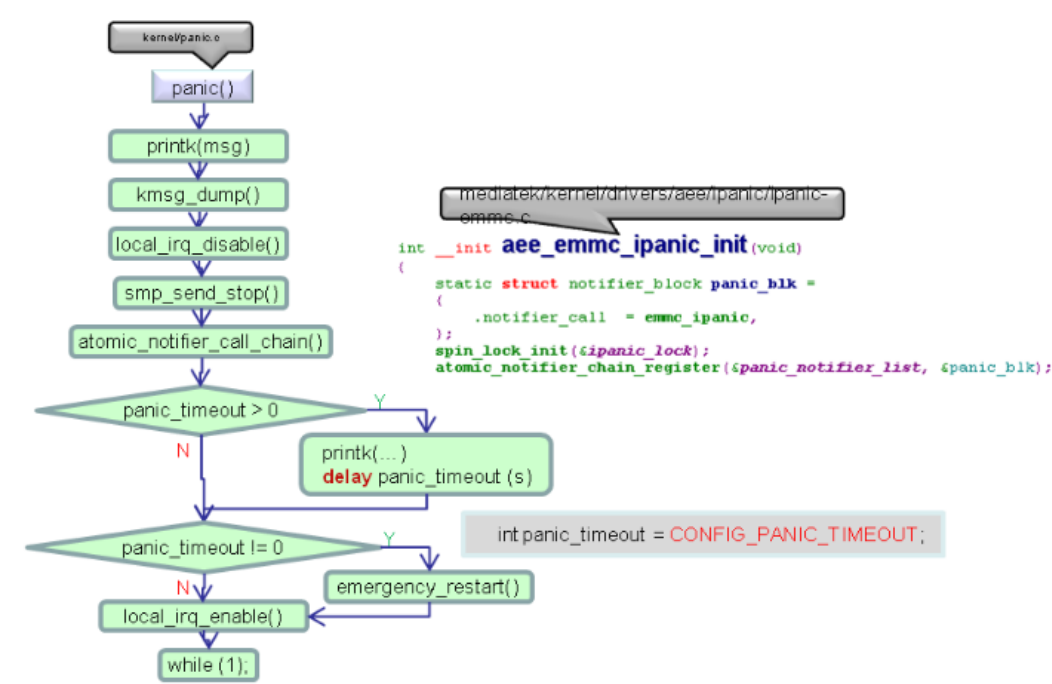
到这里 die() 函数就完成了它的使命，将重要信息输出出来了。接下来你要如何调试呢？这个就看个人的功力了，你可以：

- 通过 PC 指向的函数，用 addr2line（后面的 GNU tools 有介绍）定位到哪只文件的哪一行，大致可以知道发生了什么，如果无法一下子定位，也可以通过结合 printk() 多次观察 KE 时的 log 排查。如果是由 BUG()/BUG_ON() 引起的 KE，则就可以着手修复问题了。
- 查看调用栈，有些时候调用栈可以说明流程，看看代码是否有按预期跑，如果没有，可以结合 printk() 定位问题。
- 如果你想看函数参数或全局变量信息，那么你需要用《进阶篇：ramdump 分析》的知识调试了。

七：panic() 流程

流程走到 panic() 就里死（异常重启）不远了，关键的信息已输出到 kernel log。那么 panic() 做了什么呢？

panic() 流程



panic() 有标志性的 log 输出，大致如下：

```
<0>[ 1457.528644] (0)Kernel panic - not syncing: Fatal exception
```

因此我们也可以通过搜索关键字 Kernel panic 查找是否有 panic 发生。

panic 通知链

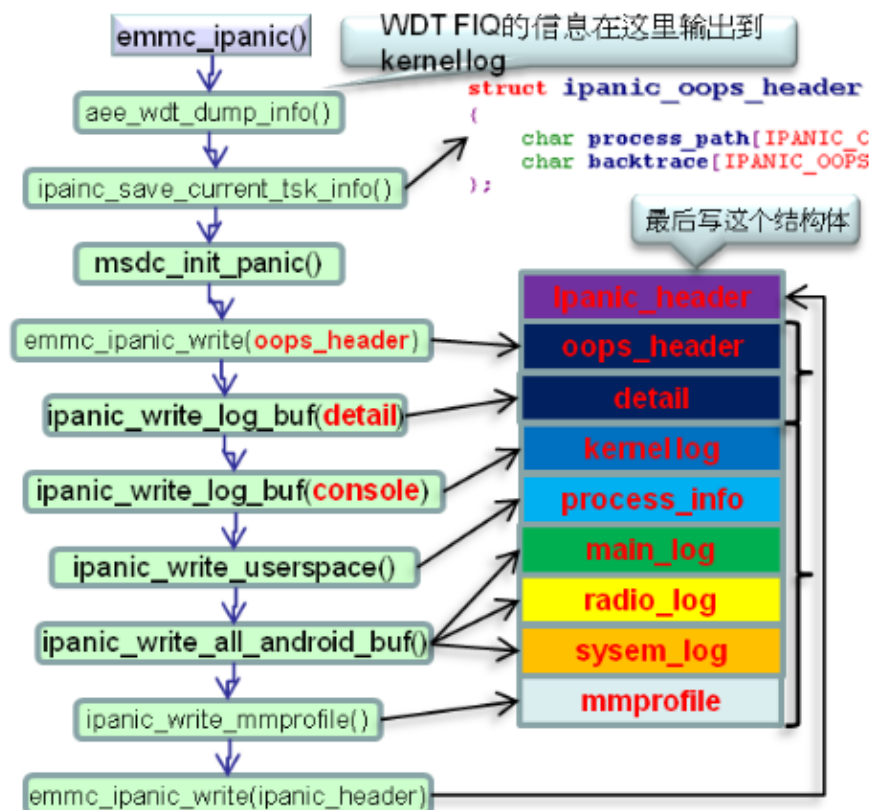
panic() 会调用栈通知链上的回调函数同时感兴趣的模块，比如我们的 aee 注册了回调函数，用于保存 kernel log/mini dump 等关键信息，并将其保存到 emmc 的 expdb 分区，等等重启后将其回读并保存成 KE db。

expdb

重启过程 DRAM 会丢失，因此信息只能保存在 flash 上了，在分区表里有一项就是 expdb 了：

13	MISC	Raw data	384 KB	384
14	LOGO	Raw data	3 MB	3072
15	EXPDB	Raw data	2 MB	2048
16	ANDROID	EXT4	513 MB	525312
17	CACHE	EXT4	513 MB	525312

流程大致如下（版本不停演进，可能有很大变化，仅供参考）：



重启后，aee 将回读 aeedb 分区资料并转化为 KE db。

八：nested panic

有时 die()/panic() 流程不一定能正常走完，可能走到某一步又发生了异常，则就形成了嵌套，这种情况，我们一般不会关注后面的异常，而是关注最开始的那个异常。

为了避免异常嵌套，在发生第 2 次异常时，我们就拦截下来，我们在 3 个地方用于拦截 nested panic：

- do_PrefetchAbort()
- do_DataAbort()
- do_undefinstr()

```

.inl void __exception do_DataAbort(unsigned l
struct thread_info *thread = current_thread_info();
.....

if (!user_mode(regs)) {
    thread->cpu_excp++;
    if (thread->cpu_excp == 1)
        thread->regs_on_excp = (void *)regs;
    /*
     * NoteXXX: The data abort exception may happen twice
     *           when calling probe_kernel_address()
     */
    if (thread->cpu_excp >= 3)
        aee_stop_nested_panic(regs);
}
  
```


拦截后不走 die()/panic() 流程，因为这些流程可能会再次发生异常，走我们写的函数 aee_stop_nested_panic() 函数：

```
static atomic_t nested_panic_time = ATOMIC_INIT(0);
void aee_stop_nested_panic(struct pt_regs *regs)
{
    struct thread_info *thread = current_thread_info();
    int i = 0;
    int len = 0;
    int timeout = 1000000;

    local_irq_disable();
    preempt_disable();
    /* Data abort handler data abort again on many cpu.
     Since ram console api is lockless, this should be prevent
    if (atomic_xchg(&nested_panic_time, 1) != 0) {
        aee_nested_printf("multicore enters nested pani
        goto ↓out;
    }
    mtk_wdt_restart(WK_WDT_LOC_TYPE_NOLOCK);
    hw_reboot_mode();

    /* must guarantee Only one cpu can run here */
    aee_nested_printf("Nested panic\n");
    aee_nested_printf("Log for the previous panic:\n");
```

在里面尽量少用 kernel 模块，很有可能也会发生异常，仅仅将寄存器等重要信息输出到 ram console 就等死（死循环等等看门狗复位!）。这时你抓回来的 db 里的 SYS_LAST_KMSG 就可以看到这些资料，大致如下（不同版本稍有区别）：

```
> 1.998988< Nested panic
> 1.998993< Log for the previous panic:
> 1.998995< pc: c0062f50 lr: c006310c psr: 600001d3
> 1.998999< sp: de52fe90 ip: 00000000 fp: de52ff2c
> 1.999003< r10: 00000041 r9: de52fed9 r8: de52e000
> 1.999007< r7: c085d15e r6: c07fbbbc r5: de52feca r4: c085d15e
> 1.999012< r3: 00000000 r2: c06ec048 r1: 00000032 r0: de52feca
> 1.999016< Log for the current panic:
> 1.999018< pc: c0593ddc lr: c0593f4c psr: 200001d3
> 1.999022< sp: de52fb20 ip: de52fb20 fp: de52fbbc
> 1.999026< r10: 00000041 r9: 00000000 r8: 00000204
> 1.999030< r7: 00000000 r6: de52fc90 r5: 00000005 r4: de52e000
> 1.999034< r3: 200001d3 r2: 00000003 r1: 00000000 r0: 00000002
stack (0xde52fe90 to 0xde530090)
fe90: 00000000 00000000 3ff79d00 00000000 600001d3 c085d1
feb0: 00000000 00000003 0000000f 00000000 00000003 205b00
fed0: 3839392e 5d363739 00000020 00000000 00000000 00000000
stack (0xde52fb20 to 0xde52fc20)
fb20: 00000000 00000000 00000000 00040000 00000000 00000000 020000
```

里面包含了寄存器信息、堆栈信息和调用栈，我们就可以通过工具（addr2line）还原当时异常的位置。

不过 nested panic 能参考的信息很少，不像普通的 KE 那样丰富。

进阶篇：ramdump 分析

一：dump 文件种类

为什么要用 ramdump?

虽然 debug 方法有多种，但是都有其局限性。拿 Logging 来讲，Logging 应该算是轻量级的 Debug 工具，并且在 linux kernel 默认就是打开的，在内核出现异常的时候我们可以很方便的得到下面这些信息：

- 出现问题前的 log，log 时间的长度取决于编译内核时配置的 log buffer 的大小。
- 问题发生时的 call stack，即当时的函数调用层次关系。

依靠这些信息，在大多数情况下我们都可以知道前面发生了什么，问题发生的大概原因是什么。但这里看到的也许并不是最根本的原因，因为它对某些情况的分析力度是有限的，比如：

- memory corruption，当有内存区域被踩坏的时候 logging 的信息通常是不够的，此类情况我们大多时候会多复现几次来从中找到规律。而复现问题本身对于低概率的问题是一个灾难。
- 硬件不稳定导致的问题。许多时候某个硬件器件不稳定导致的系统性问题都是难以通过硬件手段来定位的，而通过 logging 下来的信息也仅仅能看到一部分。尤其是硬件不稳定导致的 bit error 更是不易发现的小角落。
- Log 丢失的情况。在某些时候系统出现问题已经无法 logging 下来最后一段时间的系统行为，这也是 log 无能为力的地方。当然完善 log 机制可以得到改善，但是永远都无法在 logging 机制出现问题之前根除。

而这些 Logging 机制无法 cover 的问题，Memory dump 作为离线调试的另外一个手段则正好可以发挥其威力。

KB ramdump 文件种类

coredump 是 native 的概念，我们将它扩展到 kernel 层，在 db 里面有 2 只文件可供我们调试：

- **SYS_MINI_RDUMP**：该文件保存了发生异常的 CPU 寄存器信息及每个寄存器附近一段内存资料。包含的资料不多，所以文件也比较小，一般都是 2M 以内，这些基本足以用来调试。当然如果想看的资料超出该文件保存的内容时，就需要另外的方法了。
 - db 里基本上都有这只文件（KK 版本开始支持）。
- **SYS_COREDUMP**：该文件基本上将 DRAM 上的资料都保持下来了。里面有所有 kernel 管理的内存（排除了 security 内存/frame buffer），因此很大，比如 3G DRAM 的 SYS_COREDUMP 的大小接近 3G。
 - 有些项目默认是关闭的，详情请参考 ramdump 相关的 FAQ。

有了 dump 文件，要怎么调试呢？需要借助 debug 工具，以下表格列出了工具可以支持的文件。

工具	SYS_MINI_RDUMP	SYS_COREDUMP
gdb	支持	部分支持（看不到 task 等内核信息）
crash	不支持	支持
trace32	支持	支持

下面我们一一讲解这些工具的使用方法。

二：GNU tools

这里用到的工具和分析 NE 用到的工具没有差别，请参考：

- [MediaTek On-Line](#)> [Quick Start](#)> 深入分析 Android native exception 框架> 进阶篇：coredump 分析> GNU tools

三：AAPCS 标准

就是分析 NE 里提到的 AAPCS 标准，请参考：

- [MediaTek On-Line](#)> [Quick Start](#)> 深入分析 Android native exception 框架> 进阶篇：coredump 分析> AAPCS 标准

四：GDB 调试

这里用到的工具和分析 NE 用到的工具没有差别，请参考：

- [MediaTek On-Line](#)> [Quick Start](#)> 深入分析 Android native exception 框架> 进阶篇：coredump 分析> GDB 调试

使用方法

请查看：[【FAQ13941】](#) 如何分析 kernel panic？

用 gdb 调试 SYS_COREDUMP，是无法看到 kernel 信息，以及低于 0xC0000000（32 位）/ 0xFFFFF00000000000（64 位）的内存的。

五：crash 调试

crash 有专门的文章介绍，请参考：

- [MediaTek On-Line](#)> [Quick Start](#)> crash 使用教程

六：trace32 调试

Trace32 有专门的文章介绍，请参考：

- [MediaTek On-Line](#)> [Quick Start](#)> Trace32 使用教程
- 《Trace32 使用教程》里有 KE 分析章节，里面有讲解 KE cmm 脚本。

进阶篇：在线分析

一：JTAG 调试

不是所有问题用前面的调试手段都能解决，比如故障现场时外设寄存器信息，动态信息等是离线分析无法做到的。这时 jtag 就派上用场了，虽然用起来麻烦，但在项目初期还是比较实用的。

jtag 搭建/使用请到 DCC 上搜索查看文档：

- 《JTAG Debug User Guide.pptx》
- 《Debugger_User Guide_v5.1.docx》（版本可能会变化，比如 5.2 等）里的 Hardware Assistant Debug Tool (ICE) 章节

调试案例

[MediaTek On-Line](#)> [Quick Start](#)> [踩内存专题分析](#)> 1k 案例分析> 使能 SBC，进入 recovery 后 WDT 超时

二：KDB 调试

JTAG 必须要借助 debug 设备才行，用起来麻烦，有没有不用设备的在线调试方式呢？有！KDB 就是一种，已是 kernel 的部分了。

KDB 搭建/使用请到 DCC 上搜索查看文档：

- 《Debugger_User Guide_v5.1.docx》（版本可能会变化，比如 5.2 等）里的 KDB 章节

扩展篇：深入 Linux 内核

一：kernel 常用模块/结构

到这里，基本上对 KE 调试有基本的了解，剩下的就是对 kernel 的熟悉程度了。越熟悉，调试起来越容易，也可以根据问题对症下药。

kernel 内容非常庞大，可能不知道如何下手，建议先看 Unix/Linux 内核相关的书籍，了解内核的经典实现方法，然后再结合源码去研究 Linux 内核。这样做的原因是避免从一开始就陷入细节。

内核重点关注这几个部分：进程管理及调度，内存管理，文件及文件系统，Cache，I/O，SMP（多 CPU）。

参考的书籍有（最好是看英文原版）：

- 《Linux 内核设计与实现》
- 《Linux 内核源代码情景分析》
- 《深入理解 Linux 内核》

等等。

另外要注意，linux kernel 发展很快，有些模块/结构可能被移除或没有使用了，基本就不用关注了。

专题篇：专题分析

一：踩内存专题

踩内存问题一直都困扰着每个 kernel 工程师，调试难度很大，处于项目的各个阶段，严重影响软件品质。我们专门列一个专题分析，详情请看：

- MediaTek On-Line> Quick Start> 踩内存专题分析

二：fd 专题

fd 泄漏和意外关闭 fd 引起的逻辑问题在 kernel 比较少见，因为极其不建议在 kernel 打开文件，但不排除这种问题存在，详情请看：

- MediaTek On-Line> Quick Start> 文件描述符(fd)专题分析

三：售后收集重启专题

售后软件品质已越来越重要，以前通过返修/客退机来分析解决量产前未发现的问题，现在有更好的方法，通过后台自动收集故障信息，然后回传处理分析。我们专门列一个专题分析，详情请看：

- MediaTek On-Line> Quick Start> 售后收集重启专题分析

实例篇：案例分析

1. BUG at __schedule_bug

异常现场：

当你在 SYS_KERNEL_LOG 里看到如下 log，那么就属于 BUG at __schedule_bug 一类问题了

```
[14005.496163]<5>-(5)[1020:AudioOut_2]BUG: scheduling while atomic: AudioOut_2/1020/0x00000002
[14005.496257]<5>-(5)[1020:AudioOut_2]CPU: 5 PID: 1020 Comm: AudioOut_2 Tainted: G W 3.10.72+ #2
[14005.496265]<5>-(5)[1020:AudioOut_2]Call trace:
[14005.496279]<5>-(5)[1020:AudioOut_2]<ffffffc000088d2c> dump_backtrace+0x0/0x16c
[14005.496291]<5>-(5)[1020:AudioOut_2]<ffffffc000088ea8> show_stack+0x10/0x1c
[14005.496303]<5>-(5)[1020:AudioOut_2]<ffffffc000a7d56c> dump_stack+0x1c/0x28
[14005.496314]<5>-(5)[1020:AudioOut_2]<ffffffc000cd5c8> __schedule_bug+0x58/0x84
[14005.496327]<5>-(5)[1020:AudioOut_2]<ffffffc000a8db80> __schedule+0x6ac/0x858
[14005.496339]<5>-(5)[1020:AudioOut_2]<ffffffc000a8dd30> schedule+0x24/0x68
[14005.496351]<5>-(5)[1020:AudioOut_2]<ffffffc000a8b15c> schedule_timeout+0x134/0x218
[14005.496364]<5>-(5)[1020:AudioOut_2]<ffffffc000ac8c0> msleep+0x2c/0x40
[14005.496378]<5>-(5)[1020:AudioOut_2]<ffffffc0008bf224> RemoveMemifSubStream+0x4c/0x1c8
[14005.496390]<5>-(5)[1020:AudioOut_2]<ffffffc0008c9708> mtk_pcm_i2s0dl1_close+0xb4/0xf8
[14005.496404]<5>-(5)[1020:AudioOut_2]<ffffffc0008b1b68> soc_pcm_close+0x118/0x1d4
[14005.496418]<5>-(5)[1020:AudioOut_2]<ffffffc0008667bc> snd_pcm_release_substream.part.16+0x3c/0x90
[14005.496430]<5>-(5)[1020:AudioOut_2]<ffffffc000866a8> snd_pcm_release+0x98/0xc8
[14005.496444]<5>-(5)[1020:AudioOut_2]<ffffffc0001b2210> __fput+0x98/0x210
[14005.496455]<5>-(5)[1020:AudioOut_2]<ffffffc0001b2434> ____fput+0x8/0x14
[14005.496467]<5>-(5)[1020:AudioOut_2]<ffffffc0000bdc88> task_work_run+0x8c/0xe4
[14005.496478]<5>-(5)[1020:AudioOut_2]<ffffffc0000888c8> do_notify_resume+0x50/0x64
```

