**深圳市普联技术有限公司**

**TP-LINK TECHNOLOGIES CO., LTD.**

**C20(SP)4.0** **UES问题分析总结**

|  |  |
| --- | --- |
| **版 本:** | V1.1 |
| **完成日期:** | 17Oct2017 |
| **作 者:** | 张文开 |
| **审 核:** |  |
| **批 准:** |  |
| **文件状态:** | ■草稿 □正式发布 □正在修改 |

**版 本 历 史**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本/状态** | **责任人** | **起止日期** | **备注** |
| V1.0/草稿 | 张文开 | 10Oct2017 | 创建文档。 |
| V1.1/ 草稿 | 张文开 | 17Oct2017 | 修改文档格式。 |

目录

[案例说明 3](#_Toc496016896)

[分析解决 4](#_Toc496016897)

[总结 6](#_Toc496016898)

## 案例说明

|  |  |
| --- | --- |
| 机型 | C20(SP) 4.0 |
| 芯片方案 | MT7628A + MT7610E |
| BBA平台版本 | 1.5 |
| kernel版本 | Linux 3.10.14.x |
| 体系结构 | mips |
| SDK版本 | mtk\_ApSoC\_5020 |

C20(SP) 4.0外网问题性测试fail，具体表现：

1. 第一次，kernel panic；
2. 第二次，kernel panic，反汇编定位到free skb时，skb\_shared\_info中的frags字段不为空，导致free错误；
3. 第三次，客户端全部掉线，串口曾打印内存不足，kill掉一个无线相关的模块；

相关修改：

（1）mtk提供了2个patch，处理内存不足的情况；

（2）修复2.4G 在处理分片数据包时可能存在的内存越界；

（3）修复eth中skb recycle机制，不匹配内核3.10.14的问题，该问题也可能引起kernel panic；

（4）修复wireless skb recycle机制中，不匹配内核3.10.14的问题，同样该问题会在free skb时kernel panic；

（5）改小5G DMA RX BUFFER大小（改小64字节），保证DMA大小在一页以内，修复内存不足的问题；

（6）修复skbuff.c中，一处在free skb后赋值的情况。

UES测试中，每次fail的表现不一致，在分析，解决过程中，我们修改的地方也不止一处， UES测试时测试条件的复杂性，也不能使用变量控制的方法来唯一确认原因。鉴于上述原因，在下一节描述分析解决过程时，并不会以“问题、原因、解决方法”这样的格式来阐述，而是按实际解决问题的时间线来讲这个故事。当然，这样描述，会给理解导致该问题的原因带来困难，但通过讲述问题解决的过程也给读者一些解决UES相关问题的思路，选择讲故事的形式也是无奈之举。

阅读下一节前，最好对linux的内存管理（buddy system & slab system）、软中断机制有一定了解。

## 分析解决

分析过程，主要从如下几个方面入手：

1. 反汇编分析。这一步比较简单，先从panic信息定位出错的模块（二进制文件，结合出错指令地址以及kernel的地址分布），然后反汇编二进制文件，根据指令地址找到汇编代码段，及对应的C代码。然后根据汇编代码（mips的汇编指令、寄存器信息，简单地百度就可以获取。mips的汇编指令格式和书上x86有些区别，主要是使用了更多的寄存器），找到出错的C代码语句。

一般，简单的内核bug错误，通过这一步就可以定位。但对于一些隐藏更深的错误，这些分析并不能直接定位原因，比较遗憾的是，我们遇到的问题，通常是这种情况。另一方面，反汇编分析虽然不能帮助我们直接定位原因，但根据出错的函数，能给我们一些必要的提示。

比如，在此问题中，反汇编显示出错指令在free skb时frags数组不为空，导致free错误。

*Sk\_buff中的skb\_shared\_info用于记录分片信息，其包含两种机制：*

* *IP分片，分片数据放在frag\_list链表中，每一个成员均是一个sk\_buff结构体，在IP层处理结束时分片；*
* *DMA分配，分片数据保存在frags数组中，结构为skb\_frag\_t，数据以page形式组织，这种机制通常需要硬件支持SCATTER的DMA；*

*这两种机制，数据保存方式不同，所以free时处理也不同，前者应该调用kfree\_skb释放，而后者应该调用free page相关函数释放。*

通过review代码，确认不支持SCATTER DMA。所以猜测原因在于内存污染，某处写操作越界导致frags不为NULL，只能从数据的data path一步步分析。

1. 转发数据的data path，从驱动DMA接收数据包，一直到DMA发送数据包。其中比较重要的点在于数据sk\_buff的成员，以及其申请、释放，硬中断、软中断的处理。

在解决此UES问题中，我们发现的bug，基本上都是沿着data path这条线发现的。而实际上，我估计所有的UES问题均可以通过分析转发data path来解决的，因为作为一个router，尤其是在外网UES测试时，其核心任务就是转发网络数据。在软件层面，无非也就是从一个驱动接收到另一个驱动接收的过程。

具体地，在查看data path相关问题时，主要关注的一些代码点、或者说知识点：

1. DMA Buffer的申请、释放。

*在mtk方案中，eth和wifi均存在循环利用sk\_buff的机制。具体实现上，eth和wifi不相同，表现在：*

* *eth申请sk\_buff直接调用mtk实现的API，所有空闲sk\_buff以链表的形式保存在一个内核全局链表中，而在free\_skb时将sk\_buff挂在该链表上；而wifi驱动则是将空闲sk\_buff挂在AP\_ADAPTER结构体的一个成员链表中，申请释放时，使用显示的if语句判断，选择使用内核API dev\_alloc\_skb还是自定义的从链表中取出sk\_buff使用；*
* *eth和wifi的sk\_buff申请均在需要从DMA取出数据包处理时，而释机制却不同，eth通过在sk\_buff中增加一个释放的函数指针，在free\_skb时调用该函数，而wifi的释放，却是在驱动将数据包发送给硬件后，显式回收。所以，eth的循环机制适用于整个router中，而wifi的循环机制却仅限于wifi驱动内部。这里需要小心一种情况，那就是当eth和wifi使用的skb大小一样时，有可能会存在eth申请的sk\_buff被wifi吃掉。*

