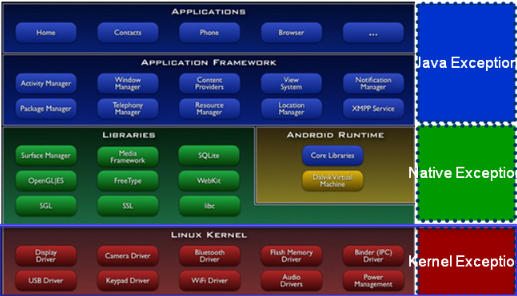
**基础篇: 通过log分析KE**

**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是美国人比较常有的口语。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？就要看在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了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coredump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，0xFFFFFFC000000000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 0000007fffffffff 512GB user  
ffffff8000000000 ffffffbbfffeffff ~240GB vmalloc  
ffffffbbffff0000 ffffffbbffffffff 64KB [guard page]  
ffffffbc00000000 ffffffbdffffffff 8GB vmemmap  
ffffffbe00000000 ffffffbffbbfffff ~8GB [guard]  
ffffffbffbc00000 ffffffbffbdfffff 2MB early con I/O space  
ffffffbffbe00000 ffffffbffbffffff 2MB PCI I/O space  
ffffffbffc000000 ffffffbfffffffff 64MB modules  
ffffffc000000000 ffffffffffffffff 256GB kernel logical memory map

**>= kernel-3.18 && < kernel-4.6**

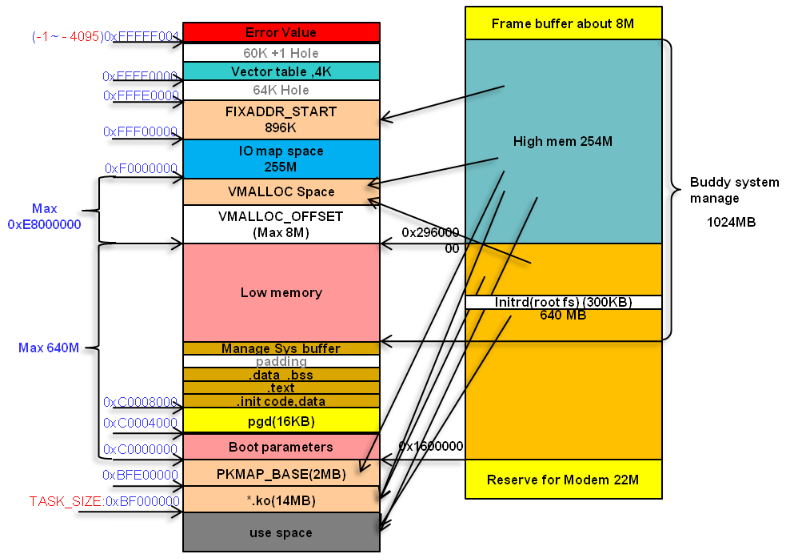
Start End Size Use  
-----------------------------------------------------------------------  
0000000000000000 0000007fffffffff 512GB user  
ffffff8000000000 ffffffbdfffeffff ~247GB vmalloc  
ffffffbdffff0000 ffffffbdffffffff 64KB [guard page]  
ffffffbe00000000 ffffffbfbfffffff 7GB vmemmap  
ffffffbfc0000000 ffffffbffbdfcfff 957MB [guard]  
ffffffbffbdfd000 ffffffbffbdfefff 8KB fixed mappings  
ffffffbffbdff000 ffffffbffbffffff ~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 ffffffbdbfff0000 246GB vmalloc  
ffffffbdbfff0000 ffffffbdc0000000 64KB [guard page]  
ffffffbdc0000000 ffffffbfc0000000 8GB vmemmap  
ffffffbfc0000000 ffffffbffe7fd000 ~1G [guard]  
ffffffbffe7fd000 ffffffbffec00000 4108KB fixed mappings  
ffffffbffec00000 ffffffbffee00000 2MB [guard]  
ffffffbffee00000 ffffffbfffe00000 16MB PCI I/O  
ffffffbfffe00000 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](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 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](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 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\_llk; /\* 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](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 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](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 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 atfer 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

#### 

64bit kernel：

原生的kernel，BUG()是直接调用panic()的：

#### 

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）

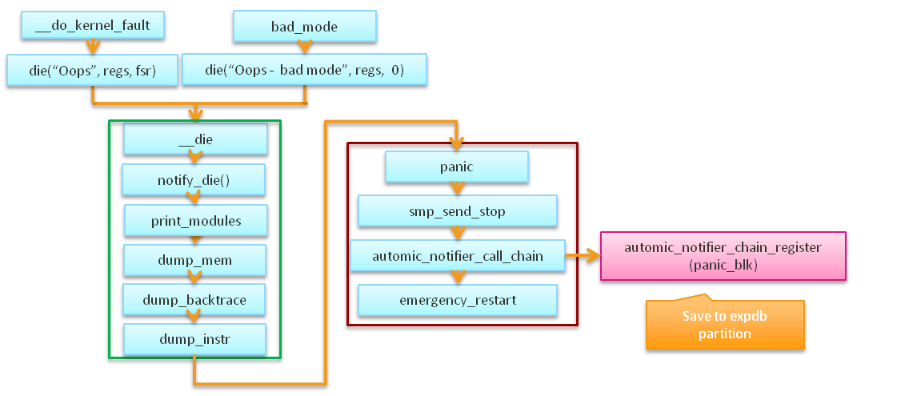
#### 

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:kworker/u8:3]Unable to handle kernel paging request at virtual address 0000dead

