

## 目录

目录 .....	1
第一章 简介.....	1
1.1 Exception trigger 是什么 .....	1
1.2 为什么要实现 exp_trigger .....	1
1.3 本文如何展开.....	1
第二章 使用方法.....	1
2.1 环境搭建 .....	1
2.2 运行验证 .....	2
第三章 背景知识.....	3
3.1 页表 .....	3
3.1.1 查表过程.....	3
3.1.2 各级描述符的格式.....	4
第四章 ARM64 异常处理.....	5
4.1 ARM64 异常分类.....	5
4.2 Linux 中对 ARM64 异常的处理 .....	6
第五章 异常触发及原理分析 .....	9
5.1 Synchronous exception .....	9
5.1.1 MMU fault.....	9
5.1.1.1 Address size fault.....	10
A. 什么是 address size fault.....	10
B. Linux 中对 Address size fault 的处理 .....	10
C. Exception trigger 如何制造 address size fault.....	11
D. 运行结果 .....	13
5.1.1.2 Translation fault.....	15
A. 什么是 translation fault.....	15
B. Linux 中对 translation fault 的处理 .....	15
C. Exception trigger 如何制造 translation fault.....	20
D. 运行结果 .....	22
5.1.1.3 Access flag fault.....	25
A. 什么是 access flag fault.....	25
B. Linux 中对 access flag fault 的处理.....	25
C. Exception trigger 如何制造 access flag fault.....	26
D. 运行结果 .....	27
5.1.1.4 Alignment fault.....	28
A. 什么是 alignment fault.....	28
B. Linux 中对 alignment fault 的处理.....	29
C. Exception trigger 如何制造 alignment fault.....	30
D. 运行结果 .....	31
5.1.1.5 Permission fault.....	32
A. 什么是 permission fault .....	32
B. Linux 中对 permission fault 的处理 .....	32

## Exception trigger 模块说明文档

C. Exception trigger 如何制造 permission fault .....	33
D. 运行结果 .....	35
5.1.1.6 synchronous external abort on translation table walk.....	36
A. 什么是 synchronous external abort on translation table walk .....	36
B. Linux 中对 synchronous external abort on translation table walk 的处理 .....	36
C. Exp_trigger 如何制造 synchronous external abort on translation table walk ...	36
D. 运行结果 .....	37
5.1.2 Synchronous external abort .....	37
5.1.2.1 什么是 synchronous external abort .....	37
5.1.2.2 Linux 下如何处理 synchronous external abort .....	37
5.1.2.3 Exp_trigger 如何触发 synchronous external abort .....	37
5.1.2.4 运行结果 .....	38
5.2 SError.....	38
5.2.1 什么是 SError.....	38
5.2.2 Linux 下如何处理 SError .....	38
5.2.3 Exp_trigger 如何触发 SError .....	38
5.2.4 运行结果.....	39
参考文档.....	39
附录 .....	39
1. Fixup exception 的例子 .....	39
2. ICE 查看 Sp alignment fault 的现场 .....	41

## 第一章 简介

### 1.1 Exception trigger 是什么

Exception trigger 是一个辅助学习 ARMv8 AARCH64 模式下，各类异常触发原因及 Linux 下对应处理流程的模块，由代码和说明文档(即本文档)两部分构成。其异常触发有简单及能稳定复现的特点，对于每一类异常用例的构建及 Linux 下相应的处理流程则由本文档给出详细说明。为简单起见，下文将 Exception trigger 缩写为 exp\_trigger。

在 exp\_trigger 的使用过程中，遇到任何问题均可直接联系我：[马佳敏, mjmthy@163.com](mailto:mjmthy@163.com)。

### 1.2 为什么要实现 exp\_trigger

稳定性问题分析是一项非常有挑战性的任务，只有在大量的项目问题分析积累以及对所分析问题的归类整理后，才能渐渐独立承担分析这类问题的重任。对于稳定性类问题分析，需要丰富的 Linux 内核背景知识以及多样化的分析技巧，但网上可供参考的资料分散且不多。另外，项目上遇到的稳定性问题在一段时间内往往是同一类问题，经验积累的周期比较长，而且也不太适合作为学习的例子。如果我们有一个模块，它人为构建了每一类异常，借助于它可以了解到每一类异常触发的最终原因，以及这类异常在 Linux 下是如何被处理的，那么无疑会是对稳定性分析的一个有力辅助。Exp\_trigger 模块就是为了这一目的而实现的。