代码位置：

kernel-3.10/kernel/sched/core.c

```
static noinline void __schedule_bug(struct task_struct *prev)
{
    .....

    printk(KERN_ERR "BUG: scheduling while atomic: %s/%d/0x%08x\n",
        prev->comm, prev->pid, preempt_count());

    .....

    dump_stack();

    add_taint(TAINT_WARN, LOCKDEP_STILL_OK);

    BUG_ON(1); /* 这里发生 KE */
}
```

问题解读：

scheduling while atomic 的意思是：在原子上下文(关闭抢断)发起调度。

什么是原子上下文呢？不能被中断的上下文为原子上下文，比如：

- 已调用了 preempt_disable()（关闭抢断）后，就不能被调度的了，一般持有 spin lock 会关闭抢断。
- 在中断上下文里。linux 内核要求在中断处理的时候，不允许系统调度，不允许抢占，要等到中断处理完成才能做其他事情。
- 在软中断上下文里。

所以在原子上下文当然无法发起调度的了。在持有 spin lock 或在中断里，都是**要尽快做完后离开**。

如果长时间处于原子上下文，就容易出问题，引起系统响应缓慢、卡顿，甚至引起看门狗复位或死锁。

问题解决：

首先要看为什么发起调度的了，通常有 2 种原因：

- 在中断处理函数中或者拿到 spin lock 之后**调用了可能引起休眠的函数**，如 semaphore, mutex, sleep 之类的可休眠的函数。
- 拿了 spinlock 或 readlock 后忘记释放。

排查的方法：打开 CONFIG_DEBUG_PREEMPT，kernel 会记录最后一次关闭抢断的函数，通过分析该函数和该函数调用的子函数，然后通过代码逻辑排查，即可找到问题。

解决的方法：检查下代码逻辑，调整这些代码移除在原子上下文外面。

比如前面的例子，在 RemoveMemifSubStream() 函数里调用的 spin_lock_irqsave() 拿到了 spin lock，后面又调用了 msleep()，这就引起了这个问题。解决方法是，delay 还是要，所以将 msleep() 换成了 udelay() 就可以了。

案例分析：

- 《调用 kmalloc 前关闭抢断引起 KE》
- 《spin lock 没有 unlock 引起 KE》
- 《spinlock 没配对导致 KE》

2. BUG at __get_vm_area_node

当你在 SYS_KERNEL_LOG 里看到如下 log，那么就属于 BUG at __get_vm_area_node 一类问题了

```
{ 271.881362}<0~>0]Internal error: Oops: 96000045 [#1] PREEMPT SMP
[ 280.012666]<0~>0]PC is at __get_vm_area_node isra.28+0x1d4/0x1ec
[ 280.012672]<0~>0]LR is at get_vm_area_caller+0x2c/0x34
[ 280.014883]<0~>0]Call trace:
[ 280.014891]<0~>0]<ffffffc00017df54>] __get_vm_area_node isra.28+0x1d4/0x1ec
[ 280.014898]<0~>0]<ffffffc00017ea48>] get_vm_area_caller+0x28/0x34
[ 280.014909]<0~>0]<ffffffc0000948c4>] __ioremap+0x64/0xe8
[ 280.014920]<0~>0]<ffffffc0006e0f2c>] hal_dma_dump_reg+0xf0/0xa0c
[ 280.014932]<0~>0]<ffffffc0006d934c>] btif_bbs_write.constprop.24+0x2b0/0x2dc
[ 280.014943]<0~>0]<ffffffc0006db1f8>] btif_dma_rx_data_receiver+0x1c/0x4c
[ 280.014950]<0~>0]<ffffffc0006e0cf8>] hal_rx_dma_irq_handler+0x170/0x2b4
[ 280.014959]<0~>0]<ffffffc0006d7c6c>] btif_rx_dma_irq_handler+0x64/0xec
[ 280.014970]<0~>0]<ffffffc00011f08>] handle_irq_event_percpu+0x98/0x3c8
[ 280.014978]<0~>0]<ffffffc000120280>] handle_irq_event+0x48/0x78
[ 280.014987]<0~>0]<ffffffc0001203c4>] handle_fastcall_irq+0xb0/0x150
[ 280.014996]<0~>0]<ffffffc000117784>] generic_handle_irq+0x30/0x4c
[ 280.015005]<0~>0]<ffffffc0000848e4>] handle_IRQ+0x94/0x1cc
[ 280.015012]<0~>0]<ffffffc000081608>] gic_handle_irq+0x3c/0x80
```

代码位置：

kernel-3.18/mm/vmalloc.c

```
static struct vm_struct * __get_vm_area_node(unsigned long size,
      unsigned long align, unsigned long flags, unsigned long start,
      unsigned long end, int node, gfp_t gfp_mask, const void *caller)
{
    struct vmmap_area *va;

    struct vm_struct *area;

    BUG_ON(in_interrupt());

    .....
}
```

问题解读：

BUG_ON 放在这里意思是在不允许在中断里调用 __get_vm_area_node() 函数。

为什么呢？应该是设计时就不允许在中断里使用的，该函数使用了全局变量，用 spin_lock 保护，这种锁无法保护到中断上下文。如果在中断里使用 __get_vm_area_node() 可能会引起一系列问题。

哪些常用函数会调用到 __get_vm_area_node() 呢？

- vmalloc() 系列函数
- ioremap() 系列函数，包括 of_iomap() 函数

也即是说，这些函数禁止在中断里调用！

问题解决：

如果是在中断里调用 ioremap() 函数则可以考虑在初始化时就 map 好。比如上面的例子，就应该在初始化时就 map 好，在中断里直接用就行了。

如果是 vmalloc()，可以考虑用 kmalloc() 替换。

3. BUG at dpm_wd_handler

异常现场：

在__exp_main.txt 和 SYS_KERNEL_LOG 里看到如下 log:

死机位置在函数 dpm_wd_handler() 中，是这里的 BUG() 触发了 Exception

```
<4>[ 341.525222] 2)PC is at dpm_wd_handler+0x2c/0x30
<7>[ 328.440667] 2)[57] bus device_suspend
<7>[ 328.440674] 2)dev->driver->name=mtk-msdc
<3>[ 328.440714] 2)msdc1 -> PM Suspend
<4>[ 328.440741] 2)msdc1 select card<0x00000000>
<4>[ 328.440776] 2)msdc LDO<2> power off
<4>[ 328.440796] 2)msdc LDO<3> power off
<3>[ 328.450820] 2)msdc1 -> set mclk to 0
<7>[ 328.451843] 2)[58] bus device_suspend
<7>[ 328.451850] 2)dev->driver->name=mtk-msdc
<3>[ 328.451863] 2)msdc0 -> PM Suspend
<4>[ 328.451880] 2)msdc0 select card<0x00000000>
<4>[ 328.451933] 2)msdc LDO<4> power off
<7>[ 328.451973] 2)[59] bus device_suspend
<7>[ 328.451988] 2)[60] bus device_suspend
<7>[ 328.452001] 2)[61] bus device_suspend
<7>[ 328.452008] 2)dev->driver->name=akm8963
* * * * *
<0>[ 340.489412] 2)akm8963 2-000c: **** DPM device timeout ****
<4>[ 340.489421] 2)Backtrace:
* * * * *
```

代码位置:

kernel-3.10/drivers/base/power/main.c

```
/**
411 * dpm_wd_handler - Driver suspend / resume watchdog handler.
412 *
413 * Called when a driver has timed out suspending or resuming.
414 * There's not much we can do here to recover so BUG() out for
415 * a crash-dump
416 */
417 static void dpm_wd_handler(unsigned long data)
418 {
419     struct dpm_watchdog *wd = (void *)data;
420     struct device *dev = wd->dev;
421     struct task_struct *tsk = wd->tsk;
422
423     dev_emerg(dev, "**** DPM device timeout ****\n");
424     show_stack(tsk, NULL);
425
426     BUG();
427 }
```

问题分析:

系统在 call 每个 Module 驱动的 suspend/resume callback function 的时候, 会设定一个 timer 来监控回调函数的执行; 如果回调函数长时间没有执行完毕, 定时器函数会调用 BUG() 让系统挂掉,

所以在客制化回调函数时, 要确保不能长时间阻塞, 否则就会走到 dpm_wdt_handler() 函数, 如上 log, 先打出“**** DPM device timeout ****”, 接着会将相关驱动的 **Backtrace** 印出来

程序如何跑入 dpm_wdt_handler() ?

在 __device_suspend() 函数中, 可以看到在函数开始位置会调用 dpm_wd_set(&wd, dev), 此处会设置一个 timer, 在 suspend 函数结尾处, 会调用 dpm_wd_clear(&wd), 用于清除 timer, 在 start timer 和 clear timer 之间的 code 是做 device suspend 的动作, 这个过程必须在一定时间内完成, 否则就会导致 timer 到期, 触发 dpm_wdt_handler()

问题解决:

第一, 根据上述 log 中 Backtrace 找到是哪个 device 导致, 如是自己添加的 device 驱动, 特别是客制化的部分, 需自己查看一下是否存在 device suspend 太久的情况。

第二, 如这部分 code 没有做任何修改, 请提交 e-service 并注明是哪部分驱动引起的问题, 以便我们迅速处理问题。

4. BUG at dpm_drv_timeout

dpm_drv_timeout() 和 dpm_wd_handler() 一样, 请参考《BUG at dpm_wd_handler》章节

5. BUG at check_bytes_and_report

异常现场:

当你在 SYS_KERNEL_LOG 里看到如下 log, 那么就属于 BUG at check_bytes_and_report 一类问题了:

```
<6>[ 492.558572]- (0) [1163:system_server]=====
<6>[ 492.558599]- (0) [1163:system_server]BUG kmalloc-128 (Tainted: P W O ): Poison overwritten
<6>[ 492.558621]- (0) [1163:system_server]-----
<6>[ 492.558621]
<6>[ 492.558649]- (0) [1163:system_server]INFO: 0xffffffc0553a93e9-0xffffffc0553a93e9. First byte 0x69 instead of 0x6b
<6>[ 492.558685]- (0) [1163:system_server]INFO: Allocated in alloc_vmap_area.isra.31+0x6c/0x394 age=525 cpu=0 pid=8631
<6>[ 492.558708]- (0) [1163:system_server] alloc_debug_processing+0x184/0x194
<6>[ 492.558731]- (0) [1163:system_server] __slab_alloc.isra.55.constprop.64+0x5ac/0x5e8
<6>[ 492.558751]- (0) [1163:system_server] kmem_cache_alloc_trace+0x11c/0x240
<6>[ 492.558772]- (0) [1163:system_server] alloc_vmap_area.isra.31+0x68/0x394
<6>[ 492.558793]- (0) [1163:system_server] __get_vm_area_node.isra.32+0x98/0x190
<6>[ 492.558814]- (0) [1163:system_server] __vmalloc_node_range+0x64/0x284
<6>[ 492.558833]- (0) [1163:system_server] vmalloc+0x2c/0x38
<6>[ 492.558854]- (0) [1163:system_server] write_pmsg+0x58/0x148
<6>[ 492.558881]- (0) [1163:system_server]INFO: Freed in rcu_process_callbacks+0x230/0x988 age=179 cpu=0 pid=3
<6>[ 492.558902]- (0) [1163:system_server] free_debug_processing+0x1f4/0x328
<6>[ 492.558921]- (0) [1163:system_server] __slab_free+0x29c/0x374
<6>[ 492.558939]- (0) [1163:system_server] kfree+0x22c/0x270
<6>[ 492.558959]- (0) [1163:system_server] rcu_process_callbacks+0x22c/0x988
```



```
<6>[ 492.558979]- (0) [1163:system_server] __do_softirq+0xd8/0x36c

<6>[ 492.558998]- (0) [1163:system_server] run_ksoftirqd+0x6c/0xec

<6>[ 492.559020]- (0) [1163:system_server] smpboot_thread_fn+0x1fc/0x2c8

<6>[ 492.559039]- (0) [1163:system_server] kthread+0xd8/0xf0

<6>[ 492.559063]- (0) [1163:system_server] INFO: Slab 0xffffffffe020a4cf8 objects=12 used=12 fp=0x (null) flags=0x0080

<6>[ 492.559085]- (0) [1163:system_server] INFO: Object 0xffffffffc0553a93c0 @offset=960 fp=0xffffffffc0553a9640

<6>[ 492.559085]

<6>[ 492.559116]- (0) [1163:system_server] Bytes b4 ffffffff0553a93b0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ

<6>[ 492.559140]- (0) [1163:system_server] Object ffffffff0553a93c0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk

<6>[ 492.559165]- (0) [1163:system_server] Object ffffffff0553a93d0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk

<6>[ 492.559189]- (0) [1163:system_server] Object ffffffff0553a93e0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk

<6>[ 492.559213]- (0) [1163:system_server] Object ffffffff0553a93f0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk

<6>[ 492.559237]- (0) [1163:system_server] Object ffffffff0553a9400: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk

<6>[ 492.559261]- (0) [1163:system_server] Object ffffffff0553a9410: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk

<6>[ 492.559286]- (0) [1163:system_server] Object ffffffff0553a9420: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk

<6>[ 492.559310]- (0) [1163:system_server] Object ffffffff0553a9430: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5 kkkkkkkkkkkkkkkk.

<6>[ 492.559333]- (0) [1163:system_server] Redzone ffffffff0553a9440: bb bb bb bb bb bb bb bb .....

<6>[ 492.559358]- (0) [1163:system_server] Padding ffffffff0553a94c0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ

<6>[ 492.559382]- (0) [1163:system_server] Padding ffffffff0553a94d0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ

<6>[ 492.559407]- (0) [1163:system_server] Padding ffffffff0553a94e0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ

<6>[ 492.559431]- (0) [1163:system_server] Padding ffffffff0553a94f0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a ZZZZZZZZZZZZZZZZ

<6>[ 492.559457]- (0) [1163:system_server] CPU: 0 PID: 1163 Comm: system_server Tainted: P B W O 3.18.22+ #1

<6>[ 492.559474]- (0) [1163:system_server] Hardware name: MT6755 (DT)

<6>[ 492.559491]- (0) [1163:system_server] Call trace:

<2>[ 492.559514]- (0) [1163:system_server] [<ffffffc00008b970>] dump_backtrace+0x0/0x15c

<2>[ 492.559534]- (0) [1163:system_server] [<ffffffc00008badc>] show_stack+0x10/0x1c

<2>[ 492.559557]- (0) [1163:system_server] [<ffffffc000abb050>] dump_stack+0x74/0xb8

<2>[ 492.559578]- (0) [1163:system_server] [<ffffffc0001c44fc>] print_trailer+0x140/0x288

<2>[ 492.559600]- (0) [1163:system_server] [<ffffffc0001c471c>] check_bytes_and_report+0xd8/0x10c

<2>[ 492.559622]- (0) [1163:system_server] [<ffffffc0001c4968>] check_object+0x1ac/0x220

<2>[ 492.559644]- (0) [1163:system_server] [<ffffffc0001c69a8>] alloc_debug_processing+0x110/0x194

<2>[ 492.559667]- (0) [1163:system_server] [<ffffffc0001c7330>] __slab_alloc.isra.55.constprop.64+0x5ac/0x5e8

<2>[ 492.559689]- (0) [1163:system_server] [<ffffffc0001c7958>] kmem_cache_alloc_trace+0x11c/0x240

<2>[ 492.559711]- (0) [1163:system_server] [<ffffffc0000c1eac>] __async_schedule+0x34/0x1a8

<2>[ 492.559734]- (0) [1163:system_server] [<ffffffc0000c2048>] async_schedule_domain+0x8/0x14

<2>[ 492.559759]- (0) [1163:system_server] [<ffffffc0008da144>] dapm_power_widgets+0x618/0x8bc

<2>[ 492.559783]- (0) [1163:system_server] [<ffffffc0008dc17c>] snd_soc_dapm_stream_event+0xbc/0x110

<2>[ 492.559805]- (0) [1163:system_server] [<ffffffc0008d04f8>] snd_soc_suspend+0x294/0x468

<2>[ 492.559829]- (0) [1163:system_server] [<ffffffc000393e18>] platform_pm_suspend+0x20/0x50

<2>[ 492.559852]- (0) [1163:system_server] [<ffffffc000399194>] dpm_run_callback+0x50/0x184

<2>[ 492.559873]- (0) [1163:system_server] [<ffffffc00039ae98>] __device_suspend+0x128/0x3cc

<2>[ 492.559894]- (0) [1163:system_server] [<ffffffc00039b1e0>] dpm_suspend+0xa4/0x358

<2>[ 492.559915]- (0) [1163:system_server] [<ffffffc00039b4fc>] dpm_suspend_start+0x68/0x78

<2>[ 492.559939]- (0) [1163:system_server] [<ffffffc0000fe8d8>] suspend_devices_and_enter+0x98/0x2d0
```

```

<2>[ 492.559960]- (0) [1163:system_server][<ffffffc000fefe4>] pm_suspend+0x3c4/0x5c8
<2>[ 492.559982]- (0) [1163:system_server][<ffffffc000fd5b0>] state_store+0xb0/0xe0
<2>[ 492.560004]- (0) [1163:system_server][<ffffffc0003270c4>] kobj_attr_store+0x10/0x24
<2>[ 492.560026]- (0) [1163:system_server][<ffffffc00023996c>] sysfs_kf_write+0x3c/0x48
<2>[ 492.560049]- (0) [1163:system_server][<ffffffc000238ef4>] kernfs_fop_write+0x10c/0x178
<2>[ 492.560070]- (0) [1163:system_server][<ffffffc0001d0808>] vfs_write+0x98/0x1b8
<2>[ 492.560091]- (0) [1163:system_server][<ffffffc0001d0f40>] SyS_write+0x40/0xa0
<2>[ 492.560124]- (0) [1163:system_server]BUG: failure at kernel-3.18/mm/slub.c:796/check_bytes_and_report()!
<6>[ 492.561176]- (0) [1163:system_server][klog]fault_level=0x7(data abort(aarch64)), fault_type=6, fault_msg=level 2 translation
fault
<6>[ 492.561201]- (0) [1163:system_server]Unable to handle kernel paging request at virtual address 0000dead
<6>[ 492.561220]- (0) [1163:system_server]pgd = fffffffc03c136000
<2>[ 492.561229]- (0) [1163:system_server][0000dead] *pgd=000000000802de003, *pud=000000000802de003, *pmd=0000000000000000
<6>[ 492.561283]- (0) [1163:system_server][klog]fault already exists:0x7, ignore:0x2
<6>[ 492.561306]- (0) [1163:system_server]Internal error: Oops: 96000046 [#1] PREEMPT SMP
<2>[ 492.561316]- (0) [1163:system_server]disable aee kernel api
<6>[ 492.561350]- (0) [1163:system_server]CPU: 0 PID: 1163 Comm: system_server Tainted: P B W O 3.18.22+ #1
<6>[ 492.561369]- (0) [1163:system_server]Hardware name: MT6755 (DT)
<6>[ 492.561395]- (0) [1163:system_server]task: fffffffc03b158000 ti: fffffffc02f018000 task.ti: fffffffc02f018000
<6>[ 492.561418]- (0) [1163:system_server]PC is at check_bytes_and_report+0x108/0x10c
<6>[ 492.561441]- (0) [1163:system_server]LR is at check_bytes_and_report+0x100/0x10c

```

代码位置:

kernel-3.18/mm/slub.c

```

static int check_bytes_and_report(struct kmem_cache *s, struct page *page, u8 *object, char *what, u8 *start, unsigned int value, unsigned int bytes)
{
    u8 *fault;
    u8 *end;

    fault = memchr_inv(start, value, bytes);
    if (!fault)
        return 1;

    end = start + bytes;
    while (end > fault && end[-1] == value)
        end--;

    slab_bug(s, "%s overwritten", what);
    pr_err("INFO: 0x%p-0x%p. First byte 0x%x instead of 0x%x\n", fault, end - 1, fault[0], value);
    print_trailer(s, page, object);

    BUG(); /* 这里发生 KE */
}

```

```

    restore_bytes(s, what, value, fault, end);

    return 0;
}

```

slub 是 kernel 非常重要的内存分配器（详情请看：doc/Documentation/vm/slub.txt），是基于 buddy system 分配器之上再细化的。其接口就是 `kmalloc()`/`kfree()` 等一系列函数，在 kernel 里被广泛使用。