#### 六：die()流程

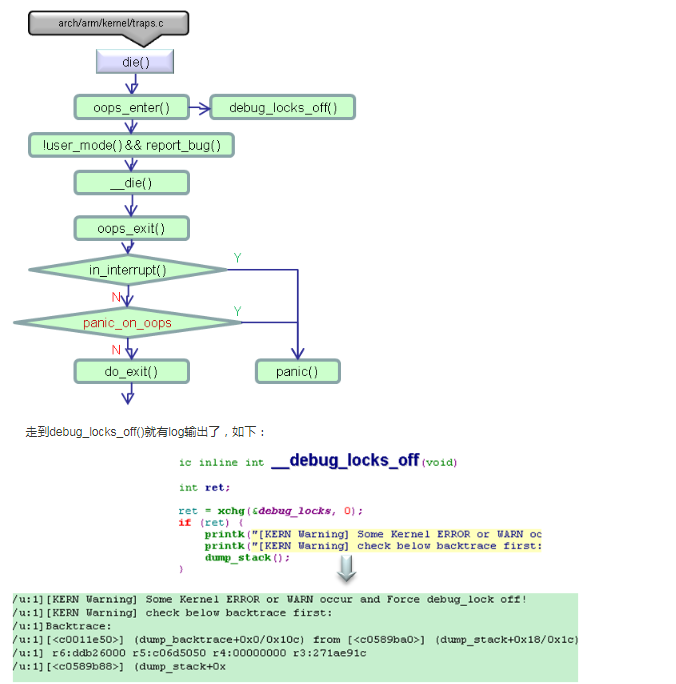
经过前面的流程，走到了die()函数，该函数主要输出便于调试的寄存器信息/堆栈信息等重要资料，我们通过log分析KE就是分析这些资料，因此要知道整个流程。die() => panic()的大致流程如下：



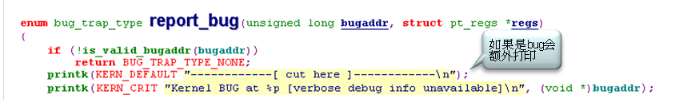
在学习这些流程时，建议结合代码和KE的log一起看，你就知道log里那些信息在代码哪处打印出来的了。

**die()总流程**

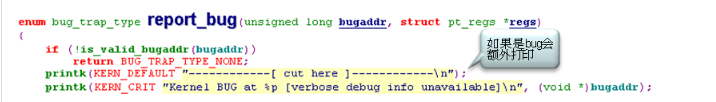
    先从die()入手，看下die()总流程：



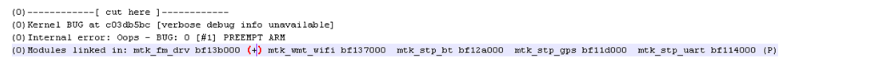
  如果这个异常是代码里调用BUG()/BUG\_ON()引起，那么有额外log说明：