### 1.3 本文如何展开

首先，在第二章中会说明如何用搭建 exp\_trigger 运行的软硬件环境。接着为了理解 exp\_trigger 的原理和实现，分别在第三章和第四章中描述了必要的背景知识以及 Linux 下对 ARM64 的异常处理流程。在第五章中，对于每一类异常，会分别从定义，Linux 下的处理流程，异常用例的构造以及异常现场这四个方展开描述。在参考文档中会列出引用到的一些文档。最后附录将作为正文的补充内容，供有兴趣的读者扩展阅读。

## 第二章 使用方法

### 2.1 环境搭建

硬件上我选取了在 Amlogic 常见的 Ampere 平台，这个平台也是目前市面上可以购买到的小米盒子的硬件解决方案。Ampere 上集成了 4 个 ARMv8.0 架构的 A53 的核，支持运行 64 位的 Linux 操作系统。软件上我选取了目前 Amlogic 在维护的 amlogic-4.9-dev 分支内核，该

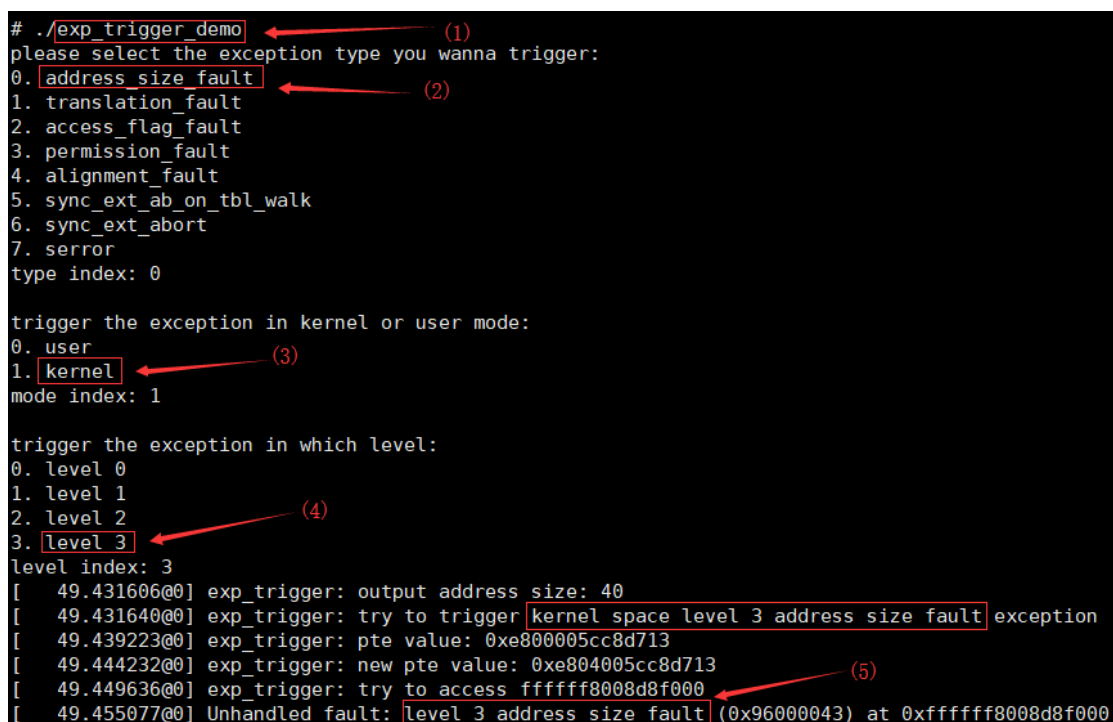
## Exception trigger 模块说明文档

内核基于 Linux mainline 4.9.113。Exp\_trigger 的代码分为驱动和用户空间应用两部分，前者与 Linux 内核一起编译，后者提供了基于 Makefile 的单独编译方式，编译出的应用名字为 exp\_trigger\_demo。具体交叉编译的配置不在本文描述范围内