任何内存管理相关的问题，slub 也会有。比如：

- use after free。通过 `kmalloc()` 申请内存，用完之后 `kfree()` 释放，但是后面又再使用了这块释放的内存。
- 内存踩坏。
 - 通过 `kmalloc()` 申请的内存，使用时超出了当时申请的尺寸，将不属于你的内存踩坏了。
 - 其他模块意外将 slub 空闲/使用中的内存踩坏。
- double free。对一块 `kmalloc()` 申请的内存连续调用 2 次 `kfree()` 函数。
- HW 故障。比如 DRAM/CPU 不稳定，导致原有的内存发生跳变，bitflip。上面的例子就是发生了 bitflip，6b 跳变为 69 了。

如何检查 slub 里是否发生异常呢？这就需要额外的内存做守卫了，kernel 已有这样的功能，只要打开 `CONFIG_SLUB_DEBUG`，`kfree()` 后的内存会被格式化成 6b，还有 red zone 格式化成 bb，padding 为 5a，还有每次申请和释放都会记录调用栈，记录谁申请/释放的。在每次的内存申请都会做相关检查，如果出现故障，就会抛出对应的 log 并主动调用 `BUG()`

问题解决：

- use after free。需要通过 slub 记录的调用栈找凶手，检查代码逻辑才行。
- 内存踩坏。通过 slub 记录的调用栈或采用 MMU 保护，请参考：[FAQ14614]如何用 MMU 保护 buddy system?
- double free。从 slub 记录的调用栈可以很明显看出。
- bitflip。直接要硬件交叉比对（CPU/DRAM），排查硬件故障，上面的 log 的例子直接 HW 交叉比对。

6. BUG at `_i2c_deal_result`

异常现场：

在 `__exp_main.txt` 和 `SYS_KERNEL_LOG` 里看到如下 log：

死机位置在函数 `dpm_wd_handler()` 中，是这里的 `BUG()` 触发了 Exception

```

[ 21.758078]-(1)[155:bat_thread_kthr]Kernel BUG at c0660080 [verbose debug info unavailable]
[ 21.758092]-(1)[155:bat_thread_kthr]Internal error: Oops - BUG: 0 [#1] PREEMPT SMP ARM
[ 29.114643]-(1)[155:bat_thread_kthr]Modules linked in:
[ 29.114666]-(1)[155:bat_thread_kthr]CPU: 1 PID: 155 Comm: bat_thread_kthr Tainted: G W 3.10.48+ #1
[ 29.114679]-(1)[155:bat_thread_kthr]task: df2ce000 ti: df810000 task.ti: df810000
[ 29.114692]-(1)[155:bat_thread_kthr]PC is at _i2c_deal_result+0x314/0x358
[ 29.114705]-(1)[155:bat_thread_kthr]LR is at i2c_release_md32_semaphore+0x40/0x9c

```

代码位置：

`kernel-3.18/drivers/misc/mediatek/i2c/$platform/i2c.c`

```

static s32 _i2c_deal_result(struct mt_i2c_t *i2c)
{
    .....

    if (!(tmo == 0 || atomic_read(&i2c->trans_err))) {

        /*Transfer success ,we need to get data from fifo */

        if ((!(i2c->dma_en) && (i2c->op == I2C_MASTER_RD || i2c->op == I2C_MASTER_WRRD)) {

            data_size = (i2c_readl(i2c, OFFSET_FIFO_STAT) >> 4) & 0x000F;

            BUG_ON(data_size > i2c->msg_len);

            /* I2CLOG("data_size=%d\n",data_size); */

            while (data_size--) {

                .....

            }

            .....

        }

        .....

    }

}

```

问题分析:

出现这种问题的原因是 I2C 模块接收到的数据超出驱动的预期（多接收）。

原因基本上是硬件故障，比如受到干扰，共用 I2C 引起的冲突等等。

问题解决:

检查驱动对应的 I2C 是否挂载了其他设备，如果可以先拔除其他共用的 I2C 设备理清问题，然后再从驱动配置和硬件电路入手排查。

7. BUG at swiotlb_full

异常现场:

在 KE 的 DB 里面我们可以明确的看到:

Exception Class: Kernel (KE)

PC is at [<ffffffc000332224>] **swiotlb_full**+0x64/0xccc

Current Executing Process:

[tx_thread, 1476][kthreadd, 2]

Backtrace:

[<ffffffc000a7d418>] __do_kernel_fault.part.5+0x70/0x84

[<ffffffc000095208>] do_page_fault+0x218/0x364

[<ffffffc000095440>] do_translation_fault+0x40/0x4c

[<ffffffc000081240>] do_mem_abort+0x38/0x9c

[<ffffffc000083c58>] ell_da+0x1c/0x88

[<ffffffc000332ff4>] swiotlb_map_page+0x150/0x15c

[<ffffffc000094aec>] __swiotlb_map_page+0x18/0x54

[<ffffffc0005d9e6c>] kalDevPortWrite+0x200/0x71c

[<ffffffc00059b898>] nicTxMsduQueue+0x2fc/0x7b4

```
[<ffffffc00059bf48>] nicTxMsduInfoList+0x1f8/0x300
[<ffffffc000577758>] wlanTxPendingPackets+0x7c/0x168
[<ffffffc0005b4ef8>] tx_thread+0x3b0/0x678
[<ffffffc0000bfd5c>] kthread+0xb0/0xbc
[<ffffffc00008446c>] ret_from_fork+0xc/0x20
[<ffffffffffffffff>] 0xffffffffffffffff
```

以及在 sys_kernel_log:

```
[95902.825452] (2)[27392:tx_thread]DMA: Out of SW-IOMMU space for 24192 bytes at device 180f0000.wifi
[95902.825472] (2)[27392:tx_thread]BUG: failure at
/work/15114/DailyBuild_Guide_30208_201604260056/15114_USR_201604260056/kernel-3.10/lib/swiotlb.c:707/swiotlb_full()!
[95902.825488] (2)[27392:tx_thread]Unable to handle kernel paging request at virtual address 0000dead
[95902.825496] (2)[27392:tx_thread]pgd = fffffffc00007d000
[95902.545793] (2)[27392:tx_thread]AEE_MONITOR_SET[status]: 0x1
```

此类 case 我们一般描述为 swiotlb full or leaks.

原理分析与可能的原因

swiotlb 全称是:software io translate buffer, 用于 DMA 操作. 对应的代码在 kernel-3.10/lib/swiotlb.c.

在 MTK 平台上, 如果没有配置 low memory 的 GMO, 默认使用巨量的 64M swiotlb.

```
/* default to 64MB */
#ifdef CONFIG_MTK_LM_MODE
#define IO_TLB_DEFAULT_SIZE (SZ_64M)
#else
#define IO_TLB_DEFAULT_SIZE ((1 << IO_TLB_SHIFT) * IO_TLB_SEGSIZE)
#endif // end of CONFIG_MTK_LM_MODE
```

在 arch init 时即完成初始化.

```
void __init
swiotlb_init(int verbose)
{
    size_t default_size = IO_TLB_DEFAULT_SIZE;
    unsigned char *vstart = 0;
    unsigned long bytes;
    phys_addr_t start;

    if (!io_tlb_nslabs) {
        io_tlb_nslabs = (default_size >> IO_TLB_SHIFT);
        io_tlb_nslabs = ALIGN(io_tlb_nslabs, IO_TLB_SEGSIZE);
    }

    bytes = io_tlb_nslabs << IO_TLB_SHIFT;
```

```

/* Get IO TLB memory from the low pages */
memblock_set_current_limit(0xffffffff); /* 4GB */

start = memblock_alloc(PAGE_ALIGN(bytes), PAGE_SIZE);

if (start) {
    vstart = __va(start);
} else {
    pr_err("iotlb allocation fail\n");
}

memblock_set_current_limit(MEMBLOCK_ALLOC_ANYWHERE);

if (vstart && !swiotlb_init_with_tbl(vstart, io_tlb_nslabs, verbose))

return;

if (io_tlb_start)
memblock_free(io_tlb_start,
PAGE_ALIGN(io_tlb_nslabs << IO_TLB_SHIFT));

pr_warn("Cannot allocate SWIOTLB buffer");

no_iotlb_memory = true;
}

```

它的申请分配与释放最终通过：

swiotlb_tlb_map_single/swiotlb_tlb_unmap_single

来达成。

管理方式很简单，针对这 64M 的 buffer，分成 blocks，每个 block 的大小是 256KB，最小可分配的大小是半个 page：2048byte。



而针对每一个 blocks，会使用一个 256KB/2K = 128 的数组来标识是否被使用，用来记录从这个 block 开始的空余的 buffer 单元。

128 127 126 1

出现 **swiotlb_full** 通常有两种情况。

case 1: swiotlb buffer 真的没有比较大的空闲空间了，基本用完了，通常此种 case 比较常见。

case 2: driver 需要的 memory 大于 256K，无法直接分配，通常这只 case 很少见，除非开发前期。

问题分析

此类最后都是直接的 KE，其实凶手可能在更远的过去把 swiotlb 给吃掉了，类似于 memory leaks。

一个普遍性的 debug 手法是，记录每一个 block 里面的 unit 的 memory 分配 backtrace，然后在 swiotlb_full 时打印出来。

参考性的代码如下：

```

/mtk71029 add for debug swiotlb full case.

#define BACKTRACE_LEVEL 12

```

```

struct swiotlb_record_t {

    pid_t pid;

    pid_t tid;

    size_t size;

    unsigned int backtrace_num;

    unsigned long entries[BACKTRACE_LEVEL];

}

static struct swiotlb_record_t * record_list = NULL;

static unsigned int get_kernel_backtrace(unsigned long *backtrace)
{
    unsigned long stack_entries[BACKTRACE_LEVEL];

    unsigned int i = 0;

    struct stack_trace trace = {

        .nr_entries = 0,

        .entries = &stack_entries[0],

        .max_entries = BACKTRACE_LEVEL,

        .skip = 2

    };

    save_stack_trace(&trace);

    if(trace.nr_entries > 0)

    {

        memcpy(backtrace, (unsigned long *)trace.entries, sizeof(unsigned int)*trace.nr_entries);

    }

    else

    {

        printk("[ERROR]can't get backtrace [get_kernel_backtrace] backtrace num: [%d]\n", trace.nr_entries);

    }

    return trace.nr_entries;

}

static void swiotlb_record_init() {

    unsigned long i = 0;

    record_list = alloc_bootmem_pages(PAGE_ALIGN(io_tlb_nslabs * sizeof(struct swiotlb_record_t)));

    for (i = 0; i < io_tlb_nslabs; i++) {

        record_list[i].pid = -1;

        record_list[i].tid = -1;

        record_list[i].backtrace_num = 0;

    }

}

static void swiotlb_record_set(int index, size_t size){

    int num;

```

```

record_list[index].pid = current->tgid;
record_list[index].tid = current->pid;
record_list[index].size = size;
num = get_kernel_backtrace(&(record_list[index].entries));
if (num > 0) {
record_list[index].backtrace_num = num;
}
}

static void swiotlb_record_clean(int index){
record_list[index].pid = -1;
record_list[index].tid = -1;
record_list[index].backtrace_num = 0;
}

static void swiotlb_record_print() {
unsigned long i = 0;
int j = 0;
for (i=0; i < io_tlb_nslabs; i++) {
if (io_tlb_list[i] == 0 && record_list[i].pid != -1 && record_list[i].tid != -1){
printk(KERN_ERR "%lu: %d %d %z", i, record_list[i].pid, record_list[i].tid, record_list[i].size);
for (j = 0; j < record_list[i].backtrace_num; j++ ) {
printk(KERN_ERR " 0x%16x", record_list[i].entries[j]);
}
printk(KERN_ERR "\n");
}
}
}

//mtk71029 add end.

static void
swiotlb_full(struct device *dev, size_t size, enum dma_data_direction dir,
int do_panic)
{
/*
* Ran out of IOMMU space for this operation. This is very bad.
* Unfortunately the drivers cannot handle this operation properly.
* unless they check for dma_mapping_error (most don't)
* When the mapping is small enough return a static buffer to limit
* the damage, or panic when the transfer is too big.
*/
//mtk71029 add for debug.
swiotlb_record_print();
//mtk71029 add end.
printk(KERN_ERR "DMA: Out of SW-IOMMU space for %zu bytes at "

```



```

"device %s\n", size, dev ? dev_name(dev) : "?");

BUG();

int __init swiotlb_init_with_tlb(char *tlb, unsigned long nslabs, int verbose)
{
    void *v_overflow_buffer;
    unsigned long i, bytes;

    bytes = nslabs << IO_TLB_SHIFT;

    io_tlb_nslabs = nslabs;
    io_tlb_start = __pa(tlb);
    io_tlb_end = io_tlb_start + bytes;

    /*
     * Get the overflow emergency buffer
     */
    v_overflow_buffer = alloc_bootmem_low_pages_nopanic(
        PAGE_ALIGN(io_tlb_overflow));
    if (!v_overflow_buffer)
        return -ENOMEM;

    io_tlb_overflow_buffer = __pa(v_overflow_buffer);

    /*
     * Allocate and initialize the free list array. This array is used
     * to find contiguous free memory regions of size up to IO_TLB_SEGSIZE
     * between io_tlb_start and io_tlb_end.
     */
    io_tlb_list = alloc_bootmem_pages(PAGE_ALIGN(io_tlb_nslabs * sizeof(int)));
    for (i = 0; i < io_tlb_nslabs; i++)
        io_tlb_list[i] = IO_TLB_SEGSIZE - OFFSET(i, IO_TLB_SEGSIZE);
    io_tlb_index = 0;
    io_tlb_orig_addr = alloc_bootmem_pages(PAGE_ALIGN(io_tlb_nslabs * sizeof(phys_addr_t)));

    //mtk71029 add for debug
    swiotlb_record_init();
    //mtk71029 add end.

    not_found:
    spin_unlock_irqrestore(&io_tlb_lock, flags);
    return SWIOTLB_MAP_ERROR;

    found:
    //mtk71029 add for debug
    swiotlb_record_set(index, size);
    //mtk71029 add end.

```

```

spin_unlock_irqrestore(&io_tlb_lock, flags);

/*
 * Save away the mapping from the original address to the DMA address.
 * This is needed when we sync the memory. Then we sync the buffer if
 * needed.
 */
for (i = 0; i < nslots; i++)
io_tlb_orig_addr[index+i] = orig_addr + (i << IO_TLB_SHIFT);
if (dir == DMA_TO_DEVICE || dir == DMA_BIDIRECTIONAL)
swiotlb_bounce(orig_addr, tlb_addr, size, DMA_TO_DEVICE);

return tlb_addr;
}

for (i = index + nslots - 1; i >= index; i--)
io_tlb_list[i] = ++count;
/*
 * Step 2: merge the returned slots with the preceding slots,
 * if available (non zero)
 */
for (i = index - 1; (OFFSET(i, IO_TLB_SEGSIZE) != IO_TLB_SEGSIZE - 1) && io_tlb_list[i]; i--)
io_tlb_list[i] = ++count;

//mtk71029 add for debug
swiotlb_record_clean(index);
//mtk71029 add end
}

spin_unlock_irqrestore(&io_tlb_lock, flags);
}

EXPORT_SYMBOL_GPL(swiotlb_tlb_unmap_single);

```

然后在 KE 后抓到这个资讯，解析 backtrace 即可捞出这些 memory 的分配情况，从而知道 swiotlb leaks 的凶手，下面说一个 MTK 内部的案例。

```

[<ffffffc000332d00>] swiotlb_alloc_coherent+0x60/0x14c
[<ffffffc000094c28>] __dma_alloc_coherent+0x24/0x74
[<ffffffc000094cd0>] __dma_alloc_noncoherent+0x58/0x16c
[<ffffffc0004542e4>] cmdq_core_alloc_hw_buffer+0xa8/0x100
[<ffffffc00045a2b4>] cmdq_core_task_realloc_buffer_size+0x54/0x24c
[<ffffffc00045a9ec>] cmdq_core_insert_read_reg_command+0xc0/0x83c
[<ffffffc000460804>] cmdqCoreSubmitTaskAsyncImpl+0x1c4/0x464
[<ffffffc0004623cc>] cmdqCoreSubmitTask+0x3c/0x100
[<ffffffc000466f88>] cmdq_ioctl+0x838/0xcb0
[<ffffffc0001fe9a8>] proc_reg_unlocked_ioctl+0x4c/0x7c
[<ffffffc0001afde4>] do_vfs_ioctl+0x354/0x584
[<ffffffc0001b0094>] Sys_ioctl+0x80/0x98

```

我们焦距到: cmdq_core_task_realloc_buffer_size 这个函数.

```
static int32_t cmdq_core_task_realloc_buffer_size(TaskStruct *pTask, uint32_t size)
{
    void *pNewBuffer = NULL;

    dma_addr_t newMVABase = 0;

    int32_t commandSize = 0;

    uint32_t *pCMDEnd = NULL;

    if (pTask->pVABase && pTask->bufferSize >= size) {
        /* buffer size is already good, do nothing. */
        return 0;
    }

    do {
        /* allocate new buffer, try if we can alloc without reclaim */
        pNewBuffer = cmdq_core_alloc_hw_buffer(cmdq_dev_get(), size,
            &newMVABase, GFP_KERNEL | __GFP_NO_KSWAPD);

        if (pNewBuffer) {
            pTask->useEmergencyBuf = false;
            break;
        }

        /* failed. Try emergency buffer */
        if (size <= CMDQ_EMERGENCY_BLOCK_SIZE)
            cmdq_core_alloc_emergency_buffer(&pNewBuffer, &newMVABase);

        if (pNewBuffer) {
            CMDQ_MSG("emergency buffer %p allocated\n", pNewBuffer);
            pTask->useEmergencyBuf = true;
            break;
        }

        /* finally try reclaim */
        pNewBuffer =
            cmdq_core_alloc_hw_buffer(cmdq_dev_get(), size, &newMVABase,
            GFP_KERNEL);

        if (pNewBuffer) {
            pTask->useEmergencyBuf = false;
            break;
        }
    } while (0);
}
```

```

if (NULL == pNewBuffer) {
    CMDQ_ERR("realloc cmd buffer of size %d failed\n", size);
    return -ENOMEM;
}

memset(pNewBuffer, 0, size);

/* copy and release old buffer */
if (pTask->pVABase)
    memcpy(pNewBuffer, pTask->pVABase, pTask->bufferSize);

/* we should keep track of pCMDEnd and cmdSize since they are cleared in free command buffer */
pCMDEnd = pTask->pCMDEnd;
commandSize = pTask->commandSize;

cmdq_task_free_task_command_buffer(pTask);

/* attach the new buffer */
pTask->pVABase = (uint32_t *) pNewBuffer;
pTask->MVABase = newMVABase;
pTask->bufferSize = size;
pTask->pCMDEnd = pCMDEnd;
pTask->commandSize = commandSize;

CMDQ_MSG("Task Buffer:0x%p, VA:%p PA:%pa\n", pTask, pTask->pVABase, &pTask->MVABase);
return 0;
}

```

在 DMA ZONE 没有大块的 memory 环境下, 会导致 memory leaks. 在 old buffer 没有释放之前, 就直接把 `pTask->useEmergencyBuf` 给设置了, 导致 old buffer 在 `cmdq_task_free_task_command_buffer(pTask)` 中无法释放。

8. BUG at mt_pmic_wrap_irq

异常现场:

当你在 SYS_KERNEL_LOG 里看到如下 log, 那么就属于 BUG at mt_pmic_wrap_irq 一类问题了:

```

[ 46.553472]<0>-(0)[0:swapper/0][PWRAP] ERROR, line=305 @@@@Timeout: elapse time10000307, start46543462771, current46553463078,
setting timer10000000
[ 46.553485]<0>-(0)[0:swapper/0][PWRAP] ERROR, line=387 wait_for_state_ready_init timeout when waiting for idle
[ 46.553496]<0>-(0)[0:swapper/0][PWRAP] ERROR, line=673 _pwrap_wacs2_nochk read fail, return_value=9
[ 46.553506]<0>-(0)[0:swapper/0][PWRAP] PMIC HW CID = 0x0
[ 46.553516]<0>-(0)[0:swapper/0][PWRAP] INT flag 0x7fffffff9
[ 46.553532]<0>-(0)[0:swapper/0]Unable to handle kernel paging request at virtual address 0000dead
.....
[ 46.554444]<0>-(0)[0:swapper/0]Internal error: Oops: 96000046 [#1] PREEMPT SMP