  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：

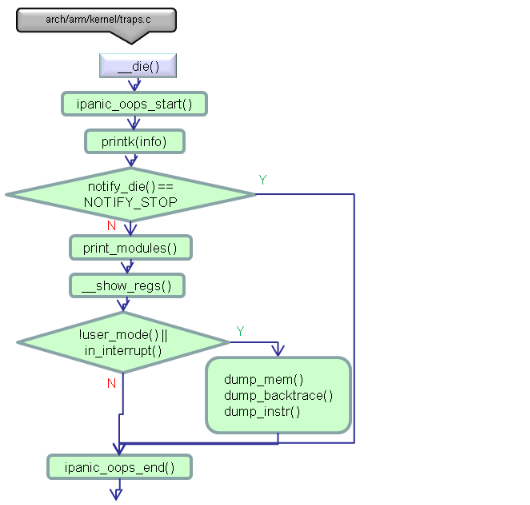


输出的log大致如下：



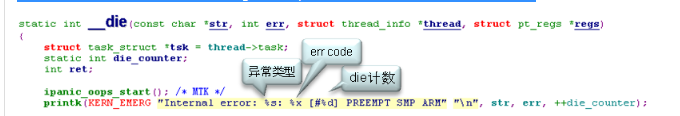
**\_\_die()流程**

    绝大部分的关键信息是由\_\_die()函数输出的，流程如下：

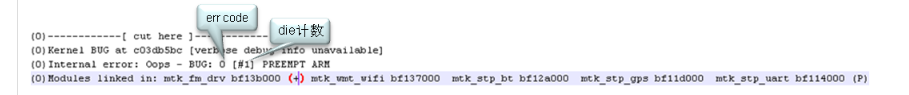


**异常类型信息**

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：



输出的信息大致如下：

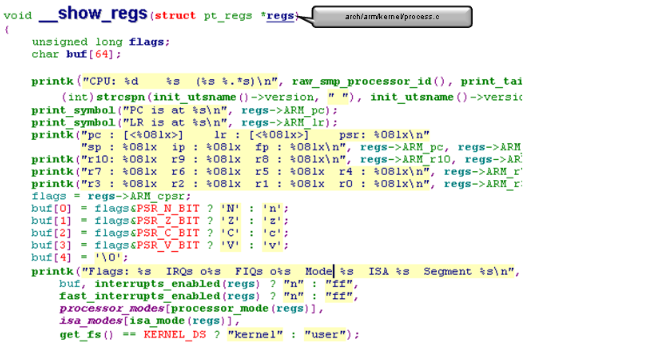


**module信息**

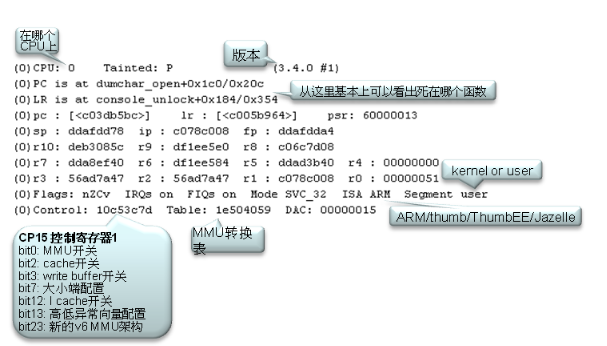
接下来是module信息，不过我们不建议使用module，这边也不打算介绍了。

**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）：

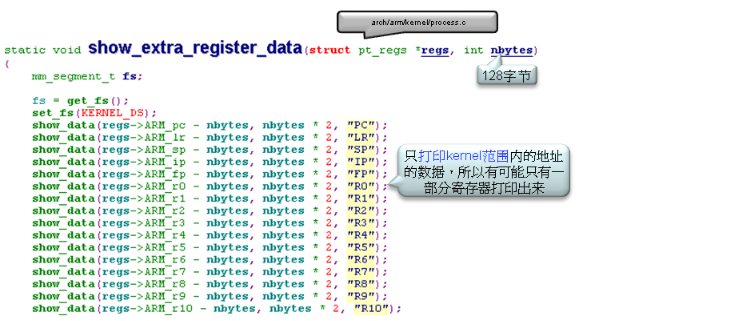


输出的信息大致如下：

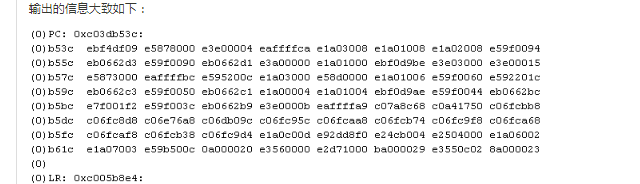


**寄存器附近的内存**

有助于我们分析问题的内存信息，问题很可能就出在里面。

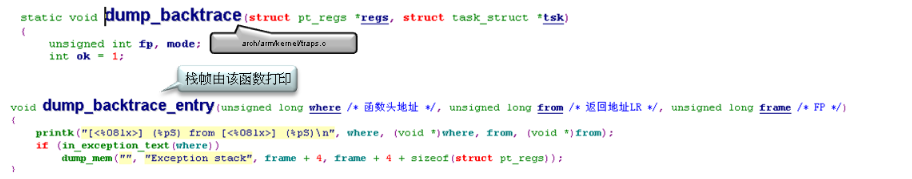


输出的信息大致如下：



**调用栈**

有时问题可以直接从调用栈看出来，由此可见调用栈是多么重要。



输出的信息大致如下：



**PC附近指令**

可以看到PC附近的指令：



**分析log**

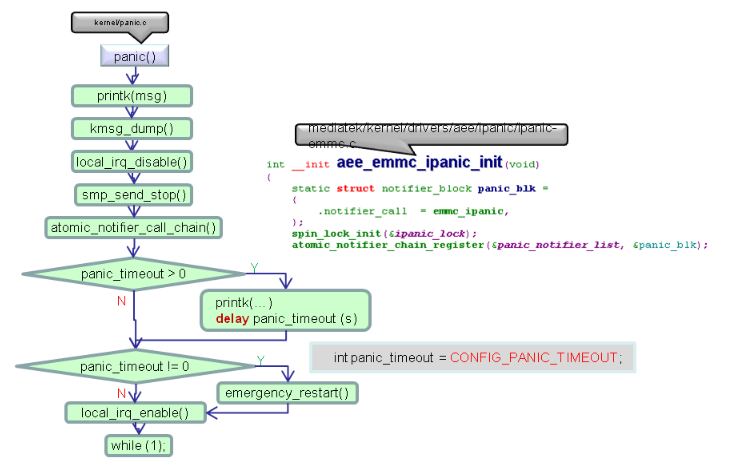
    到这里die()函数就完成了它的使命，将重要信息输出来了。接下来你要如何调试呢？这个就看个人的功力了，你可以：

* 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：



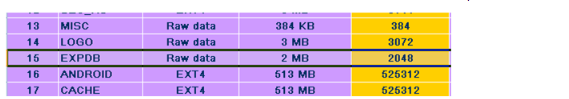
因此我们也可以通过搜索关键字Kernel panic查找是否有panic发生。

**panic通知链**

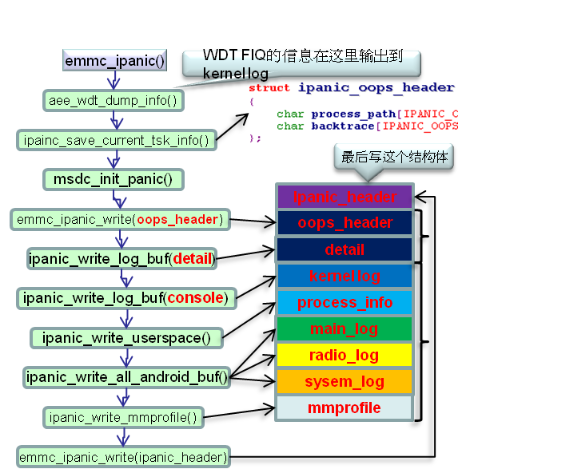
    panic()会调用栈通知链上的回调函数同时感兴趣的模块，比如我们的aee注册了回调函数，用于保存kernel log/mini dump等关键信息，并将其保存到emmc的expdb分区，等等重启后将其回读并保存成KE db。