虽然我在这里选取的软硬环境均是基于 Amlogic 平台的，但我相信代码基本不需改动就可运行在所有 Linux 4.9.113 ARMv8 环境下。

## 2.2 运行验证

Exp\_trigger 中所有支持触发的异常，均通过 ioctl 的方式向用户空间提供触发的方法。exp\_trigger 中的用户空间程序 exp\_trigger\_demo 对 ioctl 的操作作了进一步封装，可以由用户直接指定要触发的异常。如果 exp\_trigger 支持这类异常的触发，那么会直接触发该异常；如果暂不支持该类异常的触发，也会有相应的提示。exp\_trigger\_demo 是一个交互式的应用，只需按照其提示一步步操作即可。图 1 中所示的就是使用 exp\_trigger\_demo 的两个例子。



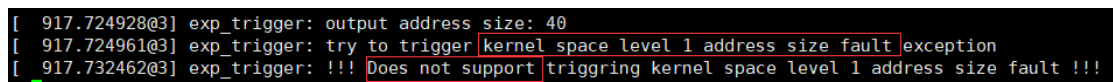
```
# ./exp_trigger_demo (1)
please select the exception type you wanna trigger:
0. address size fault (2)
1. translation_fault
2. access_flag_fault
3. permission_fault
4. alignment_fault
5. sync_ext_ab_on_tbl_walk
6. sync_ext_abort
7. serror
type index: 0

trigger the exception in kernel or user mode:
0. user
1. kernel (3)
mode index: 1

trigger the exception in which level:
0. level 0
1. level 1
2. level 2
3. level 3 (4)
level index: 3
[ 49.431606@0] exp_trigger: output address size: 40
[ 49.431640@0] exp_trigger: try to trigger kernel space level 3 address size fault exception
[ 49.439223@0] exp_trigger: pte value: 0xe800005cc8d713
[ 49.444232@0] exp_trigger: new pte value: 0xe804005cc8d713
[ 49.449636@0] exp_trigger: try to access fffffff8008d8f000 (5)
[ 49.455077@0] Unhandled fault: level 3 address size fault (0x96000043) at 0xfffff8008d8f000
```

图 1-a. exp\_trigger 使用样例，触发支持的异常

- (1) 通过应用 exp\_trigger\_demo 来触发异常
- (2) 选择想要触发的异常类型，这里选择的是 address size fault 类型的异常
- (3) 选择在用户空间还是内核空间触发该异常，这里选择的是在内核空间触发
- (4) 选择触发哪个 level 的异常，这里选择的是 level 3 的异常
- (5) 现在尝试触发的是 kernel space level 3 address size fault，在这里已经可以看到对应异常被触发的现场



```
[ 917.724928@3] exp_trigger: output address size: 40
[ 917.724961@3] exp_trigger: try to trigger kernel space level 1 address size fault exception
[ 917.732462@3] exp_trigger: !!! Does not support triggering kernel space level 1 address size fault !!!
```

图 1-b. exp\_trigger 使用样例，触发不支持的异常

## Exception trigger 模块说明文档

这里略去了通过 `exp_trigger_demo` 选择触发异常的步骤，下同。可以看到所选择触发的 `kernel space level 1 address size fault` 是当前 `exp_trigger` 模块不支持触发的异常类型，`exp_trigger` 会给出相应的提示。

## 第三章 背景知识

### 3.1 页表

要理解后续章节中 MMU Faults 这一类同步异常相关的内容，必须要了解 ARM64 下页表的相关知识。在 ARM Spec<sup>A</sup> 的 D4.2 The VMSAv8-64 address translation system 和 D4.3 VMSAv8-64 translation table format descriptors 能找到详尽的描述。在这一节中，我们会就页表管理的查表过程和各级描述符的格式展开叙述。

#### 3.1.1 查表过程

所谓的查表就是通过输入的虚拟地址，遍历页表，获取对应物理地址的过程。ARM64 下最多支持 4 级页表，而当前 Linux 下默认使用的是 3 级页表(通过宏 `CONFIG_PGTABLE_LEVELS` 配置)。其中用户空间的页表由 `ttbr0` 指向，内核空间的页表由 `ttbr1` 指向。每一级页表中的表项可能由 3 种不同的描述符构成，分别是用于指向下一级页表起始地址的 `table descriptor`；用于指向一块连续内存区域及其属性的 `block descriptor`；用于指向一个物理页及其属性的 `page descriptor`。在使用 4K 页(`CONFIG_ARM64_PAGE_SHIFT` 配置为 12)的情况下，level 1 中的 `D_Block` 指向的是物理地址 1G 连续的内存区间，level 2 中 `D_Block` 指向的是物理地址 2M 连续的内存区间，而 level 3 中的 `D_Page` 指向的是物理地址 4K 连续的页。具体的查表过程如下图所示：