```

```
.....
[ 53.502536]<0>-(0)[0:swapper/0]CPU: 0 PID: 0 Comm: swapper/0 Tainted: G W 3.10.72+ #1
[ 53.502549]<0>-(0)[0:swapper/0]task: ffffffff000e142f0 ti: ffffffff000e00000 task.ti: ffffffff000e00000
[ 53.502565]<0>-(0)[0:swapper/0]PC is at mt_pmic_wrap_irq+0x13c/0x15c
[ 53.502577]<0>-(0)[0:swapper/0]LR is at mt_pmic_wrap_irq+0x12c/0x15c
```

代码位置:

kernel-3.18/drivers/misc/mediatek/pmic_wrap/\$platform/pwrap_hal.c

```
static irqreturn_t mt_pmic_wrap_irq(int irqno, void *dev_id)
{
    unsigned long flags = 0;

    .....

    pwrap_dump_all_register();

    /* clear interrupt flag */
    WRAP_WR32(PMIC_WRAP_INT0_CLR, 0xffffffff);
    PWRAPREG("INT0 flag 0x%x\n", WRAP_RD32(PMIC_WRAP_INT0_EN));
    if (10 == g_wrap_wdt_irq_count || 1 == g_case_flag)
        BUG_ON(1); /* 这里发生 KE */
    .....
}
```

问题解读:

对 PMIC 数据通信有硬件超时保护，如果超时表示已出问题，需要检查硬件。

问题解决:

检查 PMIC 硬件

9. BUG at __cpu_up

异常现场:

当你在 SYS_KERNEL_LOG 里看到如下 log，那么就属于 BUG at mt_pmic_wrap_irq 一类问题了：

```
<4>[41262.046308]-(0)[240:hps_main][name:mrdump&]Non-crashing CPUs did not react to IPI
<4>[41262.046361]-(0)[240:hps_main]CPU: 0 PID: 240 Comm: hps_main Tainted: G S W 4.4.22 #2
<4>[41262.046375]-(0)[240:hps_main]Hardware name: mt6799 (DT)
<4>[41262.046384]-(0)[240:hps_main]task: ffffffff0afcea000 ti: ffffffff0afcec000 task.ti: ffffffff0afcec000
<4>[41262.046399]-(0)[240:hps_main]PC is at __cpu_up+0x180/0x2bc
<4>[41262.046409]-(0)[240:hps_main]LR is at __cpu_up+0x180/0x2bc
```

代码位置:

kernel-4.4/drivers/misc/mediatek/pmic_wrap/\$platform/pwrap_hal.c

```

int __cpu_up(unsigned int cpu, struct task_struct *idle)
{
    int ret;

    /* We need to tell the secondary core where to find its stack and the page tables. */
    secondary_data.stack = task_stack_page(idle) + THREAD_START_SP;
    __flush_dcache_area(&secondary_data, sizeof(secondary_data));

    /* Now bring the CPU into our world. */
    ret = boot_secondary(cpu, idle);
    TIMESTAMP_REC(hotplug_ts_rec, TIMESTAMP_FILTER, cpu, 0, 0, 0);

    if (ret == 0) {
        /* CPU was successfully started, wait for it to come online or time out. */
        wait_for_completion_timeout(&cpu_running,
            msecs_to_jiffies(1000));

        if (!cpu_online(cpu)) {
            pr_crit("CPU%u: failed to come online\n", cpu);
            ret = -EIO;

            BUG_ON(1); /* 这里会 KE, 有些版本是 Warning!!! */
        }
    } else {
        pr_err("CPU%u: failed to boot: %d\n", cpu, ret);
    }

    .....
}

```

问题解读:

通常都是被启动的 CPU 在执行 CPU_STARTING 时容易被阻塞, 比如被大量 log 阻塞。

问题解决:

检查是哪个模块打印大量 log 并修复

10. 调用 kmalloc 前关闭抢断引起 KE

问题背景:

手机后台收集的异常 db, 其中有一类问题疑似软件问题。

分析过程:

用 GAT 解开 db, 并结合对应的 vmlinux (该文件必须和 db 一致, 具体请看 FAQ06985), 利用工具分析 (也可以参考 FAQ13941), 解析出来的调用栈如下:

详细描述: 进程 Binder_9 在原子上下文 (关闭抢断) 发起调度, 请从调用栈查找关抢断代码并修复

参考信息: MediaTek On-Line> Quick Start> 深入分析 Linux kernel exception 框架> 实例篇: 案例分析> BUG at __schedule_bug

异常时间: 68526.582303 秒

== CPU 信息 ==

崩溃 CPU 信息:

CPU4: 进程名: Binder_9, 进程标识符(pid): 1523

本地调用栈:

```
vmlinux __schedule_bug(prev=0xFFFFF000E049000) + 104 <kernel/sched/core.c:3074>
vmlinux schedule_debug() + 1472 <kernel/sched/core.c:3088>
vmlinux __schedule() +
1548 <kernel/sched/core.c:3182>
vmlinux schedule() +
36 <kernel/sched/core.c:3279>
vmlinux __down_write_nested(subclass=0) + 112 <lib/rwsem-spinlock.c:213>
vmlinux __down_write() +
12 <lib/rwsem-spinlock.c:225>
vmlinux down_write() +
8 <kernel/rwsem.c:50>
vmlinux zram_bvec_rw(zram=0xFFFFF000460CF000, bvec=0xFFFFF00037C75C68, index=27300, offset=0, rw=1) + 56
<drivers/staging/zram/zram_drv.c:900>
vmlinux __zram_make_request() + 268 <drivers/staging/zram/zram_drv.c:955>
vmlinux zram_make_request(bio=0xFFFFF00037C75C00) + 400 <drivers/staging/zram/zram_drv.c:1010>
vmlinux generic_make_request(bio=0xFFFFF00037C75C00) + 132 <block/blk-core.c:1844>
vmlinux submit_bio(rw=1, bio=0xFFFFF00037C75C00) + 172 <block/blk-core.c:1936>
vmlinux __swap_writepage(page=0xFFFFF000C025F6290) + 552 <mm/page_io.c:321>
vmlinux swap_writepage(page=0xFFFFF000C025F6290, wbc=0xFFFFF0006712B4A8) + 44 <mm/page_io.c:250>
vmlinux pageout(page=0xFFFFF000C025F6290, mapping=0xFFFFF0000E84688) + 368 <mm/vmscan.c:492>
vmlinux shrink_page_list(page_list=0xFFFFF0006712B690, zone=0xFFFFF0000F36640, sc=0xFFFFF0006712B8E8, ttu_flags=TTU_UNMAP) + 800
<mm/vmscan.c:993>
vmlinux shrink_inactive_list(lruvec=0xFFFFF0000F36A88, sc=0xFFFFF0006712B8E8, lru=LRU_INACTIVE_ANON) + 608 <mm/vmscan.c:1476>
vmlinux shrink_list() + 428 <mm/vmscan.c:1807>
vmlinux shrink_lruvec(lruvec=0xFFFFF0000F36A88, sc=0xFFFFF0006712B8E8) + 1100 <mm/vmscan.c:2154>
vmlinux shrink_zone(sc=0xFFFFF0006712B8E8) + 60 <mm/vmscan.c:2323>
vmlinux shrink_zones() + 128 <mm/vmscan.c:2468>
vmlinux do_try_to_free_pages(zonelist=0xFFFFF0000F37900, sc=0xFFFFF0006712B8E8, shrink=0xFFFFF0006712B8D8) + 240
<mm/vmscan.c:2535>
vmlinux try_to_free_pages(zonelist=0xFFFFF0000F37900, order=1, gfp_mask=2118352, nodemask=0) + 236 <mm/vmscan.c:2766>
vmlinux __perform_reclaim() + 52 <mm/page_alloc.c:2332>
vmlinux __alloc_pages_direct_reclaim() + 52 <mm/page_alloc.c:2353>
vmlinux __alloc_pages_slowpath() + 700 <mm/page_alloc.c:2626>
vmlinux __alloc_pages_nodemask() + 1144 <mm/page_alloc.c:2845>
vmlinux __alloc_pages() <include/linux/gfp.h:311>
vmlinux alloc_pages_exact_node() <include/linux/gfp.h:329>
vmlinux alloc_slab_page() + 48 <mm/slub.c:1314>
vmlinux allocate_slab() + 76 <mm/slub.c:1336>
vmlinux new_slab(s=0xFFFFF00047001840, flags=32976, node=-1) + 132 <mm/slub.c:1397>
vmlinux new_slab_objects() + 104 <mm/slub.c:2171>
```

```

vmlinux __slab_alloc(s=0xFFFFFFFFC047001840, gfpflags=32976) + 640 <mm/slub.c:2332>
vmlinux slab_alloc_node() + 296 <mm/slub.c:2406>
vmlinux slab_alloc() + 296 <mm/slub.c:2446>
vmlinux __kmallocc(size=8192, flags=32976) + 372 <mm/slub.c:3278>
vmlinux kmallocc() + 20 <include/linux/slub_def.h:174>
vmlinux kzallocc() + 20 <include/linux/slab.h:520>
vmlinux cmdq_rec_reallocc_cmd_buffer(handle=0xFFFFFFFFC0B1A5E800) + 36 <drivers/misc/mediatek/cmdq/cmdq_record.c:51>
vmlinux cmdq_rec_reallocc_cmd_buffer() + 12 <drivers/misc/mediatek/cmdq/cmdq_record.c:47>
vmlinux cmdq_append_command(handle=0xFFFFFFFFC0B1A5E800, code=CMDQ_CODE_WRITE, argA=335634148, argB=1043319794) + 172
<drivers/misc/mediatek/cmdq/cmdq_record.c:197>
vmlinux cmdqRecWrite(handle=0xFFFFFFFFC0B1A5E800, value=1043319794, mask=4294967295) + 76
<drivers/misc/mediatek/cmdq/cmdq_record.c:435>
vmlinux disp_gamma_write_lut_reg(cmdq=0xFFFFFFFFC0B1A5E800, id=DISP_GAMMA0, lock=0) + 452
<drivers/misc/mediatek/disp/disp_gamma.c:110>
vmlinux disp_gamma_set_lut() + 136 <drivers/misc/mediatek/disp/disp_gamma.c:153>
vmlinux disp_gamma_io(module=DISP_MODULE_GAMMA, msg=1208252439) + 196 <drivers/misc/mediatek/disp/disp_gamma.c:297>
vmlinux dpmgr_path_user_cmd(dp_handle=0xFFFFFFFFC045E0F000, msg=1208252439, arg=547229475328, cmdqhandle=0xFFFFFFFFC0B1A5E800) + 380
<drivers/misc/mediatek/disp/disp_gamma.c:1124>
vmlinux primary_display_user_cmd() + 184 <drivers/misc/mediatek/video/mt6795/primary_display.c:4917>
vmlinux mtk_disp_mgr_ioctl() + 1008 <drivers/misc/mediatek/video/mt6795/mtk_disp_mgr.c:2040>
vmlinux disp_unlocked_ioctl() + 8 <drivers/misc/mediatek/video/mt6795/mtk_mira.c:21>
vmlinux proc_reg_unlocked_ioctl(file=0xFFFFFFFFC07209A100, arg=547229475328) + 76 <fs/proc/inode.c:252>
vmlinux vfs_ioctl() + 20 <fs/ioctl.c:43>
vmlinux do_vfs_ioctl(filp=0xFFFFFFFFC07209A100, fd=310, cmd=1208252439, arg=547229475328) + 852 <fs/ioctl.c:598>
vmlinux SYSC_ioctl() + 112 <fs/ioctl.c:613>
vmlinux Sys_ioctl() + 128 <fs/ioctl.c:604>
vmlinux cpu_switch_to() + 72 <arch/arm64/kernel/entry.S:673>
== 栈结束 ==

```

这题是 BUG at __schedule_bug，因此需要找出是谁关闭了抢断，需要从调用栈入手。

我们一一排查，不过前面一截都是 kernel 原生函数，不应该有问题，因此从 cmdq_rec_reallocc_cmd_buffer() 函数开始排查，一一检查哪里关闭了抢断。

我们发现 disp_gamma_set_lut() 函数有问题，代码如下：

```

static int disp_gamma_set_lut(const DISP_GAMMA_LUT_T __user *user_gamma_lut, void *cmdq)
{
    .....

    if (0 <= id && id < DISP_GAMMA_TOTAL) {
        spin_lock(&g_gamma_global_lock);

        old_lut = g_disp_gamma_lut[id];
        g_disp_gamma_lut[id] = gamma_lut;

        ret = disp_gamma_write_lut_reg(cmdq, id, 0);
    }
}

```



```
spin_unlock(&g_gamma_global_lock);  
.....  
}
```

可以看到在调用 `disp_gamma_write_lut_reg()` 函数前，调用了 `spin_lock()`，而 `spin_lock()` 函数会关闭抢断的!!!

一般情况下 `spin lock` 包含的代码必须尽快执行完后释放掉，以免出现死锁等复杂问题。

而 `kmalloc()` 一般带有 `GFP_KERNEL`，可能会引起睡眠！这直接导致了这题的 KE。

根本原因：

`kmalloc(x, GFP_KERNEL)` 会引起睡眠，不能在 `spin lock` 内使用！用其他可睡眠的锁或用 `GFP_ATOMIC` 类型分配内存。

kernel 开发的工程师要特别注意这个问题。

解决方法：

将 `spin_lock` 换成 `mutex_lock`。

11. 变量没有锁保护

问题背景：

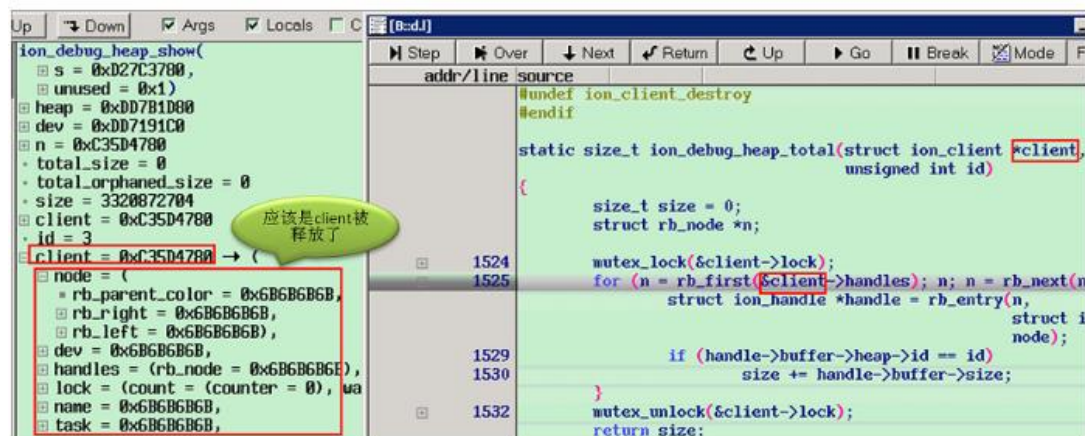
跑 monkey 跑出 KE。

分析过程：

用 GAT 解开 db，查看 db.14.dbg/SYS_KERNEL_LOG:

```
<1>[58586.202903] (0)[10683:aee_dumpstate]Unable to handle kernel paging request at virtual address 6b6b6b73  
.....  
<4>[58586.204622] (1)[10683:aee_dumpstate][<c028d508>] (rb_first+0x0/0x30) from [<c02bf1f4>] (ion_debug_heap_show+0x80/0x2a0)  
<4>[58586.204642] (1)[10683:aee_dumpstate][<c02bf174>] (ion_debug_heap_show+0x0/0x2a0) from [<c0172538>] (seq_read+0x1c0/0x4a4)  
<4>[58586.204662] (1)[10683:aee_dumpstate][<c0172378>] (seq_read+0x0/0x4a4) from [<c0151e80>] (vfs_read+0xac/0x13c)  
<4>[58586.204681] (1)[10683:aee_dumpstate][<c0151dd4>] (vfs_read+0x0/0x13c) from [<c0151f54>] (sys_read+0x44/0x70)  
.....  
<4>[58586.204833]-(1)[10683:aee_dumpstate]PC is at rb_first+0x20/0x30  
<4>[58586.204844]-(1)[10683:aee_dumpstate]LR is at ion_debug_heap_show+0x80/0x2a0  
<4>[58586.204856]-(1)[10683:aee_dumpstate]pc : [<c028d528>] lr : [<c02bf1f4>] psr: 20000013  
<4>[58586.204863]-(1)[10683:aee_dumpstate]sp : c5f07e78 ip : c5f07e88 fp : c5f07e84  
<4>[58586.204873]-(1)[10683:aee_dumpstate]r10: 00000003 r9 : c35d4794 r8 : c5f07f00  
<4>[58586.204884]-(1)[10683:aee_dumpstate]r7 : dd7191c0 r6 : d27c3780 r5 : dd7b1d80 r4 : c35d4780  
<4>[58586.204895]-(1)[10683:aee_dumpstate]r3 : 00000000 r2 : 00d9b000 r1 : 00000000 r0 : 6b6b6b6b  
<4>[58586.204907]-(1)[10683:aee_dumpstate]Flags: nzCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user  
<4>[58586.204919]-(1)[10683:aee_dumpstate]Control: 10c5387d Table: 9279c06a DAC: 00000015
```

地址 `0x6b6b6b73`，感觉像是 `slub free object` 填充字节，拿到 `vmlinux`，建立 `trace32 debug` 环境：



查看 client 所在地址:

```

NSD:C35D4770 5A5A5A5A 5A5A5A5A 5A5A5A5A 5A5A5A5A
NSD:C35D4780 6B6B6B6B 6B6B6B6B 6B6B6B6B 6B6B6B6B
NSD:C35D4790 6B6B6B6B 00000000 00000000 6B6B6B6B
NSD:C35D47A0 6B6B6B6B 6B6B6B6B 6B6B6B6B 6B6B6B6B

```

可以明显看到, 刚好只向 object 头部, 所以证实了 client 被释放的猜测。

接着查看 client 从哪里来, 看到代码:

```

static int ion_debug_heap_show(struct seq_file *s, void *unused)
{
    struct ion_heap *heap = s->private;
    struct ion_device *dev = heap->dev;
    struct rb_node *n;
    size_t total_size = 0;
    size_t total_orphaned_size = 0;

    seq_printf(s, "%s", "pid", "size");
    seq_printf(s, "-----\n");

    for (n = rb_first(&dev->clients); n; n = rb_next(n)) {
        struct ion_client *client = rb_entry(n, struct ion_client,
                                             node);
        size_t size = ion_debug_heap_total(client, heap->id);
    }
}

```

所以是红黑树节点出现问题了, 查看 ion.c 的逻辑比较简单, 检查所有对这红黑树的操作 (搜索 dev->clients 关键字), 需要特别关注的是删除操作, 结果发现:

```

static size_t ion_debug_heap_total(struct ion_client *client,
                                   unsigned int id)
{
    size_t size = 0;
    struct rb_node *n;
    mutex_lock(&client->lock);
    for (n = rb_first(&client->handles); n; n = rb_next(n)) {
        struct ion_handle *handle = rb_entry(n,
                                              struct ion_handle,
                                              node);
        if (handle->buffer->heap->id == id)
            size += handle->buffer->size;
    }
    mutex_unlock(&client->lock);
    return size;
}

void ion_client_destroy(struct ion_client *client)
{
    struct ion_device *dev = client->dev;
    struct rb_node *n;

    pr_debug("%s: %d\n", __func__, __LINE__);
    while ((n = rb_first(&client->handles))) {
        struct ion_handle *handle = rb_entry(n, struct ion_handle,
                                              node);
        ion_handle_destroy(&handle->ref);
    }
    down_write(&dev->lock);
    if (client->task)
        put_task_struct(client->task);
    rb_erase(&client->node, &dev->clients);
    debugfs_remove_recursive(client->debug_root);
    up_write(&dev->lock);
}

```

有client->lock保护

没有client->lock保护?