**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：



流程大致如下（版本不停演进，可能有很大变化，仅供参考）：



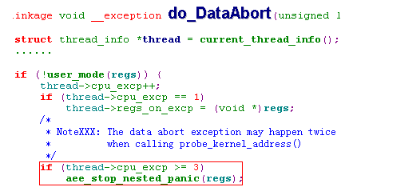
重启后，aee将回读aeedb分区资料并转化为KE db。

#### 八：nested panic

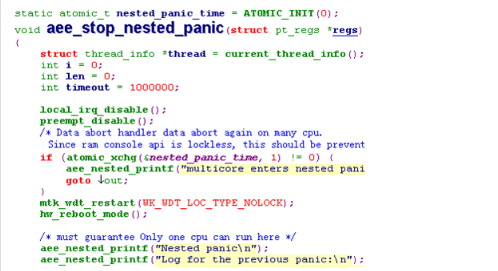
有时die()/panic()流程不一定能正常走完，可能走到某一步又发生了异常，则就形成了嵌套，这种情况，我们一般不会关注后面的异常，而是关注最开始的那个异常。

    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：

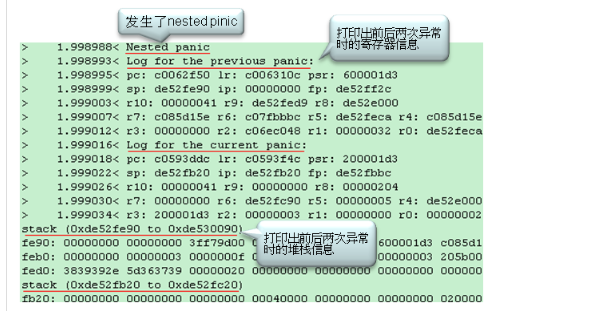
* do\_PrefetchAbort()
* do\_DataAbort()
* do\_undefinstr()



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：



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）：



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。

**KE 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](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 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](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 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](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 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）/ 0xFFFFFFC000000000（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](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> crash使用教程

#### 六：trace32调试

Trace32有专门的文章介绍，请参考：

* [MediaTek On-Line](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 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](https://online.mediatek.com/_layouts/15/mol/topic/ext/Topic.aspx?mappingId=c7b85b31-2ee8-42c3-acdd-fa538a77abcb)> lk 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

#### 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



**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



**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 vmap\_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**

#### 

**代码位置：**

[**kernel-3.10**](http://10.16.20.15/l0mp8/xref/kernel-3.10/)**/**[**drivers**](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/)**/**[**base**](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/)**/**[**power**](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/)**/**[**main.c**](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c)

 /\*\*

[411](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "411) \* dpm\_wd\_handler - Driver suspend / resume watchdog handler.

[412](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "412) \*

[413](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "413) \* Called when a driver has timed out suspending or resuming.

[414](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "414) \* There's not much we can do here to recover so BUG() out for

[415](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "415) \* a crash-dump

[416](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "416) \*/

[417](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "417)**static** **void** [**dpm\_wd\_handler**](http://10.16.20.15/l0mp8/s?refs=dpm_wd_handler&project=kernel-3.10)(**unsigned** **long** **[data](http://10.16.20.15/l0mp8/s?refs=data&project=kernel-3.10)**)

[418](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "418){

[419](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "419) **struct** [**dpm\_watchdog**](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c#dpm_watchdog) \***[wd](http://10.16.20.15/l0mp8/s?refs=wd&project=kernel-3.10)** = (**void** \*)[data](http://10.16.20.15/l0mp8/s?defs=data&project=kernel-3.10);

[420](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "420) **struct** [device](http://10.16.20.15/l0mp8/s?defs=device&project=kernel-3.10) \***[dev](http://10.16.20.15/l0mp8/s?refs=dev&project=kernel-3.10)** = [wd](http://10.16.20.15/l0mp8/s?defs=wd&project=kernel-3.10)->**[dev](http://10.16.20.15/l0mp8/s?refs=dev&project=kernel-3.10)**;

[421](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "421) **struct** [task\_struct](http://10.16.20.15/l0mp8/s?defs=task_struct&project=kernel-3.10) \***[tsk](http://10.16.20.15/l0mp8/s?refs=tsk&project=kernel-3.10)** = [wd](http://10.16.20.15/l0mp8/s?defs=wd&project=kernel-3.10)->**[tsk](http://10.16.20.15/l0mp8/s?refs=tsk&project=kernel-3.10)**;

[422](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "422)

[423](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "423) [dev\_emerg](http://10.16.20.15/l0mp8/s?defs=dev_emerg&project=kernel-3.10)([dev](http://10.16.20.15/l0mp8/s?defs=dev&project=kernel-3.10), "\*\*\*\* DPM device timeout \*\*\*\*\n");

[424](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "424) [show\_stack](http://10.16.20.15/l0mp8/s?defs=show_stack&project=kernel-3.10)([tsk](http://10.16.20.15/l0mp8/s?defs=tsk&project=kernel-3.10), [NULL](http://10.16.20.15/l0mp8/s?defs=NULL&project=kernel-3.10));