*循环利用sk\_buff的机制，一方面加快了sk\_buff的申请过程，另一方面，也使router运行更加稳定，减少内存碎片的产生。*

1. 内核版本变化

C20(EU) 4.0基于linux kernel 2.6.36，而C20(SP) 4.0则基于linux kernel 3.10.14，与此问题相关的变化较大的在于申请sk\_buff时，3.10.14引入了一个机制：

使用dev\_alloc\_skb时，如果申请的sk\_buff长度小于PAGE\_SIZE（4k），直接从页分配系统分配skb的数据，而不通过slab来分配。这样的sk\_buff在初始化时，需要设置sk\_buff ->head\_frag标识，在释放时，通过判断head\_frag来区分调用free\_page还是通过slab释放API。前面我们提到的循环机制中，存在一些mtk自定义的API，其中包含了sk\_buff的初始化，其中就没有考虑此种情况。

* *Dev\_alloc\_skb通常用于在驱动中申请skb，其通过设置atomic原子标识来保证不阻塞。Mtk的eth驱动并没有调用该API，仅wifi驱动中调用。*
* *对应于dev\_alloc\_skb，还存在一个dev\_release\_skb函数，该函数用于在驱动中释放sk\_buff。它仅将sk\_buff挂在每CPU变量soft\_net\_data中的一个成员链表中，真正的释放发生在NET\_TX\_SOFTIRQ的处理函数中。*

另外，slab系统有一个不好的地方，如果申请的内存长度必须是2的幂。这样，如果需要的sk\_buff长度刚好为2049，那么需要申请的内存为4096，浪费一半左右的内存。而通过page来申请则不会出现这种情况。

当然，就个人理解而言，slab系统也有优于page系统的地方，slab会有染色机制，这样在CPU cache的运用上，slab应该会优于page系统（特别是申请的sk\_buff大小刚好是cache line大小的整数倍时，当然具体需要看cache的情况）。

1. 硬中断、软中断（NAPI、tasklet以及转发数据内核协议栈的处理），以及其相互的优先级关系

在阅读代码时，函数的调用关系通常能给我们勾画出一条清晰明白的时间线，告诉我们先发生了什么，然后发生什么。但当遇到中断时，一切都变了，硬中端、软中断、进程，以及穿插其中的内核抢占，异常处理（CPU内部中断，这里主要是指缺页异常），各个内核路径相互交叉，时间线不再明显。

如果对内核、驱动的代码，在什么时候执行，处于什么上下文（这对于检查代码的同步是否正确很重要），以及其相互依赖关系不明确的话，可以尝试先弄清楚这些看上去很深奥的概念的准确含义以及实现原理。下面，列一些我认为的重点（水平所限，可能存在错误）：

* *Mtk的驱动硬中断，习惯将很多事件放在一个中断中，通过查看寄存器的值来判断真正发生的事件是什么，中断处理函数中仅raise软中断，不做实际的数据包处理;*
* *Mtk eth驱动使用NAPI，NAPI通过NET\_RX\_SOFTIRQ的处理函数中调用。NAPI存在一个budget用于限制每次NAPI POLL的数据包的最大量，而在NET\_RX\_SOFTIRQ中，也存在一个weight，用于限制每次NET\_RX\_SOFTIRQ处理数据包的最大量，注意这些变量的值。*
* *Mtk wifi驱动使用tasklet或者workqueue，配合软中断NET\_RX\_SOFTIRQ来处理中断下半部。这两者的区别是，tasklet通过HI\_SOFTIRQ（这里不同于一般的kernel情况，一般的kernel是使用TASKLET\_SOFTIRQ来实现的，区别在于其优先级。很微妙的是，HI\_SOFTIRQ优先级高于NET\_RX\_SOFTIRQ）实现，在软中断上下文中，而后者则仅仅是一个内核进程。*

*这两者最大的区别在于处理的优先级，前者通过软中断实现，虽然软中断也存在调用内核进程softirq来处理的情况，但总的说来，处理的优先级、及时性应该高于后者。本来这两者之前的优先级关系并不特别重要，但是考虑到它们需要和另一个软中断NET\_RX\_SOFTIRQ配合来完成数据包的后续处理，这样就很重要了。Tasklet、workqueue的任务是将数据包从DMA上取出，处理wifi数据包头部及wifi管理帧，然后将数据包挂在每CPU变量softirq\_data中，等待NET\_RX\_SOFTIRQ处理函数调度处理（协议栈，以及交给eth驱动）。*

*前面提到tasklet的任务是将数据包放在每CPU变量的链表中，等待NET\_RX\_SOFTIRQ调度处理。所以链表长度、以及NET\_RX\_SOFTIRQ每次调度处理的数据包数量也是很重要的参数，需要关注。*

## 总结

在工作中，UES问题总是困扰我们，也是我们最不愿意遇见然而又不得不面对的bug，通常这样的bug，我们会丢给供应商去解决，分析解决UES问题能有效提高我们对内核核心任务的理解。

本文简单总结了我们在C20(SP) 4.0 UES问题的分析解决过程，主要是一些心得和对相关知识点的理解，希望对读者有一些启发。