所以很明显，在删除时没有锁保护导致变量被释放了。

根本原因:

变量没有锁保护导致的 race condition。

解决方法:

请对应工程师修复。

PS: 如果那块释放的 object 又被分配出去，那就非常麻烦了。

12. work 没有初始化就使用引起 KE

问题背景:

在弱光的环境下，开启闪光灯拍照，手机死机重启。

分析过程:

用 GAT 解开 db，并结合对应的 vmlinux（该文件必须和 db 一致，具体请看 FAQ06985），先查看 db 里的 __exp_main.txt，发现是 KE，这时再查看 SYS_KERNEL_LOG，log 如下（SYS_KERNEL_LOG）：

```

<2>[ 506.632801]-(0)[0:swapper/0]Kernel BUG at c0078c8c [verbose debug info unavailable]

<0>[ 506.632835]-(0)[0:swapper/0]Internal error: Oops - BUG: 0 [#1] PREEMPT SMP ARM

<4>[ 506.632867]-(0)[0:swapper/0]Modules linked in: wlan_mt6628 bf1ea000 ccci bf1bf000 ccci_plat bf192000 mtk_wmt_wifi
bf18e000 mtk_fm_drv bf161000 mtk_stp_bt bf153000 mtk_stp_gps bf146000 mtk_stp_uart bf13d000 mtk_stp_wmt
bf0ba000 mtk_hif_sdio bf0a4000 devinfo bf0a0000 devapc bf099000 sec bf077000 vcodec_kernel_driver bf066000 mtklfb
bf05b000 pvrsvrkm bf000000

```

```

<4>[ 506.633140]-(0)[0:swapper/0]CPU: 0      Tainted: G          W          (3.4.5 #1)
<4>[ 506.633181]-(0)[0:swapper/0]PC is at __queue_work+0x3b8/0x4c8
<4>[ 506.633213]-(0)[0:swapper/0]LR is at __queue_work+0x68/0x4c8
<4>[ 506.633248]-(0)[0:swapper/0]pc : [<c0078c8c>]      lr : [<c007893c>]      psr: 00000193
<4>[ 506.633265]-(0)[0:swapper/0]sp : c0981c38   ip : c0981c38   fp : c0981c6c
<4>[ 506.633300]-(0)[0:swapper/0]r10: 00000003   r9 : c22d0b80   r8 : c22d0940
<4>[ 506.633332]-(0)[0:swapper/0]r7 : 00000000   r6 : 00000000   r5 : c22d6100   r4 : c11f9d80
<4>[ 506.633366]-(0)[0:swapper/0]r3 : c11f9d84   r2 : 00000000   r1 : dd833000   r0 : c22d6100
<4>[ 506.633402]-(0)[0:swapper/0]Flags: nzcv   IRQs off   FIQs on   Mode SVC_32   ISA ARM   Segment kernel
<4>[ 506.633439]-(0)[0:swapper/0]Control: 10c5387d   Table: 98dfc06a   DAC: 00000015

```

BUG at c0078c8c :可以知道是直接调用 BUG() trigger ke 重启。 使用 arm-linux-androideabi-objdump 将 vmlinux 反编译出来:

```

119067 c007893c: e30523b8      movw    r2, #21432      ; 0x53b8
119068 c0078940: e34c20a0      movt    r2, #49312     ; 0xc0a0
119069 c0078944: e5923018      ldr     r3, [r2, #24]
119070 c0078948: e3530000      cmp     r3, #0
119071 c007894c: e1a05000      mov     r5, r0
119072 c0078950: 1a000095      bne     c0078bac <__queue_work+0x2d8>
119073 c0078954: e5942004      ldr     r2, [r4, #4]
119074 c0078958: e2843004      add     r3, r4, #4
119075 c007895c: e1520003      cmp     r2, r3
119076 c0078960: 1a0000c9      bne     c0078c8c <__queue_work+0x3b8>
119077 c0078964: e5952008      ldr     r2, [r5, #8]

```

搜索 c0078c8c , 是从 0xc0078960 跳过去的, 使用 trace32 加载 vmlinux 看到的结果:

```

__NSD:C0078954|E5942004_____ldr____r2, [r4, #0x4]
NSD:C0078958|E2843004 add r3, r4, #0x4 ; r3, work, #4
NSD:C007895C|E1520003 cmp r2, r3
NSD:C0078960|1A0000C9 bne 0xC0078C8C
从这里可以知道 R4 c11f9d80 为 work, 其成员的 value 为:
__address_|_____0_____4_____8_____C_0123456789ABCDEF
NSD:C11F9D80|>00000001(data) 00000000(entry->next) 00000000(entry->prev) 00000000(func) .....
NSD:C11F9D90| 00000000 00000000 00000000 00000000 .....
NSD:C11F9DA0| 00000000 00000001 00000000 00000000 .....
NSD:C11F9DB0| 00000190 00000000 C11F9DB8 00000000 .....

```

对照 C 代码: 去判断 work->entry 是否为 NULL, 即 work->entry+0x04 是否等于 work->entry->next, 条件不成立导致 bug.

```

static void __queue_work(unsigned int cpu, struct workqueue_struct *wq, struct work_struct *work)
{
    .....

    BUG_ON(!list_empty(&work->entry));

    .....
}

```

从 work 变量的 value 来看, 有 2 种可能性:

1. work 被踩。
2. work 没有初始化。

从反馈来看每次都必现，比较怀疑是 work 没有初始化就去 queue_work。从 backtrace 知道是 ledTimeOutCallback 调用的 queue_work:

```
<4>[ 506.647360]-(0)[0:swapper/0][<c00788d4>] (__queue_work+0x0/0x4c8) from [<c0078e10>] (queue_work_on+0x44/0x4c)
<4>[ 506.647406]-(0)[0:swapper/0][<c0078dcc>] (queue_work_on+0x0/0x4c) from [<c0078e68>] (queue_work+0x2c/0x60)
<4>[ 506.647441]-(0)[0:swapper/0] r6:c22d0bd8 r5:dd833000 r4:c11f9d80 r3:00010003
<4>[ 506.647527]-(0)[0:swapper/0][<c0078e3c>] (queue_work+0x0/0x60) from [<c0078ebc>] (schedule_work+0x20/0x24)
<4>[ 506.647562]-(0)[0:swapper/0] r5:00000000 r4:c11f9db8
<4>[ 506.647625]-(0)[0:swapper/0][<c0078e9c>] (schedule_work+0x0/0x24) from [<c044ba04>] (ledTimeOutCallback+0x18/0x20)
```

拿到对应的 source code:

```
enum hrtimer_restart ledTimeOutCallback(struct hrtimer *timer)
{
    PK_DBG("ledTimeOut_callback\n");
    schedule_work(&workTimeOut);
    return HRTIMER_NORESTART;
}
```

review source code, 发现在 open 时候有去 init_work, 这就比较奇怪了, 后来确认, 在 open 这个 devier 的时候并没有去 init_work, 提供的 code 是之后修改的. 使用修改后的 leds_strobe.c 文件, 测试没有发生重启情况。

根本原因:

workTimeOut 没有 init, 就执行 queue_work 导致 KE。

解决方法:

在第一次 constant_flashlight_open 的时候对 workTimeOut 进行初始化一次, 之后 open 不需要:

```
constant_flashlight_open()
{
    .....
    static flag = 0;

    if (!flag) {
        INIT_WORK(&workTimeOut, work_timeOutFunc);
        flag = 1;
    }
    .....
}
```

13. tasklet 没有初始化就使用引起 KE

问题背景:

擦除坏卡引起 KE, 必现。

分析过程:

用 GAT 解开 db, 并结合对应的 vmlinux (该文件必须和 db 一致, 具体请看 FAQ06985), 利用工具 (E-Consulter.jar) 分析 (也可以参考 FAQ13941), 解析出来的调用栈如下:

```
== 异常报告 v1.4 (仅供参考) ==

详细描述: tasklet_hi_action() 调用了错误的函数指针使程序跑到非法地址 (0x0000000000000000) 执行, 请结合崩溃进程调用栈检查相关代码
异常时间: 2799.826556 秒, Mon Feb 29 14:21:44 CST 2016

== CPU 信息 ==

崩溃 CPU 信息:
CPU0: 进程名: ksoftirqd/0, 进程标识符(pid): 3

本地调用栈:
.....
vmlinux tasklet_hi_action(a=softirq_vec) + 156 <kernel/softirq.c:556>
vmlinux __do_softirq() + 200 <kernel/softirq.c:279>
vmlinux run_ksoftirqd(cpu=0) + 60 <kernel/softirq.c:677>
vmlinux smpboot_thread_fn(data=0xFFFFF0BD848E40) + 508 <kernel/smpboot.c:160>
vmlinux kthread(_create=0xFFFFF0BD848E80) + 212 <kernel/kthread.c:207>
== 栈结束 ==
```

报告告诉你跑飞了, 查看对应的代码:

```
static void tasklet_hi_action(struct softirq_action *a)
{
    struct tasklet_struct *list;

    local_irq_disable();

    list = __this_cpu_read(tasklet_hi_vec.head);
    __this_cpu_write(tasklet_hi_vec.head, NULL);
    __this_cpu_write(tasklet_hi_vec.tail, this_cpu_ptr(&tasklet_hi_vec.head));
    local_irq_enable();

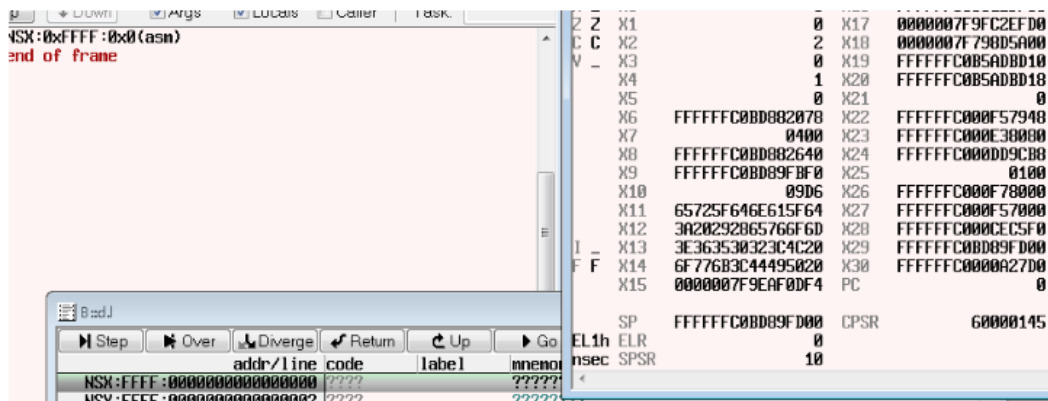
    while (list) {
        struct tasklet_struct *t = list;

        list = list->next;

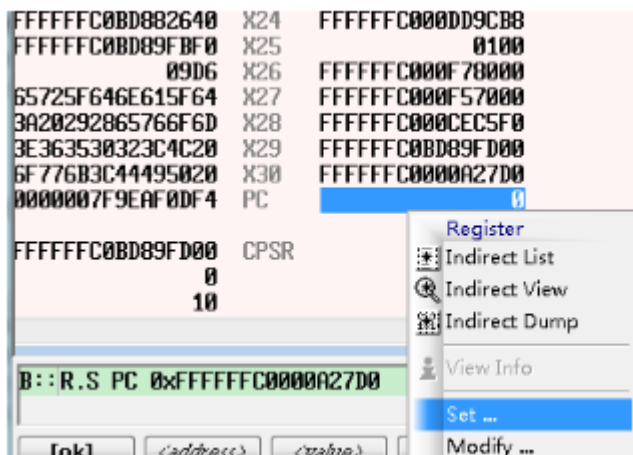
        if (tasklet_trylock(t)) {
            if (!atomic_read(&t->count)) {
                if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                    BUG();

                t->func(t->data); /* 这里发生 KE */
            }
        }
    }
}
```

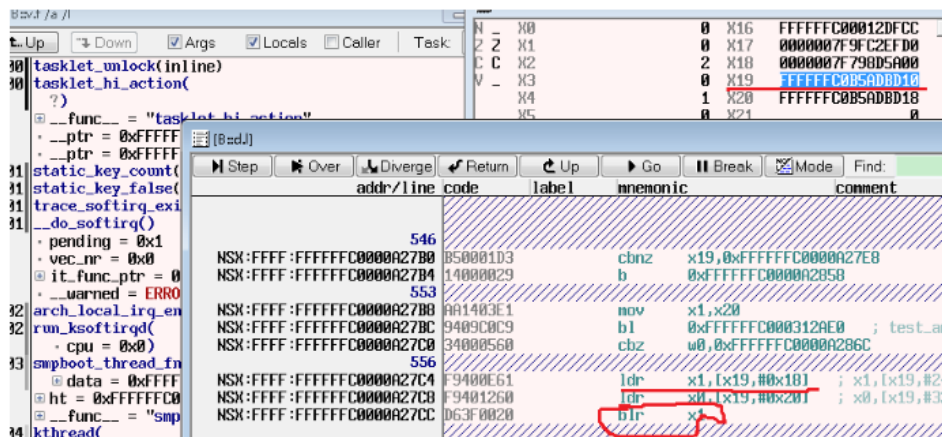
这时需要启动 trace32 了, E-Consulter.jar 会自动生成 debug.cmm, 用 trace32 加载, 完成后如图:



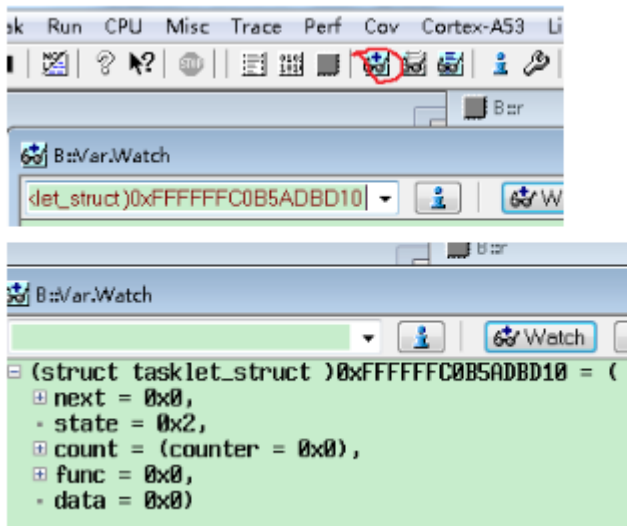
因为跑飞了，所以什么都看不到，我们要把 LR 的值赋值给 PC（函数调用都会把返回地址赋值给 LR）：



设定完成后，现场就出来了：



可以看到 BLR X1 指令是函数调用，X1 的值为 0，所以跑飞了，结合代码来看，X1 = t->func，而 X19 就是 t 了，我们可以借助 trace32 查看 t：



明显看到 func 为 NULL 引起的 KE，那这是哪个 tasklet 呢？这个结构体无法告诉我们是谁插入 tasklet 的。

回头看下这个问题，是擦除坏卡引起，是否和 msdc1 有关呢，查看 log：

```
[ 2799.826167] <0> (0) [8203:kworker/0:0][sd]msdc1 -> abort timeout. Card stuck in 7 state, bad card! remove it! <-
msdc_check_write_timeout() : L<8280> PID<kworker/0:0><0x200b>

[ 2799.826248] <0> (0) [8203:kworker/0:0][sd]msdc1 -> remove the bad card, block_bad_card=1,card_inserted=0 <-
msdc_set_bad_card_and_remove() : L<2056> PID<kworker/0:0><0x200b>
```

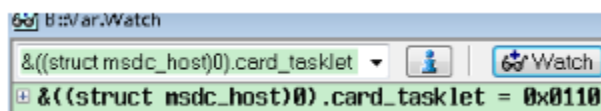
很可能是 msdc 的问题，搜索所有调用 __tasklet_hi_schedule() / __tasklet_hi_schedule_first() 的函数，发现有一处和 msdc 有关，代码如下：

```
static void msdc_set_bad_card_and_remove(struct msdc_host *host)
{
    .....

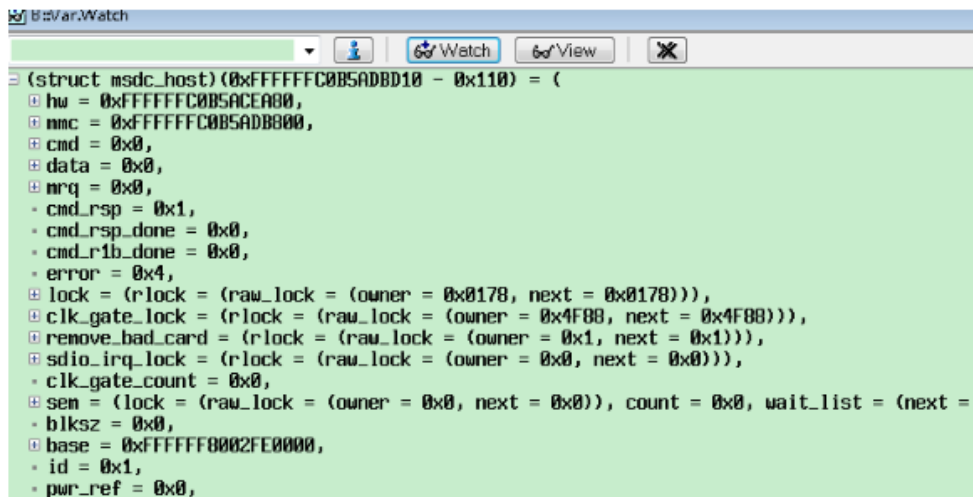
    if (! (host->mmc->caps & MMC_CAP_NONREMOVABLE) && (got_polarity ^ host->hw->cd_level))
        tasklet_hi_schedule(&host->card_tasklet); /* 这里插入 tasklet */

    .....
}
```

是否就是这个 host->card_tasklet 的问题呢？我们可以反推下 host 结构体，然后看是否是一个正常的 mmc_host 结构体，先计算下 host 到 card_tasklet 成员的偏移量：



结果似乎 0x110，然后 t - 0x110 的地址为 msdc_host 结构体地址：



查看这个结构体的几个关键栏位，确定是正常的 msdc host 结构体，也就是说正是 msdc_set_bad_card_and_remove() 函数插入的 tasklet 引起了 KE。

检查下所有用到 card_tasklet 的代码，发现就是没有任何代码调用 tasklet_init() 来初始化 card_tasklet 成员，问题就处在这里。

根本原因：

msdc 没有初始化 card_tasklet 就插入 tasklet。

解决方法：

不需要 tasklet，相关代码需要删除。

14. 内存泄漏引起 KE

问题背景：

用户反馈系统重启。

分析过程：

用 GAT 解开 db，查看 __exp_main.txt：

```
Exception Class: Kernel (KE)
PC is at [<fffffc0003e2800>] mrdump_mini_ke_cpu_regs+0x50/0xc4
```

看到这个信息，表示系统主动调用 kernel panic。这时需要看 SYS_KERNEL_LOG，搜索关键字 Kernel panic - not syncing:

```
[35649.303767]<3>-(3) [28285:netd]Kernel panic - not syncing: Out of memory and no killable processes...
```

发现 panic 的原因是 OOM，这行 log 是在这里的代码打印的：

[kernel-3.18/mm/oom_kill.c](#):

```
void out_of_memory(struct zonelist *zonelist, gfp_t gfp_mask, int order, nodemask_t *nodemask, bool force_kill)
{
    const nodemask_t *mpol_mask;
    struct task_struct *p;
```

```

unsigned long totalpages;

.....

p = select_bad_process(&points, totalpages, mpol_mask, force_kill);

/* Found nothing?!?! Either we hang forever, or we panic. */

if (!p) {

    dump_header(NULL, gfp_mask, order, NULL, mpol_mask);

    panic("Out of memory and no killable processes...\n");

}

.....

}

```

OOM 表示 kernel 已没有内存可以回收了，需要看哪里泄漏了。可能的原因有：

1. 用户进程泄漏，LMK 不是所有进程都去回收的，是根据 adj 来判断，低于 0 不回收，由 init 启动的 service 都不回收，像 system_server，surfaceflinger 等是不会被回收的，而如果这些进程发生内存泄漏，最后只能 OOM 了。
2. kernel 泄漏，比如 slub、vmalloc 泄漏。