[425](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "425)

[426](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "426) [BUG](http://10.16.20.15/l0mp8/s?defs=BUG&project=kernel-3.10)();

[427](http://10.16.20.15/l0mp8/xref/kernel-3.10/drivers/base/power/main.c" \l "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** 0xffffffbe020a4cf8 objects=12 used=12 fp=0x (null) flags=0x0080  
<6>[ 492.559085]-(0)[1163:system\_server]**INFO: Object** 0xffffffc0553a93c0 @offset=960 fp=0xffffffc0553a9640  
<6>[ 492.559085]  
<6>[ 492.559116]-(0)[1163:system\_server]**Bytes** b4 ffffffc0553a93b0: 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** ffffffc0553a93c0: 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** ffffffc0553a93d0: 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** ffffffc0553a93e0: 6b 6b 6b 6b 6b 6b 6b 6b 6b 69 6b 6b 6b 6b 6b 6b kkkkkkkkkikkkkkk  
<6>[ 492.559213]-(0)[1163:system\_server]**Object** ffffffc0553a93f0: 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** ffffffc0553a9400: 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** ffffffc0553a9410: 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** ffffffc0553a9420: 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** ffffffc0553a9430: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5 kkkkkkkkkkkkkkk.  
<6>[ 492.559333]-(0)[1163:system\_server]**Redzone** ffffffc0553a9440: bb bb bb bb bb bb bb bb ........  
<6>[ 492.559358]-(0)[1163:system\_server]**Padding** ffffffc0553a94c0: 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** ffffffc0553a94d0: 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** ffffffc0553a94e0: 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** ffffffc0553a94f0: 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][<ffffffc0000fefe4>] pm\_suspend+0x3c4/0x5c8  
<2>[ 492.559982]-(0)[1163:system\_server][<ffffffc0000fd5b0>] 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 = ffffffc03c136000  
<2>[ 492.561229]-(0)[1163:system\_server][0000dead] \*pgd=00000000802de003, \*pud=00000000802de003, \*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: ffffffc03b158000 ti: ffffffc02f018000 task.ti: ffffffc02f018000  
<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**](http://10.16.20.15/l0mp8/xref/kernel-3.10/)

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**+0x64/0xcc

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>] el1\_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 = ffffffc00007d000  
[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\_tbl\_map\_single/swiotlb\_tbl\_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.

|--------------------------------------------------64M-----------------------------------------------------------|  
|-----256K-------|-----256K-------|-----256K-------|..........|-----256K-------|-----256K-------|-----256K-------|

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\_tbl(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\_tbl\_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 0x7ffffff9 [ 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: ffffffc000e142f0 ti: ffffffc000e00000 task.ti: ffffffc000e00000 [ 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: ffffffc0afcea000 ti: ffffffc0afcec000 task.ti: ffffffc0afcec000 <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=0xFFFFFFC00E049000) + 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=0xFFFFFFC0460CF000, bvec=0xFFFFFFC037C75C68, 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=0xFFFFFFC037C75C00) + 400 <drivers/staging/zram/zram\_drv.c:1010>  
vmlinux generic\_make\_request(bio=0xFFFFFFC037C75C00) + 132 <block/blk-core.c:1844>  
vmlinux submit\_bio(rw=1, bio=0xFFFFFFC037C75C00) + 172 <block/blk-core.c:1936>  
vmlinux \_\_swap\_writepage(page=0xFFFFFFBC025F6290) + 552 <mm/page\_io.c:321>  
vmlinux swap\_writepage(page=0xFFFFFFBC025F6290, wbc=0xFFFFFFC06712B4A8) + 44 <mm/page\_io.c:250>  
vmlinux pageout(page=0xFFFFFFBC025F6290, mapping=0xFFFFFFC000E84688) + 368 <mm/vmscan.c:492>  
vmlinux shrink\_page\_list(page\_list=0xFFFFFFC06712B690, zone=0xFFFFFFC000F36640, sc=0xFFFFFFC06712B8E8, ttu\_flags=TTU\_UNMAP) + 800 <mm/vmscan.c:993>  
vmlinux shrink\_inactive\_list(lruvec=0xFFFFFFC000F36A88, sc=0xFFFFFFC06712B8E8, lru=LRU\_INACTIVE\_ANON) + 608 <mm/vmscan.c:1476>  
vmlinux shrink\_list() + 428 <mm/vmscan.c:1807>  
vmlinux shrink\_lruvec(lruvec=0xFFFFFFC000F36A88, sc=0xFFFFFFC06712B8E8) + 1100 <mm/vmscan.c:2154>  
vmlinux shrink\_zone(sc=0xFFFFFFC06712B8E8) + 60 <mm/vmscan.c:2323>  
vmlinux shrink\_zones() + 128 <mm/vmscan.c:2468>  
vmlinux do\_try\_to\_free\_pages(zonelist=0xFFFFFFC000F37900, sc=0xFFFFFFC06712B8E8, shrink=0xFFFFFFC06712B8D8) + 240 <mm/vmscan.c:2535>  
vmlinux try\_to\_free\_pages(zonelist=0xFFFFFFC000F37900, 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=0xFFFFFFC047001840, flags=32976, node=-1) + 132 <mm/slub.c:1397>  
vmlinux new\_slab\_objects() + 104 <mm/slub.c:2171>  
vmlinux \_\_slab\_alloc(s=0xFFFFFFC047001840, 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 \_\_kmalloc(size=8192, flags=32976) + 372 <mm/slub.c:3278>  
vmlinux kmalloc() + 20 <include/linux/slub\_def.h:174>  
vmlinux kzalloc() + 20 <include/linux/slab.h:520>  
vmlinux cmdq\_rec\_realloc\_cmd\_buffer(handle=0xFFFFFFC0B1A5E800) + 36 <drivers/misc/mediatek/cmdq/cmdq\_record.c:51>  
vmlinux cmdq\_rec\_realloc\_cmd\_buffer() + 12 <drivers/misc/mediatek/cmdq/cmdq\_record.c:47>  
vmlinux cmdq\_append\_command(handle=0xFFFFFFC0B1A5E800, code=CMDQ\_CODE\_WRITE, argA=335634148, argB=1043319794) + 172 <drivers/misc/mediatek/cmdq/cmdq\_record.c:197>  
vmlinux cmdqRecWrite(handle=0xFFFFFFC0B1A5E800, value=1043319794, mask=4294967295) + 76 <drivers/misc/mediatek/cmdq/cmdq\_record.c:435>  
vmlinux disp\_gamma\_write\_lut\_reg(cmdq=0xFFFFFFC0B1A5E800, id=DISP\_GAMMA0, lock=0) + 452 <drivers/misc/mediatek/dispsys/mt6795/ddp\_gamma.c:110>  
vmlinux disp\_gamma\_set\_lut() + 136 <drivers/misc/mediatek/dispsys/mt6795/ddp\_gamma.c:153>  
vmlinux disp\_gamma\_io(module=DISP\_MODULE\_GAMMA, msg=1208252439) + 196 <drivers/misc/mediatek/dispsys/mt6795/ddp\_gamma.c:297>  
vmlinux dpmgr\_path\_user\_cmd(dp\_handle=0xFFFFFFC045E0F000, msg=1208252439, arg=547229475328, cmdqhandle=0xFFFFFFC0B1A5E800) + 380 <drivers/misc/mediatek/dispsys/mt6795/ddp\_manager.c:1124>  
vmlinux primary\_display\_user\_cmd() + 184 <drivers/misc/mediatek/videox/mt6795/primary\_display.c:4917>  
vmlinux mtk\_disp\_mgr\_ioctl() + 1008 <drivers/misc/mediatek/videox/mt6795/mtk\_disp\_mgr.c:2040>  
vmlinux disp\_unlocked\_ioctl() + 8 <drivers/misc/mediatek/videox/mt6795/mtk\_mira.c:21>  
vmlinux proc\_reg\_unlocked\_ioctl(file=0xFFFFFFC07209A100, arg=547229475328) + 76 <fs/proc/inode.c:252>  
vmlinux vfs\_ioctl() + 20 <fs/ioctl.c:43>  
vmlinux do\_vfs\_ioctl(filp=0xFFFFFFC07209A100, 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\_realloc\_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：

