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

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。

（2） 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：

a. 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。

b. 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）。

c. 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。