一般 OOM 前会印系统内存信息，有助于我们判断哪里泄漏，在 panic 前有看到如下 log：

```

[35649.261853]<3> (3) [28285:netd]netd invoked oom-killer: gfp_mask=0x3000d0, order=2, oom_score_adj=-1000
[35649.261873]<3> (3) [28285:netd]CPU: 3 PID: 28285 Comm: netd Tainted: G W 3.10.65+ #1
[35649.261883]<3> (3) [28285:netd]Call trace:
[35649.261901]<3> (3) [28285:netd] [<ffffffc000088f50>] dump_backtrace+0x0/0x16c
[35649.261915]<3> (3) [28285:netd] [<ffffffc0000890cc>] show_stack+0x10/0x1c
[35649.261931]<3> (3) [28285:netd] [<ffffffc0009a03b8>] dump_stack+0x1c/0x28
[35649.261947]<3> (3) [28285:netd] [<ffffffc000150244>] dump_header.isra.14+0x6c/0x1b0
[35649.261960]<3> (3) [28285:netd] [<ffffffc000150ef8>] out_of_memory+0x2ec/0x2fc
[35649.261975]<3> (3) [28285:netd] [<ffffffc000155b54>] __alloc_pages_nodemask+0x788/0x79c
[35649.261991]<3> (3) [28285:netd] [<ffffffc0000984a4>] copy_process+0x114/0x10fc
[35649.262003]<3> (3) [28285:netd] [<ffffffc00009953c>] do_fork+0x7c/0x3c4
[35649.262015]<3> (3) [28285:netd] [<ffffffc00009991c>] SyS_clone+0x10/0x1c
[35649.262023]<3> (3) [28285:netd]Mem-Info:
.....
[35649.262096]<3> (3) [28285:netd]active_anon:552580 inactive_anon:138202
[35649.262096]<3> active_file:1172 inactive_file:1532 isolated_file:32
[35649.262096]<3> unevictable:631 dirty:23 writeback:1590
[35649.262096]<3> free:1971 slab_reclaimable:2893 slab_unreclaimable:7057
[35649.262096]<3> mapped:1387 shmem:123 pagetables:2379
[35649.262139]<3> (3) [28285:netd]DMA free:7884kB min:6932kB low:19372kB high:21104kB active_anon:2210320kB
inactive_anon:552808kB active_file:4688kB inactive_file:6128kB unevictable:2524kB
isolated(file):128kB present:3072960kB managed:3004616kB dirty:92kB writeback:6360kB mapped:5548kB shmem:492kB
slab_reclaimable:11572kB slab_unreclaimable:28228kB kernel_stack:6160kB pagetables:9516kB pages_scanned:16804 all_unreclaimable?
yes
[35649.212755]<3> (3) [28285:netd]powerkey_kick:primary_display_trigger:2,6
[35649.262154]<3> (3) [28285:netd]lowmem_reserve[]:[35649.262161]<3> (3) [28285:netd] 0 0 0

```

```

[35649.262173]<3> (3) [28285:netd]DMA: 1922*4kB (UEM) 1*8kB (R) 7*16kB (R) 1*32kB (R) 0*64kB 2*128kB (R) 0*256kB 0*512kB 0*1024kB
0*2048kB 0*4096kB = 8096kB
[35649.262230]<3> (3) [28285:netd]3923 total pagecache pages
[35649.262239]<3> (3) [28285:netd]294 pages in swap cache
[35649.262249]<3> (3) [28285:netd]Swap cache stats: add 211983, delete 211689, find 245614/274392
[35649.262257]<3> (3) [28285:netd]Free swap = 0kB
[35649.262265]<3> (3) [28285:netd]Total swap = 524284kB
[35649.302860]<3> (3) [28285:netd]768240 pages RAM
[35649.302874]<3> (3) [28285:netd]16722 pages reserved
[35649.302882]<3> (3) [28285:netd]46561 pages shared
[35649.302891]<3> (3) [28285:netd]740566 pages non-shared
[35649.302900]<3> (3) [28285:netd][ pid ] uid tgid total_vm rss nr_ptes swapents oom_score_adj name
[35649.303030]<3> (3) [28285:netd][ 236] 1000 236 138735 1303 63 862 -1000 surfaceflinger
[35649.303251]<3> (3) [28285:netd][ 277] 1021 277 12652 190 26 137 -1000 mtk_agpsd
[35649.303303]<3> (3) [28285:netd][ 279] 1021 279 4003 145 8 61 -1000 mnlld
[35649.303467]<3> (3) [28285:netd][ 305] 0 305 242916 2214 114 2237 -1000 main
[35649.303508]<3> (3) [28285:netd][ 312] 1023 312 4074 1078 9 45 -1000 sdcard
[35649.303536]<3> (3) [28285:netd][ 330] 0 330 107 2 4 9 -1000 ku.sud
[35649.303549]<3> (3) [28285:netd][ 332] 0 332 112 15 4 1 -1000 ku.sud
[35649.303562]<3> (3) [28285:netd][ 424] 0 424 790291 668810 1547 121425 -1000 ku.sud
[35649.303576]<3> (3) [28285:netd][ 609] 2000 609 3731 5 8 158 -1000 emdlogger1
[35649.303590]<3> (3) [28285:netd][ 610] 1001 610 6958 46 13 68 -1000 gsm0710muxd
[35649.303604]<3> (3) [28285:netd][ 760] 0 760 2714 29 6 122 -1000 debuggerd_real
[35649.303617]<3> (3) [28285:netd][ 853] 1001 853 12059 132 22 151 -1000 mtkrild
[35649.303631]<3> (3) [28285:netd][ 1473] 0 1473 65 3 5 7 -1000 kd
[35649.303644]<3> (3) [28285:netd][ 5758] 0 5758 63 1 5 7 -1000 ktools
[35649.303658]<3> (3) [28285:netd][10776] 0 10776 65 10 5 0 -1000 kd
[35649.303676]<3> (3) [28285:netd][26884] 1010 26884 4356 245 9 0 -1000 wpa_supplicant
[35649.303689]<3> (3) [28285:netd][27143] 1014 27143 2586 79 6 0 -1000 dhcpcd
[35649.303703]<3> (3) [28285:netd][27915] 1003 27915 67212 5752 61 0 -1000 bootanimation
[35649.303717]<3> (3) [28285:netd][28284] 0 28284 17478 922 34 0 -1000 zygote64
[35649.303731]<3> (3) [28285:netd][28285] 0 28285 3226 194 7 0 -1000 netd
[35649.303744]<3> (3) [28285:netd][28286] 0 28286 9923 594 19 0 -1000 mediaserver
[35649.303757]<3> (3) [28285:netd][28415] 0 28415 207610 12988 61 0 -1000 main
.....
[35649.303767]<3>-(3) [28285:netd]Kernel panic - not syncing: Out of memory and no killable processes...

```

我们来看如何解读这个 log，第 1 行 log 表示 netd 要申请内存申请不到，需要启动 LMK 释放一些内存，申请的尺寸是 order = 2， $2^2 = 4$ page = 4 * 4K。

zone 只有一个 dma，管理的内存大小是：3G（present:3072960kB）。

DMA zone 只有 8M（8096kB）的内存了，已经低于低水位了（low:19372kB），存在泄漏，而且 zram 也用光了（Free swap = 0kB）。

我们发现 active_anon:2210320kB inactive_anon:552808kB 这 2 个加起来超过 2.7G，表示用户进程泄漏了。

查看用户进程 rss 栏位，找出最大的那个进程是 ku.sud，泄漏内存大小是 668810 * 4K = 2.6G。很明显需要分析这个进程泄漏原因。

用户进程泄漏排查方法，可以参考：[FAQ14715] 如何分析 native memory leak

根本原因：

用户进程泄漏。

解决方法：

排查这个用户进程泄漏问题。

PS：kernel 内存管理是很大一个系统，需要较多背景知识，建议大家了解下这个背景，对分析问题很有帮助。

15. spin lock 没有 unlock 引起 KE

问题背景：

当插上 USB Cable 连接 PC 时，在 Setting 中选择 Factory Reset，手机会正常 Reboot，但在 Reboot 过程中发生了 KE。

问题必现。

分析过程：

用 GAT 解开 db，并结合对应的 vmlinux（该文件必须和 db 一致，具体请看 FAQ06985），利用工具 E-Consulter 分析，产生分析报告如下：

== 异常报告 v1.7 (仅供参考) ==

报告解读：MediaTek On-Line> Quick Start> E-Consulter 之 NE/KE 分析报告解读> KE 分析报告

详细描述：函数返回跑飞 (0x00014D20)，可能是栈帧被踩坏，请结合崩溃进程调用栈检查相关代码

异常时间：9.451213 秒，Thu Jan 1 06:06:47 GMT 2015

== CPU 信息 ==

崩溃 CPU 信息：

CPU5: 进程名：init，进程标识符(pid): 0

寄存器异常：SP (0xBEB2DB58) 异常

本地调用栈：

..... 0x00014D20 ()

== 栈结束 ==

0x00014D20 明显不是 kernel space 的，而是 user space 的，而且 SP 也是 user space 的。特别是 SP，kernel space 的 SP 不可能跑到 user space 去的（SP 是编译器按 ABI 规则操作），怎么回事？需要从 SYS_KERNEL_LOG 检查下实际情况：

```
[ 9.451213].(5)[1:init]Unable to handle kernel paging request at virtual address b6475c40
```

```
[ 9.451219].(5)[1:init]pgd = db884000
```

```
[ 9.451237].(5)[1:init][b6475c40] *pgd=9b88a831, *pte=a01377df, *ppte=a0137e7f
```

```
[ 9.451248].(5)[1:init][KERN Warning] ERROR/WARN forces debug_lock off!
```

```
[ 9.451252].(5)[1:init][KERN Warning] check backtrace:
```

```
[ 9.451260].(5)[1:init]Backtrace:
```

```
[ 9.451287].(5)[1:init][] (dump_backtrace+0x0/0x110) from [] (dump_stack+0x18/0x1c)
```

```
[ 9.451307].(5)[1:init] r6:df84bfb0 r5:c0c57408 r4:b6475c40 r3:00000000
```

```
[ 9.451327].(5)[1:init][] (dump_stack+0x0/0x1c) from [] (debug_locks_off+0x5c/0x78)
[ 9.451350].(5)[1:init][] (debug_locks_off+0x0/0x78) from [] (oops_enter+0x14/0x30)
[ 9.451367].(5)[1:init][] (oops_enter+0x0/0x30) from [] (die+0x30/0x438)
[ 9.451387].(5)[1:init][] (die+0x0/0x438) from [] (__do_kernel_fault.part.11+0x6c/0x7c)
[ 9.451405].(5)[1:init][] (__do_kernel_fault.part.11+0x0/0x7c) from [] (do_page_fault+0x2f8/0x3d8)
[ 9.451416].(5)[1:init] r7:db80e000 r3:df84bfb0
[ 9.451433].(5)[1:init][] (do_page_fault+0x0/0x3d8) from [] (do_DataAbort+0x7c/0x110)
[ 9.451448].(5)[1:init][] (do_DataAbort+0x0/0x110) from [] (__dabt_usr+0x38/0x40)
[ 9.451456].(5)[1:init]Exception stack(0xdf84bfb0 to 0xdf84bff8)
[ 9.451465].(5)[1:init]bfa0: 00000000 00000bdf 00000000 00000000
[ 9.451476].(5)[1:init]bfc0: 00000000 00000000 b6475c20 00000001 00000000 beb2dbca 00000000 b6475c20
[ 9.451485].(5)[1:init]bfe0: 00000000 beb2db58 0001e4e5 00014d20 000e0030 ffffffff
[ 9.451493]-(5)[1:init]Internal error: Oops: 80f [#1] PREEMPT SMP ARM
```

整理的调用栈如下：

```
die
__do_kernel_fault
do_page_fault
do_DataAbort
__dabt_usr
```

比较奇怪的是 user space 发生缺页异常，最后怎么就跑到__do_kernel_fault()呢？流程不对。检查下所有从 do_page_fault() 调用到 __do_kernel_fault()的代码：

```
static int __kprobes do_page_fault(unsigned long addr, unsigned int fsr, struct pt_regs *regs)
{
    .....

    /* If we're in an interrupt, or have no irqs, or have no user context, we must not take the fault.. */
    if (in_atomic() || irqs_disabled() || !mm)
        goto no_context;

    .....

    if (!down_read_trylock(&mm->mmap_sem)) {
        if (!user_mode(regs) && !search_exception_tables(regs->ARM_pc))
            goto no_context;

    retry:
        down_read(&mm->mmap_sem);
    }

    .....

    /* If we are in kernel mode at this point, we have no context to handle this fault with. */
    if (!user_mode(regs))
        goto no_context;

    .....

    return 0;
}
```

```
no_context:
    __do_kernel_fault(mm, addr, fsr, regs);

    return 0;
}
```

有 3 个地方跳到了 `__do_kernel_fault()`，可以排除最后 2 个，因为都是 `user_mode`！第 1 个的意思是（缺页异常时 `mm` 不为 `NULL`）：

- 抢断被关闭了
- 中断被关闭了

我们检查下抢断和中断的状态，先检查中断，这里用 `gdb` 加载 `SYS_MINI_RDUMP`，E-Consulter 会生成 `gdb.bat`（需要安装 GAT 才行），双击进入 `gdb` 环境：

```
(gdb) info registers
r0 0x0 0
r1 0xbdf 3039
r2 0x0 0
r3 0x0 0
r4 0x0 0
r5 0x0 0
r6 0xb6475c20 3058129952
r7 0x1 1
r8 0x0 0
r9 0xb6b2dbca 3199392714
r10 0x0 0
r11 0xb6475c20 3058129952
r12 0x0 0
sp 0xb6b2db58 0xb6b2db58
lr 0x1e4e5 124133
pc 0x14d20 0x14d20
cpsr 0xe0030 917552
(gdb)
```

看到 CPSR，检查 `bit7`，`bit7` 为 0 表示中断是打开的。

那么一定是抢断被关闭了，检查抢断状态，抢断是 `preempt_count()`，保存在 `thread_info` 结构体里，要找到 `thread_info`，需要先找到 `kernel space` 的 SP 指针，`gdb` 里列出的是 `user space` 的 SP 指针，不能用，需要从 `SYS_KERNEL_LOG` 里捞：

```
[ 9.451456].(5)[1:init]Exception stack(0xdf84bfb0 to 0xdf84bff8)
```

根据 SP 找到 `thread_info`：

```
(gdb) p *(struct thread_info *)0xdf84a000
$1 = {flags = 0, preempt_count = 1, addr_limit = 3204448256,
      task = 0xdf82f000, exec_domain = 0xc0c6e3fc, cpu = 5,
```

```

cpu_domain = 21, cpu_context = {r4 = 3682656256, r5 = 0, r6 = 3749900288,
r7 = 3251788736, r8 = 3750273024, r9 = 3234167456, sl = 3750010896,
fp = 3750018644, sp = 3750018448, pc = 3230389240, extra = {0, 0}},
syscall = 0, used_cp = '\000' , tp_value = 375848,
fpstate = {hard = {save = {0 }}, soft = {save = {
0 }}}}, vfpstate = {hard = {fpregs = {
4428012001347007264, 700294288497737198, 8457589029229192297,
8030591510794691188, 511, 0, 3622055676180045924,
8102936033331473952, 0, 0, 0, 0, 0, 0, 0, 0, 9, 734,
0 }}, fpexc = 1073741824, fpscr = 0,
fpinst = 3682656256, fpinst2 = 3750010936, cpu = 8}},
thumbee_state = 0, restart_block = {
fn = 0xc0079fc0 , {futex = {
uaddr = 0x0 , val = 0, flags = 0, bitset = 0,
time = 0, uaddr2 = 0x0 }, nanosleep = {clockid = 0,
rmtp = 0x0 , expires = 0}, poll = {
ufds = 0x0 , nfds = 0, has_timeout = 0, tv_sec = 0,
tv_nsec = 0}}}, cpu_excp = 0, regs_on_excp = 0xdf84bd98}
Warning: the current language does not match this frame.
(gdb)

```

看到 `preempt_count = 1`，明显抢断被关闭了！！

抢断只有在 `kernel space` 才能设定，那么问题是怎么发生的呢？一定是返回 `user space` 前抢断就被关闭了！！

什么情况下会返回 `user space`？有以下情况：

- 进程创建第一次返回
- 系统调用返回
- 缺页异常返回
- 被中断打断后返回

只有系统调用返回是有风险的，因为其他都是 `kernel common flow`，不可能出现这种问题，而系统调用可能会调用到具体的驱动，驱动没有写好代码可能关闭抢断。

驱动里什么情况下会关闭抢断？有如下情况：

- `spin lock` 没有 `unlock`
- `spin lock` 实现的 `rw lock`
- 手动关闭抢断，不过在驱动里一般不会这样干。

如何排查这样的问题？有想到如下方法：

- 检查代码。这问题必现，在打入某笔 `patch` 前没有问题，问题就出在 `patch` 身上，但 `patch` 涉及的文件多，检查代码不现实！
- 在 `add_preempt_count()` 添加调用栈打印。

选择了在 `add_preempt_count()` 添加代码查看谁关闭了抢断：

```

kernel-3.10/kernel/sched/core.c 里的

void __kprobes add_preempt_count(int val)
{
    if (DEBUG_LOCKS_WARN_ON((preempt_count() < 0)))
        return;

    preempt_count() += val;

    if ((preempt_count() & PREEMPT_MASK) == 1 && !strcmp(current->comm, "init")) //添加这行代码
        show_stack(NULL, NULL); //添加这行代码
    .....
}

```

结果导入后不开机了，发现 log 量太大了，优化如下：

```

if ((preempt_count() & PREEMPT_MASK) == 1 && !strcmp(current->comm, "init")) {
    unsigned long ip = get_parent_ip(CALLER_ADDR1);
    char buf[KSYM_SYMBOL_LEN];

    sprint_symbol(buf, ip);
    if (strncmp(buf, "mtk_uart_console_write", sizeof("mtk_uart_console_write") - 1)
        && strncmp(buf, "__slab_", sizeof("__slab_") - 1)
        && strncmp(buf, "deactivate_slab", sizeof("deactivate_slab") - 1)
        && strncmp(buf, "console_unlock", sizeof("console_unlock") - 1)
        && strncmp(buf, "vprintk_emit", sizeof("vprintk_emit") - 1)
        && strncmp(buf, "__wake_up", sizeof("__wake_up") - 1)
        && strncmp(buf, "handle_irq_event", sizeof("handle_irq_event") - 1)
        && strncmp(buf, "hrtimer_", sizeof("hrtimer_") - 1)
        && strncmp(buf, "up+", sizeof("up+") - 1)
        && strncmp(buf, "down_trylock", sizeof("down_trylock") - 1)
        && strncmp(buf, "kmem_cache_alloc", sizeof("kmem_cache_alloc") - 1)
        && strncmp(buf, "free_debug_processing", sizeof("free_debug_processing") - 1)
        && strncmp(buf, "mutex_lock", sizeof("mutex_lock") - 1)
        && strncmp(buf, "__kmalloc", sizeof("__kmalloc") - 1))
        printk("%s", buf);
}

```

结果还是不行，log 还是很乱，找不到问题，打印 log 不合适。

回到问题本身，在返回 user space 前就出问题了，所以应该在返回前拦截，再配合 add_preempt_count() 抓取调用栈，然后在拦截后将调用栈打印出来。修改代码如下：

```

1. arch/arm/kernel/entry-header.S

在 restore_user_regs 里面添加调用 preempt_check()

.macro restore_user_regs, fast = 0, offset = 0

```



```
bl preempt_check @add this line
```

```
ldr r1, ....
```

```
.....
```

```
.endm
```