#### 

#### 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：

#### 

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

#### 

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  pvrsrvkm 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:

#### 

#### 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queuework：

|  |
| --- |
| <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queuework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=0xFFFFFFC0BD848E40) + 508 <kernel/smpboot.c:160> vmlinux kthread(\_create=0xFFFFFFC0BD848E80) + 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，然后看后是否是一个正常的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 [<ffffffc0003e2800>] 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](http://10.16.20.15/alps-mp-m0-mp9/xref/kernel-3.18/)/[mm](http://10.16.20.15/alps-mp-m0-mp9/xref/kernel-3.18/mm/)/[oom\_kill.c](http://10.16.20.15/alps-mp-m0-mp9/xref/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 mnld [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：**

当插着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 0xbeb2dbca 3199392714  
r10 0x0 0  
r11 0xb6475c20 3058129952  
r12 0x0 0  
sp 0xbeb2db58 0xbeb2db58  
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, 7002942884977337198, 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preemption\_check()  
 .macro restore\_user\_regs, fast = 0, offset = 0  
 bl preemption\_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 preemption\_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=0xFFFFFFC0E0CC7C88, 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 : FFFFFFC00032CC1C: LDR X3, [X1], #0x8 ; \_\_copy\_from\_user()参数 2可能有问题  
 当时的寄存器值:  
 X0: FFFFFFC0E0CC7C88, X1: 00000000E67A1FFC, X2: 0000000000000010, X3: 0000000000000002  
 X4: 00000000E67A2014, X5: FFFFFFC00101D000, X6: 0000000000000D1D, X7: 0000000000000D1A  
 X8: 0000000000000001, X9: FFFFFFC001265800, X10: FFFFFF80095E5000, X11: 0101010101010101  
 X12: 0000000000000038, X13: 00000000E4F83540, X14: 00000000F5BB69C7, X15: 0000000000000000  
 X16: FFFFFFC000120764, X17: 00000000E4F834C0, X18: 00000000E4F834B8, X19: 00000000E67A1FFC  
 X20: 0000000000000080, X21: 00000027F05179E0, X22: 0000000000000001, X23: 0000000000000001  
 X24: 0000000000000004, X25: 000000000000007F, X26: FFFFFFC0E0CC4000, X27: 0000000000000051  
 X28: 0000000000000080, X29: FFFFFFC0E0CC7C20, X30: FFFFFFC0008E5884, SP: FFFFFFC0E0CC7C20  
 PC: FFFFFFC00032CC1C

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 = ffffffc0f9f8c000  
[ 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：**