2. kernel\sched\core.c

添加代码到 add_preempt_count():

```
void __kprobes add_preempt_count(int val)
```

```
{
```

```
    if (DEBUG_LOCKS_WARN_ON((preempt_count() < 0)))
```

```
        return;
```

```
    preempt_count() += val;
```

```
    if ((preempt_count() & PREEMPT_MASK) == 1 && !strcmp(current->comm, "init")) { /* 添加这段代码 */
```

```
        struct stack_trace trace;
```

```
        trace.nr_entries = 0;
```

```
        trace.max_entries = 5;
```

```
        trace.entries = current->tracetbl;
```

```
        trace.skip = 1;
```

```
        memcpy(trace.entries, 0, 5 * sizeof(*trace.entries));
```

```
        save_stack_trace(&trace);
```

```
    }
```

```
    .....
```

```
}
```

3. kernel\sched\core.c

添加函数:

```
asmlinkage void preempt_check(void)
```

```
{
```

```
    if (preempt_count()) {
```

```
        printk("yan: %pS, %pS, %pS, %pS, %pS
```

```
", (void *)current->tracetbl[0], (void *)current->tracetbl[1], (void *)current->tracetbl[2], (void *)current->tracetbl[3], (void *)current->tracetbl[4]);
```

```
        BUG();
```

```
    }
```

```
}
```

4. include/linux/sched.h

添加一个数组到 task_struct 的最后

```
struct task_struct
```

```
{  
  
    volatile long state;  
  
    .....  
  
    unsigned long tracetbl[5]; //add this line  
  
}
```

导入复现抓到 db, 分析报告如下:

== 异常报告 v1.7(仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> E-Consulter 之 NE/KE 分析报告解读> KE 分析报告

详细描述: 断言失败(主动调用 BUG()/BUG_ON()), 请结合崩溃进程调用栈检查相关代码

异常时间: 9.928621 秒, Thu Jan 1 06:06:58 GMT 2015

== CPU 信息 ==

崩溃 CPU 信息:

CPU1: 进程名: init, 进程标识符(pid): 0, 中断: 关

本地调用栈:

vmlinux preemption_check() + 96 <kernel/sched/core.c:8641>

vmlinux 0xC000F12C() <arch/arm/kernel/entry-common.S:44>

== 栈结束 ==

对应汇编指令:

行号 地址 指令 提示

kernel/sched/core.c

8641: C00A019C: UNDEF E7F001F2 ; 进程停止在这里

当时的寄存器值:

R0: 00000043, R1: 0001392E, R2: 010CC000, R3: DF82F000, R4: 00000005, R5: BED1CBF0, R6: B6429727, R7: 00000004

R8: C000F2C4, R9: DF84A000, R10: 00000000, R11: DF84BFA4, R12: DF84BEE8, SP: DF84BF90, R14: C00ECB0C, PC: C00A019C

可以抓到现场了, 查看 SYS_KERNEL_LOG:

```
[ 9.478543]. (4) [116:bat_thread_kthr]yan: _raw_spin_lock+0x10/0x40, mt_usb_connect+0x134/0x433, mt_usb_store_cmode+0x433/0x500, dev_attr_store+0x20/0x40, sysfs_write_file+0x40/0x60
```

检查 mt_usb_connect() 函数的代码:

```
void mt_usb_connect(void)  
{  
  
    .....  
  
    spin_lock(&musb_connect_lock); /* 这里拿到了 spin lock */  
#ifdef MTK_CDP_ENABLE  
    .....  
#else  
    if(cable_mode != CABLE_MODE_NORMAL) {
```

```

        DBG(0, "musb_sync_with_bat, USB_CONFIGURED\n");

        musb_sync_with_bat(mtk_musb, USB_CONFIGURED);

        mtk_musb->power = true;

        return; /* 这里没有 unlock 就返回了!!! */
    }
#endif

    .....

    spin_unlock(&musb_connect_lock);

    .....
}

```

明显有条路径没有 unlock，导致了抢断没有被打开。

后面检查代码，发现这个 lock 属于 MTK_CDP_ENABLE 里的，应该将 lock 用 MTK_CDP_ENABLE 包起来：

```

void mt_usb_connect(void)
{
    .....

#ifdef MTK_CDP_ENABLE

    spin_lock(&musb_connect_lock); /* 这里拿到了 spin lock */

    .....

#else

    if(cable_mode != CABLE_MODE_NORMAL) {

        DBG(0, "musb_sync_with_bat, USB_CONFIGURED\n");

        musb_sync_with_bat(mtk_musb, USB_CONFIGURED);

        mtk_musb->power = true;

        return;

    }

#endif

    .....

#ifdef MTK_CDP_ENABLE

    spin_unlock(&musb_connect_lock);

#endif

    .....
}

```

解决方法：

musb_connect_lock 用宏 MTK_CDP_ENABLE 包起来

结语：

在驱动里用 spinlock 要特别小心，所有路径都要注意，否则就会出现这种没有 unlock 的情况

16. copy_from_user 复制 non cacheable 内存引起 KE

问题背景：

拔 usb 时出现 KE at __copy_from_user+0xc/0x60。

复现一次。

分析过程：

用 GAT 解开 db，并结合对应的 vmlinux（该文件必须和 db 一致，具体请看 FAQ06985），利用工具 E-Consulter 分析，产生分析报告如下：

== 异常报告 v1.5(仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> E-Consulter 之 NE/KE 分析报告解读> KE 分析报告

详细描述: 从错误的地址 (0x00000000E67A1FFC) 读数据，请结合崩溃进程调用栈检查相关代码

平台 : MT6757

版本 : alps-mp-m1.mp3/userdebug build

异常时间: 2357.948363 秒, Sun Sep 18 10:45:23 CST 2016

== CPU 信息 ==

崩溃 CPU 信息:

CPU1: 进程名: MtkOmxVdecDecod, 进程标识符(pid): 9546, 中断: 关

寄存器异常: 内核栈溢出(已踩坏底部 thread_info 结构体)

本地调用栈:

vmlinux __copy_from_user(参数 1=0xFFFFF0C0E0CC7C88, 参数 2=0x00000000E67A1FFC, 参数 3=24) + 12 <arch/arm64/lib/copy_from_user.S: 35>

vmlinux arm_backtrace_eabi() + 140 <drivers/gator/gator_backtrace.c:87>

vmlinux gator_add_sample() + 528 <drivers/gator/gator_backtrace.c:228>

vmlinux gator_backtrace_handler() + 576 <drivers/gator/gator_main.c:341>

.....

== 栈结束 ==

对应汇编指令:

行号 地址 指令 提示

arch/arm64/lib/copy_from_user.S

35 : FFFFFFFC00032CC1C: LDR X3, [X1], #0x8 ; __copy_from_user() 参数 2 可能有问题

当时的寄存器值:

X0: FFFFFFFC0E0CC7C88, X1: 00000000E67A1FFC, X2: 0000000000000010, X3: 0000000000000002

X4: 00000000E67A2014, X5: FFFFFFFC00101D000, X6: 0000000000000D1D, X7: 0000000000000D1A

X8: 0000000000000001, X9: FFFFFFFC001265800, X10: FFFFFFFF80095E5000, X11: 0101010101010101

X12: 0000000000000038, X13: 00000000E4F83540, X14: 00000000F5BB69C7, X15: 0000000000000000

X16: FFFFFFFC000120764, X17: 00000000E4F834C0, X18: 00000000E4F834B8, X19: 00000000E67A1FFC

X20: 0000000000000080, X21: 00000027F05179E0, X22: 0000000000000001, X23: 0000000000000001

X24: 0000000000000004, X25: 000000000000007F, X26: FFFFFFFC0E0CC4000, X27: 0000000000000051

X28: 0000000000000080, X29: FFFFFFFC0E0CC7C20, X30: FFFFFFFC0008E5884, SP: FFFFFFFC0E0CC7C20

PC: FFFFFFFC00032CC1C

是在__copy_from_user()函数里 KE, X1=0xE67A1FFC,这个地址应该是 user space 传下来的。比较奇怪，在__copy_from_user 函数里即使 source address 是错误的，也不会崩溃，最多返回没复制的字节数。

我们需要从 kernel log 看个究竟：

```
[ 2357.947351]-(1)[9546:MtkOmxVdecDecod]Unhandled fault: alignment fault (0x96000021) at 0x00000000e67a1ffc
[ 2357.947385]-(1)[9546:MtkOmxVdecDecod]SCTLR : 34d5d91d
[ 2357.947401]-(1)[9546:MtkOmxVdecDecod]pgd = fffffffc0f9f8c000
[ 2357.947414]-(1)[9546:MtkOmxVdecDecod][e67a1ffc] *pgd=00000000f89f4003, *pud=00000000f89f4003, *pmd=00000000dc0ff003, *pte=03e0000016021f43
```

看到是发生对齐错误，X1 是 4 字节对齐，不过对齐势能是关闭的，查看 SCTLR 寄存器，通过查找 ARM datasheet 知道，bit1 表示是否要关闭对齐检查，这里是 1，表示关闭了对齐检查，那么 LDR X3, [X1], #0x8 就不应该发生对齐错误！

这是怎么回事呢？其实有 2 种情况是无视对其检查开关的：

- 互斥访问指令，比如 LDREX、STREX 等指令
- 访问的内存地址是 device memory 的。

在这里是第 2 种情况。如何确认内存地址是否是 device memory 呢？刚好 kernel log 有印出页表信息，在页表信息里有包含 cache 属性。放在哪里呢？根据 ARM datasheet，放在 PTE 里，我们解读 PTE 如下：

```
*pte=03e0000016021f43
```

- phys = 0x16021000
- 其他 bit 位：nG:1, AF:1, SH:11, AP:01, NS:0, AttrIndex:000

其中 attrindex 为内存属性，在 kernel 里将 000 设定为 device memory：

```
#define MT_DEVICE_nGnRnE 0
#define pgprot_noncached(prot) __pgprot_modify(prot, PTE_ATTRINDX_MASK, PTE_ATTRINDX(MT_DEVICE_nGnRnE) | PTE_PXN | PTE_UXN)
```

看到 PTE 对应的物理地址是 VDEC 设备的寄存器地址，且是 device memory 的，不能用 __copy_from_user，只能用 memcpy_io 和 memset_io 等给 io 操作的函数。

这题还需追查为何会将设备地址传递下来。

根本原因

不能用 copy_from_user 复制设备内存。

解决方法：

追查代码逻辑，杜绝传递设备内存给含有 copy_from_user 的驱动

结语：

需要对 ARM load/store 异常逻辑熟悉，才能想到 root cause。

17. 并发操作 pinctrl_select_state 导致的 KE

问题背景：

压力测试发生 KE

分析过程：

用 GAT 解开 db，并结合对应的 vmlinux（该文件必须和 db 一致，具体请看 FAQ06985），利用工具（E-Consulter.jar）分析（也可以参考 FAQ13941），解析出来的调用栈如下：

```
报告解读: MediaTek On-Line> Quick Start> NE/KE 分析报告解读> KE 分析报告
详细描述: 从错误的地址 (0x0000000000000000) 读数据，请结合崩溃进程调用栈检查相关代码
平台 : MT6771
版本 : alps-mp-p0.mp3/user build
异常时间: 9138.599936 秒, Fri Jan 4 13:56:11 CST 2019

== 平台信息 ==
-- CPU [0] 1 2 3 4 5 6 7 --
-- DDR4 --
-- Low Power --
SPM last pc: 0x0

== CPU 信息 ==
崩溃 CPU 信息:
CPU0: kworker/u16:5, pid: 24345
本地调用栈:
vmlinux __pi_streamp(参数 1=0, 参数 2=0xFFFFFFFFC05D9396C0) + 24 <arch/arm64/lib/streamp.S:77>
vmlinux pin_request() + 244 <drivers/pinctrl/pinmux.c:120>
vmlinux pinmux_enable_setting(setting=0xFFFFFFFFC059519E40) + 456 <drivers/pinctrl/pinmux.c:429>
vmlinux pinctrl_commit_state(p=0xFFFFFFFFC059511400, state=0xFFFFFFFFC059519E00) + 180 <drivers/pinctrl/core.c:1012>
vmlinux pinctrl_select_state() + 20 <drivers/pinctrl/core.c:1066>
vmlinux ccm_xxx_disable_vib(hap=0xFFFFFFFFC059515618) + 96 <drivers/misc/xxx_vibcam/src/xxx_vib8846.c:347>
vmlinux stop_motor_work(hap=0xFFFFFFFFC059515618) + 164 <drivers/misc/xxx_vibcam/src/xxx_vib8846.c:529>
vmlinux xxx_vib_stage_enable(hap=0xFFFFFFFFC059515618) + 268 <drivers/misc/xxx_vibcam/src/xxx_vib8846.c:611>
vmlinux motor_work_func(work=0xFFFFFFFFC059515698) + 120 <drivers/misc/xxx_vibcam/src/xxx_vib8846.c:794>
vmlinux process_one_work() + 504 <kernel/workqueue.c:2064>
vmlinux worker_thread() + 844 <kernel/workqueue.c:2122>
vmlinux kthread() + 276 <kernel/kthread.c:212>
vmlinux ret_from_fork() + 16 <arch/arm64/kernel/entry.S:1029>
== 栈结束 ==
```

通过 trace32 推导 backtrace 可知, 是因为 desc->mux_owner 空指针导致:

```

-000|__pi_strcmp(asm)
-001|pin_request()
    descriptor = (modname = ERROR:MMUFAIL, function = ERROR:MMUF
    @pctldev = 0xFFFFFC05EA710C0 ≡ end+0x40547200C0
    · status = 0xFFFFFEEA
    @ops = 0xFFFFF8008E1E658 ≡ mtk_pmx_ops
    @desc = 0xFFFFFC05EA77B80 ≡ end+0x4054726880 → (
    @pctldev = 0xFFFFFC05EA710C0 ≡ end+0x40547200C0,
    @name = 0xFFFFF800918DDDE ≡ → "GPIO170",
    · dynamic_name = FALSE,
    · mux_usecount = 0x2,
    @mux_owner = 0x0 ≡ → NULL,
    @mux_setting = 0xFFFFFC059519F28 ≡ end+0x404F1C8F28,
    @gpio_owner = 0x0 ≡ → NULL)

```

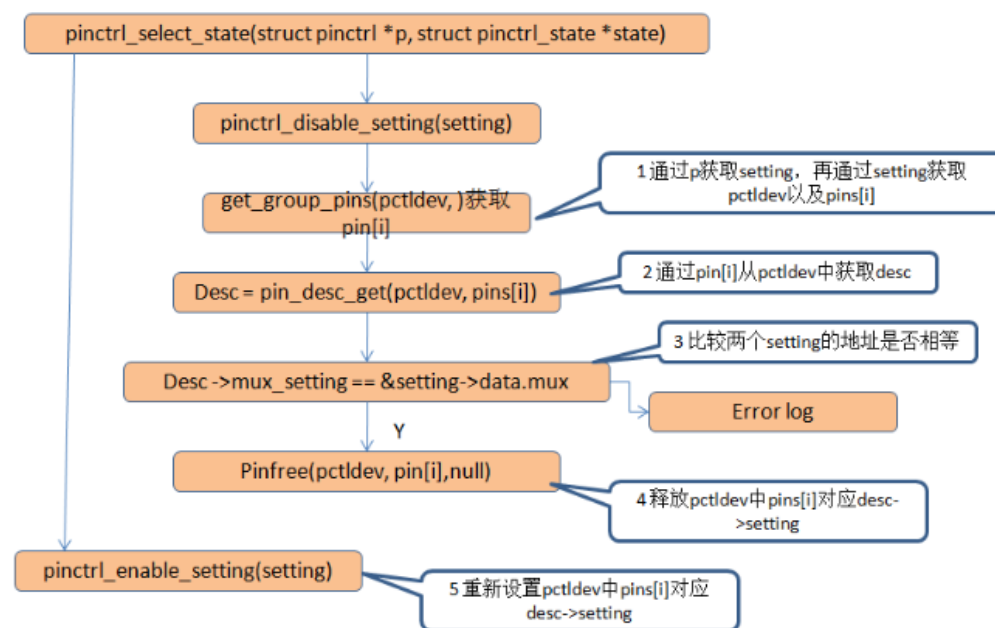
而 desc 是通过 desc = pin_desc_get(pctldev, pin); 获取的，而 pctldev, pin 是 pinmux_enable_setting() 传递给 pin_request() 的参数：

drivers/pinctrl/pinmux.c:429

```
ret = pin_request(pctldev, pins[i], setting->dev_name, NULL);
```

即是通过 pins[i] 从 pctldev 中获取

而从 backtrace 中可以看到，是从客户 driver 调用到 pinctrl_select_state() call 上来的，将 pinctrl_select_state() 的大概流程展开：



从以上流程可以推测，假设有人同时调用 pinctrl_select_state() 且第一个参数 p 是相同的 (这里假设是 A 和 B)，那么如果 A 在 run 到 1 后，B 跑到 4 或者 5，那么 A 通过 pin[i] 获取的 desc 已经被改变

从 trace32 看到，当前是 GPIO170，所以怀疑是有人同时也在操作 GPIO170，请客户 review 自己的 driver code，即使是单一模块在用，但可能存在多线程同时调用 pinctrl_select_state 的情况。

根本原因：

并发操作 pinctrl_select_state

解决方法:

建议在调用 `pinctrl_select_state` 的函数加锁保护（不是在 `pinctrl_select_state` 加锁），以防止多 thread 同时调用

18. strncpy_from_user 复制 device memory 引起 KE

问题背景:

monkey 测试出现 KE at `strncpy_from_user+0x6c/0x11c`。

复现好几次。

分析过程:

用 GAT 解开 db，并结合对应的 `vmlinux`（该文件必须和 db 一致，具体请看 FAQ06985），利用工具 E-Consulter 分析，产生分析报告如下：

== 异常报告 v1.5(仅供参考) ==

报告解读: MediaTek On-Line> Quick Start> E-Consulter 之 NE/KE 分析报告解读> KE 分析报告

详细描述: 从错误的地址 (0x0000000F0A7FFFC) 读数据，请结合崩溃进程调用栈检查相关代码

版本 : alps-trunk-n0.tk/userdebug build

异常时间: 58878.250223 秒, Sat May 6 11:37:32 CST 2017

== CPU 信息 ==

崩溃 CPU 信息:

CPU8: 进程名: media.life, 进程标识符(pid): 16831

本地调用栈:

vmlinux do_strncpy_from_user() + 80 <lib/strncpy_from_user.c:43>

vmlinux strncpy_from_user(src=0x0000000F0A7FFFC, count=4068) + 108 <lib/strncpy_from_user.c:115>

vmlinux getname_flags(flags=0, empty=0) + 104 <fs/namei.c:146>

vmlinux getname() + 16 <fs/namei.c:206>

vmlinux do_sys_open(dfd=-100) + 272 <fs/open.c:1013>

vmlinux C_SYSC_openat() <fs/compat.c:1101>

vmlinux compat_Sys_openat() + 8 <fs/compat.c:1099>

vmlinux el0_svc() + 48 <arch/arm64/kernel/entry.S:860>

== 栈结束 ==

对应汇编指令:

行号 地址 指令 提示

lib/strncpy_from_user.c

39 : FFFFFFFF800839C4BC: MOV X3, #0 ; strncpy_from_user() 参数 2 可能有问题

arch/arm64/include/asm/word-at-a-time.h

34 : FFFFFFFF800839C4DC: B 0xFFFFFFFF800839C500

arch/arm64/include/asm/uaccess.h

205 : FFFFFFFF800839C500:

FFFFFFF800839C504: ADD X6, X1, X3

lib/strncpy_from_user.c

43 : FFFFFFFF800839C50C: LDR X8, [X6] ; 进程停止在这里

当时的寄存器值:

```
X0: FFFFFFFC0AF30A01C, X1: 00000000F0A7FFCC, X2: 000000000000FE4, X3: 000000000000030
X4: 000000000000FB4, X5: 000000000000000, X6: 00000000F0A7FFFC, X7: 000000000000000
X8: 616572E6769666E, X9: FEFEFEFEFEFEFEF, X10: 000000000000000, X11: 00000000FFCEF35C
X12: 00000000FFCEEC8, X13: 00000000FFCEEC68, X14: 00000000F3E7C86D, X15: 000000000000000
X16: FFFFFFFF80082261C0, X17: 000000000000000, X18: 000000000000000, X19: FFFFFFFC0AF30A000
X20: 00000000F0A7FFCC, X21: FFFFFFFF800946C000, X22: 000000000000000, X23: 000000000000000
X24: 000000000000011, X25: 0000000000000186, X26: 0000000000000142, X27: FFFFFFFF8008DF3000
X28: FFFFFFFC015A24000, X29: FFFFFFFC015A27DF0, X30: FFFFFFFF80081DF44C, SP: FFFFFFFC015A27DF0
PC: FFFFFFFF800839C50C
```

是在 `do_strncpy_from_user()` 函数里 KE, X6=0xF0A7FFFC, 这个地址是 user space 传下来的。比较奇怪, 在 `do_strncpy_from_user` 函数里即使 `source address` 是错误的, 也不会崩溃, 最多返回没复制的字节数。

我们需要从 kernel log 看个究竟:

```
[58878.250060] (8)[16831:media.life]Unhandled fault: alignment fault (0x96000021) at 0x00000000f0a80000
```

看到是发生对齐错误, X6 是 4 字节对齐, 但 X6 的值和 log 里 0xf0a80000 对不上, 刚好差 4 字节, 而且 0xf0a80000 是对齐的, 这是怎么回事?