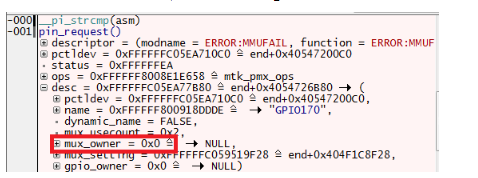
压力测试发生KE

**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\_strcmp(参数1=0, 参数2=0xFFFFFFC05D9396C0) + 24 <arch/arm64/lib/strcmp.S:77>  
 vmlinux pin\_request() + 244 <drivers/pinctrl/pinmux.c:120>  
 vmlinux pinmux\_enable\_setting(setting=0xFFFFFFC059519E40) + 456 <drivers/pinctrl/pinmux.c:429>  
 vmlinux pinctrl\_commit\_state(p=0xFFFFFFC059511400, state=0xFFFFFFC059519E00) + 180 <drivers/pinctrl/core.c:1012>  
 vmlinux pinctrl\_select\_state() + 20 <drivers/pinctrl/core.c:1066>  
 vmlinux ccm\_xxx\_disable\_vib(hap=0xFFFFFFC059515618) + 96 <drivers/misc/xxx\_vibcam/src/xxx\_vib8846.c:347>  
 vmlinux stop\_motor\_work(hap=0xFFFFFFC059515618) + 164 <drivers/misc/xxx\_vibcam/src/xxx\_vib8846.c:529>  
 vmlinux xxx\_vib\_stage\_enable(hap=0xFFFFFFC059515618) + 268 <drivers/misc/xxx\_vibcam/src/xxx\_vib8846.c:611>  
 vmlinux motor\_work\_func(work=0xFFFFFFC059515698) + 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:



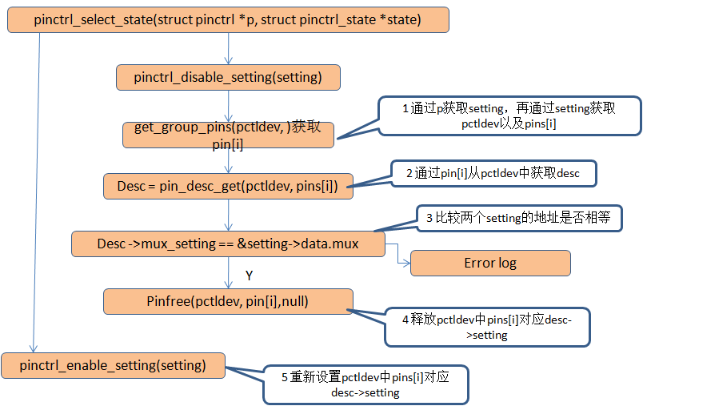
而desc是通过desc = pin\_desc\_get(pctldev, pin);获取的，而pctldev, pin是pinmux\_enable\_setting()传递给pin\_request() 的参数:

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跑完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(0x00000000F0A7FFFC)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=0x00000000F0A7FFCC, 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 : FFFFFF800839C4BC: MOV X3, #0 ; strncpy\_from\_user()参数 2可能有问题  
 arch/arm64/include/asm/word-at-a-time.h  
 34 : FFFFFF800839C4DC: B 0xFFFFFF800839C500  
 arch/arm64/include/asm/uaccess.h  
 205 : FFFFFF800839C500:   
 FFFFFF800839C504: ADD X6, X1, X3  
 lib/strncpy\_from\_user.c  
 43 : FFFFFF800839C50C: LDR X8, [X6] ; 进程停止在这里  
 当时的寄存器值:  
 X0: FFFFFFC0AF30A01C, X1: 00000000F0A7FFCC, X2: 0000000000000FE4, X3: 0000000000000030  
 X4: 0000000000000FB4, X5: 0000000000000000, X6: 00000000F0A7FFFC, X7: 0000000000000000  
 X8: 6165722E6769666E, X9: FEFEFEFEFEFEFEFF, X10: 0000000000000000, X11: 00000000FFCEF35C  
 X12: 00000000FFCEECC8, X13: 00000000FFCEEC68, X14: 00000000F3E7C86D, X15: 0000000000000000  
 X16: FFFFFF80082261C0, X17: 0000000000000000, X18: 0000000000000000, X19: FFFFFFC0AF30A000  
 X20: 00000000F0A7FFCC, X21: FFFFFF800946C000, X22: 0000000000000000, X23: 0000000000000000  
 X24: 0000000000000011, X25: 0000000000000186, X26: 0000000000000142, X27: FFFFFF8008DF3000  
 X28: FFFFFFC015A24000, X29: FFFFFFC015A27DF0, X30: FFFFFF80081DF44C, SP: FFFFFFC015A27DF0  
 PC: FFFFFF800839C50C

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 0xFFFFFFC0293F6000，其中0xFFFFFFC0293F6000为task对应的task\_struct地址