没有印出 `SCTLR_EL1`, 不清楚对其使能是否打开 (kernel 默认是关闭的), 有 2 种情况是无视对其检查开关的:

- 互斥访问指令, 比如 `LDREX`、`STREX` 等指令
- 访问的内存地址是 device memory 的。

在这里是第 2 种情况, 因此怀疑 0xf0a80000 是 device memory。如何确认内存地址是否是 device memory 呢? 有完整的 `ramdump`, 先看下进程的 `maps`:

- `task.maps` 0xFFFFFFFF0293F6000, 其中 0xFFFFFFFF0293F6000 为 task 对应的 `task_struct` 地址

```
FFFFFFFF0557B3270|00000000F09FF000--00000000F0A00000|00003000|rw-p|03:16|2705|/vendor/lib/libfbc.so
FFFFFFFF0097C2AF8|00000000F0A00000--00000000F0A80000|00003000|rw-p|00:00|0|[anon:libc_malloc]
FFFFFFFF055617EA0|00000000F0A80000--00000000F0A81000|00044000|r--s|00:10|14460|/dri/renderD128
FFFFFFFF055617000|00000000F0A81000--00000000F0A82000|0001E000|rw-s|00:10|14460|/dri/renderD128
FFFFFFFF081FEA888|00000000F0A82000--00000000F0A84000|00000000|rw-p|00:05|24445162|dev/ashmem/dalvik-indirect ref table
```

可以看到 0xf0a80000 刚好是 /dri/renderD128, 是 kernel driver mapping 上去的, 经过确认, 属于 GPU driver mapping 的。然后需要我们手动推导页表, 看 `page` 属性是否为 device memory。先获取 `task->mm.pgd`:

```
((struct thread_info)0xFFFFFFFF015A24000).task = 0xFFFFFFFF0293F6000-> (
state = 0x0,
stack = 0xFFFFFFFF015A24000 = end+0x400BD4C000,
.....
mm = 0xFFFFFFFF03DD3B3C0 = end+0x40340633C0 -> (
mmap = 0xFFFFFFFF03F2C5888 = end+0x40355ED888,
mm_rb = (rb_node = 0xFFFFFFFF02CB29158 = end+0x4022E51158),
vmacache_seqnum = 0x023A,
```

```

get_unmapped_area = 0xFFFFFFFF80081B2470 = arch_get_unmapped_area_topdown,

mmap_base = 0xF5F51000,

mmap_legacy_base = 0x0,

task_size = 0x0000000100000000,

highest_vm_end = 0xFFFFF1000,

pgd_ = 0xFFFFFFFFC05EDB8000,

```

然后按照 datasheet 里做一次 page table walk, 可以算出:

```

va = F0A7FFFC -> pa = 0xA61A1FFC

pgd index = 3

pmd index = 0x185

pte index = 0x7f

pgd[3] = C8699003 -> 0xFFFFFFFFC088699000

pmd[0x185] = 0xE716B003 -> 0xFFFFFFFFC0A716B000

pte[0x7f] = 00E80000A61A1F53 -> 0xFFFFFFFFC0661A1000

00E80000A61A1F53 对应的属性:

SH=inner shareable

AP=RW

AttrIdx=4

va = F0A80000 -> pa = 0x8964F000

pgd index = 3

pmd index = 0x185

pte index = 0x80

pgd[3] = C8699003 -> 0xFFFFFFFFC088699000

pmd[0x185] = 0xE716B003 -> 0xFFFFFFFFC0A716B000

pte[0x80] = 016000008964FFC3 -> 0xFFFFFFFFC04964F000

016000008964FFC3 对应的属性:

SH=inner shareable

AP=R0

AttrIdx=0

```

其中 attrindex 为内存属性, 在 kernel 里将 000 设定为 device memory:

```

#define MT_DEVICE_nGnRnE 0

#define pgprot_noncached(prot) __pgprot_modify(prot, PTE_ATTRINDX_MASK, PTE_ATTRINDX(MT_DEVICE_nGnRnE) | PTE_PXN | PTE_UXN)

```

因此明显是 device memory 的。

为何 kernel driver 会意外 copy 到 device memory? 这是因为 kernel 为了效率, 每一次复制使用 unsigned long (aarch64 下是 8 字节), 因此有可能超出有效范围:

```
static inline long do_strncpy_from_user(char *dst, const char __user *src, long count, unsigned long max)
{
    const struct word_at_a_time constants = WORD_AT_A_TIME_CONSTANTS;

    long res = 0;

    .....

    if (IS_UNALIGNED(src, dst))
        goto byte_at_a_time;

    while (max >= sizeof(unsigned long)) {
        unsigned long c, data;

        /* Fall back to byte-at-a-time if we get a page fault */

        unsafe_get_user(c, (unsigned long __user *) (src+res), byte_at_a_time);
    }
}
```

如果后面的 page 刚好是不可访问的，那么会跳转到 byte_at_a_time 里一个一个字节复制，但凑巧的是后面的 page 是 device memory，字节发生 alignment fault 了。

是否是 driver 有问题呢？正常情况下 DRAM memory (PA=0x8964F000 是 DRAM) 不应该设定为 device memory，而是设定为 non-cachable 的。因此需要检查 GPU driver。不过 kernel 也不应该因为 user space program 出错而崩溃。

经过网络搜索，发现已有同样的问题：

- mainline 讨论的 mail thread: <https://patchwork.kernel.org/patch/8320941/>
- patch: <https://patchwork.kernel.org/patch/8311781/>

后面发现 kernel4.5 就包含这个 patch:

```
commit 52d7523d84d534c241ebac5ac89f5c0a6cb51e41
Author: EunTaik Lee <eun.taik.lee@samsung.com>
Date: Tue Feb 16 04:44:35 2016 +0000

    arm64: mm: allow the kernel to handle alignment faults on user accesses

$git describe --contains 52d7523d84d534c241ebac5ac89f5c0a6cb51e41

v4.5-rc5~11^2
```

根本原因

不能用 strncpy_from_user 复制设备内存。

解决方法:

patch: <https://patchwork.kernel.org/patch/8311781/>

结语:

需要对 ARM load/store 异常逻辑熟悉，熟悉 page table walk，才能想到 root cause

19. failed to come online 引起 KE

问题背景:

手机重启(kernel panic)

分析过程:

用 GAT 解开 db, 并结合对应的 vmlinux (该文件必须和 db 一致, 具体请看 FAQ06985), 利用工具 (E-Consulter.jar) 分析 (也可以参考 FAQ13941), 解析出来的调用栈如下:

报告解读: MediaTek On-Line> Quick Start> E-Consulter 之 NE/KE 分析报告解读> KE 分析报告

详细描述: 写数据 0x8C1D90C25566700E 到错误的地址 (0x0000000000000000), 请结合崩溃进程调用栈检查相关代码

平台 : MT6755

版本 : alps-mp-m0.mp7/user build

异常时间: 179278.424868 秒, Thu Jun 30 09:28:10 CST 2016

== CPU 信息 ==

崩溃 CPU 信息:

CPU6: 进程名: WorkerPool/3052, 进程标识符(pid): 30525

本地调用栈:

vmlinux memcpy() + 272 <arch/arm64/lib/memcpy.S:183>

vmlinux __zs_map_object() + 112 <mm/zsmalloc.c:819>

vmlinux zs_map_object(pool=0xFFFFF0C0A337C000, handle=84945057873922, mm=ZS_MM_R0) + 404 <mm/zsmalloc.c:1143>

vmlinux zram_decompress_page(mem=0xFFFFF0C0434F0000, index=62286, zram=0xFFFFF0C0434F0000, zram=0x000000000000F34E) + 220 <drivers/block/zram/zram_drv.c:776>

vmlinux zram_bvec_read() + 1472 <drivers/block/zram/zram_drv.c:826>

vmlinux zram_bvec_rw(zram=0xFFFFF0C0B0F93840, bvec=0xFFFFF0C0182AFA40, index=62286, offset=0) + 1540 <drivers/block/zram/zram_drv.c:1067>

vmlinux __zram_make_request() + 428 <drivers/block/zram/zram_drv.c:1311>

vmlinux zram_make_request(bio=0xFFFFF0C0573C99C0) + 556 <drivers/block/zram/zram_drv.c:1341>

vmlinux generic_make_request(bio=0xFFFFF0C0573C99C0) + 136 <block/blk-core.c:1924>

vmlinux submit_bio(rw=0, bio=0xFFFFF0C0573C99C0) + 128 <block/blk-core.c:2005>

vmlinux swap_readpage(page=0xFFFFF0BE01CB9480) + 352 <mm/page_io.c:390>

vmlinux read_swap_cache_async(entry=0x000000000000F34E, gfp_mask=131290, vma=0xFFFFF0C0AD9E0840, addr=3304787968) + 220 <mm/swap_state.c:376>

vmlinux swapin_readahead(entry=0x000000000000F34E, gfp_mask=131290, vma=0xFFFFF0C0AD9E0840, addr=3304787968) + 316 <mm/swap_state.c:492>

vmlinux do_swap_page() + 1260 <mm/memory.c:2446>

vmlinux handle_pte_fault() + 1400 <mm/memory.c:3223>

vmlinux __handle_mm_fault() + 1508 <mm/memory.c:3341>

vmlinux handle_mm_fault(mm=0xFFFFF0C012A2BC00, vma=0xFFFFF0C0AD9E0840, address=3304787968, flags=168) + 1628 <mm/memory.c:3370>

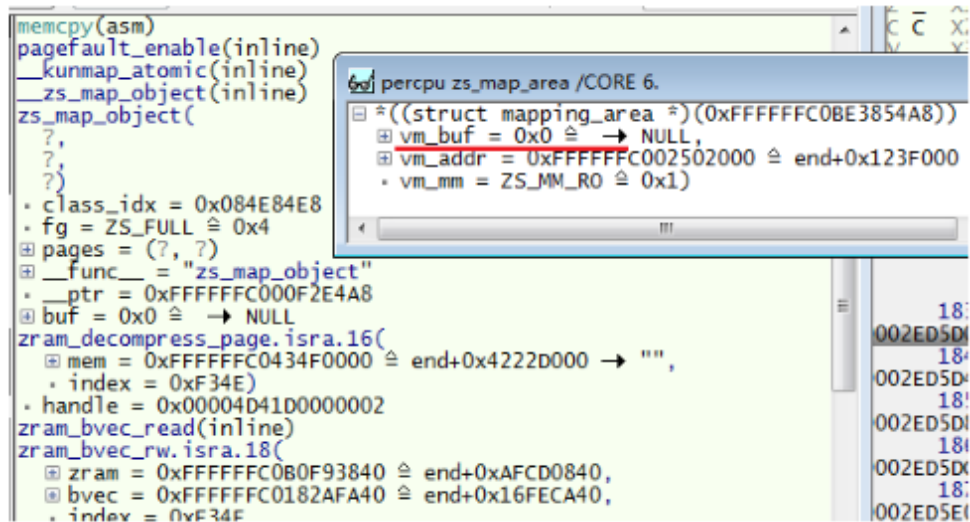
vmlinux __do_page_fault() + 412 <arch/arm64/mm/fault.c:185>

vmlinux do_page_fault(addr=3304790056, esr=2449473543, regs=0xFFFFF0C0182AFED0) + 564 <arch/arm64/mm/fault.c:249>

vmlinux do_mem_abort() + 56 <arch/arm64/mm/fault.c:461>

vmlinux el0_sync() + 640 <arch/arm64/kernel/entry.S:622>

通过 trace32 分析 backtrace 是 area->vm_buf 为 NULL 导致:



查看 code, area->vm_buf 在 up 时 get free page, down 时 free:

```
static inline int __zs_cpu_up(struct mapping_area *area)
{
    /*
     * Make sure we don't leak memory if a cpu UP notification
     * and zs_init() race and both call zs_cpu_up() on the same cpu
     */
    if (area->vm_buf)
        return 0;
    area->vm_buf = (char *) get_free_page(GFP_KERNEL);
    if (!area->vm_buf)
        return -ENOMEM;
    return 0;
}

static inline void __zs_cpu_down(struct mapping_area *area)
{
    if (area->vm_buf)
        free_page((unsigned long)area->vm_buf);
    area->vm_buf = NULL;
}
```

而从 kernel log 中看到:

```
[179276.969805] (1) [111:hps_main]CPU6: failed to come online

[179276.971539] (1) [111:hps_main]CPU2: shutdown

[179276.974974] (0) [111:hps_main]CPU1: shutdown

[179276.977142] (0) [111:hps_main][CPUHVFS] [01c6396e] cluster0 off, pause = 0x0, RSV4: 0x5370d670

[179276.979905] -(6) [0:swapper/6]CPU6: update cpu capacity 1024
```

CPU6: failed to come online 但是后面又有起来, 从其它数据看起来正常, 不太像是 memory crash. 怀疑是因为 CPU6: failed to come online 将 area->vm buf=NULL, 而后面又有起来但是并没有按正常 follow 设置 area->vm buf.

为什么 CPU6: failed to come online 呢? 看看 failed to come online 之前发生了什么。

查看 kernel log:

```
[179275.970671] -(6)[0:swapper/6]Detected VIPT I-cache on CPU6 //准备开启 CPU6

[179275.993162] (0)[11734:kworker/0:3]<ALS/PS> als data[53]

[179276.192932] (0)[20830:kworker/0:4]<ALS/PS> als data[53]

[179276.395234] (0)[20830:kworker/0:4]<ALS/PS> als data[0]

[179276.594651] (0)[23482:kworker/0:0]<ALS/PS> als data[0]

[179276.710449] (2)[289:logd.auditd]type=1400 audit(1467250068.100:422199): avc: denied { search } for pid=30905 comm="pool-2
-protocal" name="mtkcooler" dev="proc" ino=4026533878 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:object_r:proc_mtkcoole
r:s0 tclass=dir permissive=0

[179276.711557] (2)[289:logd.auditd]type=1400 audit(1467250068.100:422200): avc: denied { search } for pid=30905 comm="pool-2
-protocal" name="1" dev="proc" ino=5261498 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:init:s0 tclass=dir permissive=0

[179276.711677] (2)[289:logd.auditd]type=1400 audit(1467250068.100:422201): avc: denied { search } for pid=30905 comm="pool-2
-protocal" name="2" dev="proc" ino=5261499 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive
=0

[179276.711782] (2)[289:logd.auditd]type=1400 audit(1467250068.100:422202): avc: denied { search } for pid=30905 comm="pool-2
-protocal" name="3" dev="proc" ino=5261500 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive
=0

[179276.711884] (2)[289:logd.auditd]type=1400 audit(1467250068.100:422203): avc: denied { search } for pid=30905 comm="pool-2
-protocal" name="5" dev="proc" ino=5261501 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive
=0

[179276.711996] (2)[289:logd.auditd]type=1400 audit(1467250068.100:422204): avc: denied { search } for pid=30905 comm="pool-2
-protocal" name="7" dev="proc" ino=5261502 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive
=0

[179276.712097] (2)[289:logd.auditd]type=1400 audit(1467250068.100:422205): avc: denied { search } for pid=30905 comm="pool-2
-protocal" name="8" dev="proc" ino=5261503 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive
=0

[179276.712198] (2)[289:logd.auditd]type=1400 audit(1467250068.100:422206): avc: denied { search } for pid=30905 comm="pool-2
-protocal" name="9" dev="proc" ino=5261504 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive
=0
```

[179276.712297] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422207): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="10" dev="proc" ino=5261505 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

[179276.712398] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422208): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="11" dev="proc" ino=5261506 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

[179276.712413] (4) [30905:pool-2-protocol] audit: audit_lost=352917 audit_rate_limit=20 audit_backlog_limit=64

[179276.712426] (4) [30905:pool-2-protocol] audit: rate limit exceeded

[179276.712500] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422209): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="12" dev="proc" ino=5261507 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

[179276.712601] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422210): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="14" dev="proc" ino=5261508 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

[179276.712704] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422211): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="15" dev="proc" ino=5261509 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

[179276.712807] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422212): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="16" dev="proc" ino=5261510 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

[179276.712908] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422213): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="17" dev="proc" ino=5261511 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

[179276.713013] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422214): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="18" dev="proc" ino=5261512 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

[179276.713116] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422215): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="19" dev="proc" ino=5261513 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

[179276.713216] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422216): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="20" dev="proc" ino=5261514 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0

```

[179276.713316] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422217): avc: denied { search } for pid=30905 comm="pool-2
-protocol" name="22" dev="proc" ino=5261515 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissiv
e=0

[179276.713417] (2) [289:logd.auditd] type=1400 audit(1467250068.100:422218): avc: denied { search } for pid=30905 comm="pool-2
-protocol" name="23" dev="proc" ino=5261516 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissiv
e=0

[179276.792945] (0) [23482:kworker/0:0] <ALS/PS> als data[0]

[179276.803929] AEE_MONITOR_SET[status]: 0x1

[179276.914829] (4) [236:mtk_charger_hv_] array_size = 16

[179276.969805] (1) [111:hps_main] CPU6: failed to come online //大约一秒无回应, 因此判断 CPU6 fail

[179276.971539] (1) [111:hps_main] CPU2: shutdown

[179276.974974] (0) [111:hps_main] CPU1: shutdown

[179276.977142] (0) [111:hps_main] [CPUHVFS] [01c6396e] cluster0 off, pause = 0x0, RSV4: 0x5370d670

[179276.979905] -(6) [0:swapper/6] CPU6: update cpu_capacity 1024

```

Linux kernel 的机制, 开 CPU timeout 是 1 秒, 从 log 看起来, CPU6 已经 timeout, 之后被判断为 boot fail, 此时如果后面被开启, 会有问题。

而从 log 看到, 在 cpu6 开启的过程中大量打印 auditd 的信息, 极有可能是因为这里导致的 timeout。

而这些 audit 的 log 打印的原因是 M 上有新加 MLS constrain, 以上报的 selinux 权限问题, 是因为 MLS constrain 没过, MLS 是对 TEAC 的加强。

MLS 准则: 在 /system/sepolicy/mls 这支文件中:

```

# Read operations: Subject must dominate object unless the subject

# or the object is trusted.

mlsconstrain dir { read getattr search }

(t2 == app_data_file or l1 dom l2 or t1 == mltrustedsubject or t2 == mltrustedobject);

```

添加新的 MLS 定义到 policy, 例如:


```
[179276.712097] (2)[289:logd.auditd]type=1400 audit(1467250068.100:422205): avc: denied { search } for pid=30905 comm="pool-2-protocol" name="8" dev="proc" ino=5261503 scontext=u:r:untrusted_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0
```

可在 `untrusted_app.te` 中添加: `typeattribute untrusted_app mltrustedsubject;` 也可以将 `target` 定义为 `mltrustedobject`.

添加新的 MLS 定义到 `policy` 后问题不再复现。

根本原因:

selinux 权限问题导致开 CPU timeout。

解决方法:

添加新的 MLS 定义到 `policy`。

结语:

对于报 `failed to come online` 的问题应该查看 `Detected VIPT I-cache on CPU` 到 `failed to come online` 之间系统发生了什么。

附录

相关 FAQ