FFFFFFC0557B3270|00000000F09FF000--00000000F0A00000|00003000|rw-p |03:16|  2705|/vendor/lib/libfbc.so  
FFFFFFC0097C2AF8|00000000F0A00000--00000000F0A80000|00003000|rw-p |00:00|    0|[anon:libc\_malloc]  
FFFFFFC055617EA0|00000000F0A80000--00000000F0A81000|00044000|r--s |00:10|  14460|/dri/renderD128  
FFFFFFC055617000|00000000F0A81000--00000000F0A82000|0001E000|rw-s |00:10|  14460|/dri/renderD128  
FFFFFFC081FEA888|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)0xFFFFFFC015A24000).task = 0xFFFFFFC0293F6000-> (  
 state = 0x0,  
 stack = 0xFFFFFFC015A24000 = end+0x400BD4C000,  
 ......  
 mm = 0xFFFFFFC03DD3B3C0 = end+0x40340633C0 -> (  
 mmap = 0xFFFFFFC03F2C5888 = end+0x40355ED888,  
 mm\_rb = (rb\_node = 0xFFFFFFC02CB29158 = end+0x4022E51158),  
 vmacache\_seqnum = 0x023A,  
 get\_unmapped\_area = 0xFFFFFF80081B2470 = arch\_get\_unmapped\_area\_topdown,  
 mmap\_base = 0xF5F51000,  
 mmap\_legacy\_base = 0x0,  
 task\_size = 0x0000000100000000,  
 highest\_vm\_end = 0xFFFF1000,  
 pgd\_=\_0xFFFFFFC05EDB8000,

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  -> 0xFFFFFFC088699000  
pmd[0x185] = 0xE716B003      -> 0xFFFFFFC0A716B000  
pte[0x7F] = 00E80000A61A1F53 -> 0xFFFFFFC0661A1000  
00E80000A61A1F53对应的属性：  
SH=inner shareable  
AP=RW  
AttrIndx=4  
  
  
va = F0A80000 -> pa = 0x8964F000  
pgd index = 3  
pmd index = 0x185  
pte index = 0x80  
  
pgd[3] = C8699003  -> 0xFFFFFFC088699000  
pmd[0x185] = 0xE716B003      -> 0xFFFFFFC0A716B000  
pte[0x80] = 016000008964FFC3 -> 0xFFFFFFC04964F000  
016000008964FFC3对应的属性：  
SH=inner shareable  
AP=RO  
AttrIndx=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：**

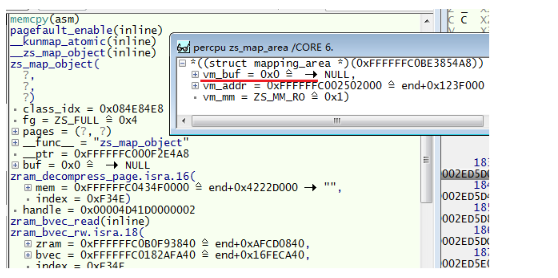
手机重启(kernel panic)

**分析过程：**

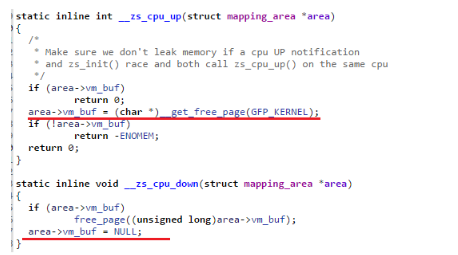
用GAT解开db，并结合对应的vmlinux（该文件必须和db一致，具体请看FAQ06985），利用工具（E-Consulter.jar）分析（也可以参考FAQ13941），解析出来的调用栈如下：

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=0xFFFFFFC0A337C000, handle=84945057873922, mm=ZS\_MM\_RO) + 404 <mm/zsmalloc.c:1143>  
 vmlinux zram\_decompress\_page(mem=0xFFFFFFC0434F0000, index=62286, zram=0xFFFFFFC0434F0000, 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=0xFFFFFFC0B0F93840, bvec=0xFFFFFFC0182AFA40, 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=0xFFFFFFC0573C99C0) + 556 <drivers/block/zram/zram\_drv.c:1341>  
 vmlinux generic\_make\_request(bio=0xFFFFFFC0573C99C0) + 136 <block/blk-core.c:1924>  
 vmlinux submit\_bio(rw=0, bio=0xFFFFFFC0573C99C0) + 128 <block/blk-core.c:2005>  
 vmlinux swap\_readpage(page=0xFFFFFFBE01CB9480) + 352 <mm/page\_io.c:390>  
 vmlinux read\_swap\_cache\_async(entry=0x000000000000F34E, gfp\_mask=131290, vma=0xFFFFFFC0AD9E0840, addr=3304787968) + 220 <mm/swap\_state.c:376>  
 vmlinux swapin\_readahead(entry=0x000000000000F34E, gfp\_mask=131290, vma=0xFFFFFFC0AD9E0840, 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=0xFFFFFFC012A2BC00, vma=0xFFFFFFC0AD9E0840, 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=0xFFFFFFC0182AFED0) + 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:



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\_mtkcooler: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-protocal" 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-protocal" 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-protocal]audit: audit\_lost=352917 audit\_rate\_limit=20 audit\_backlog\_limit=64  
  
[179276.712426]  (4)[30905:pool-2-protocal]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-protocal" 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-protocal" 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-protocal" 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-protocal" 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-protocal" 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-protocal" 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-protocal" 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-protocal" 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-protocal" name="22" dev="proc" ino=5261515 scontext=u:r:untrusted\_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=0  
  
[179276.713417]  (2)[289:logd.auditd]type=1400 audit(1467250068.100:422218): avc: denied { search } for pid=30905 comm="pool-2-protocal" name="23" dev="proc" ino=5261516 scontext=u:r:untrusted\_app:s0:c512,c768 tcontext=u:r:kernel:s0 tclass=dir permissive=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看起來,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 == mlstrustedsubject or t2 == mlstrustedobject);

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-protocal" 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 mlstrustedsubject;  也可以将 target定义为mlstrustedobject.

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