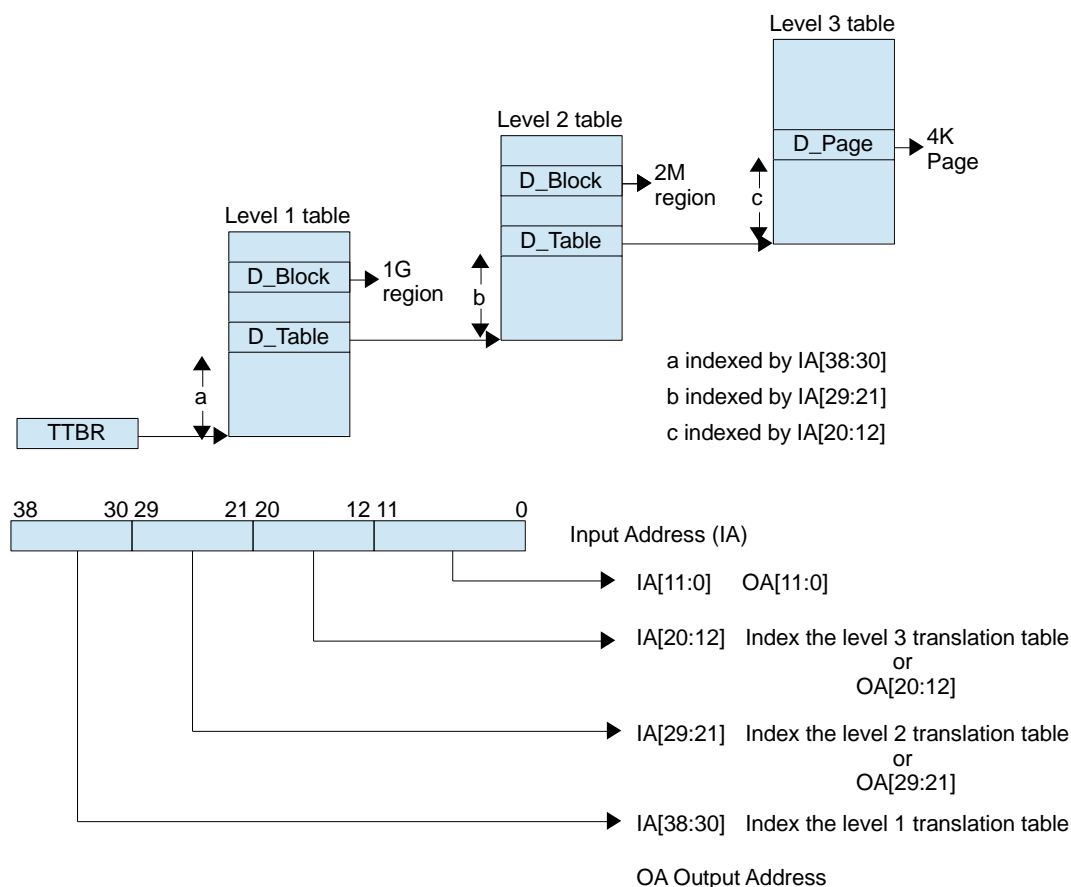


图 2. Linux ARM64 下 3 级页表查表过程

在 3 级页表下，使用的虚拟地址位宽是 39 位。在查表时会先根据 `ttbr` 寄存器获取第一级页表的起始地址，然后根据虚拟地址的第 30~38 这 9 个 bit 获取第 1 级页表表项的下标，根据这两个信息即可得到第 1 级页表中的描述符。如果描述符是一个 **Block Descriptor** (具体的格式在下一小节中叙述，下同)，则通过 **Block Descriptor** 可以知道该 1G 内存区域的起始地址，并通过 IA[29:0] 获取该 1G 区域中的偏移；如果描述符是一个 **Table Descriptor**，则通过该 **Table Descriptor** 可以知道第 2 级页表的起始地址。接着通过虚拟地址的第 21~29 这 9 个 bit 获取第 2 级页表表项的下标，从而得到第 2 级页表中的描述符，如果描述符是一个 **Block Descriptor**，则通过 **Block Descriptor** 可以知道该 2M 内存区域的起始地址，并通过 IA[20:0] 获取该 2M 区域中的偏移；如果描述符是一个 **Table Descriptor**，则通过该 **Table Descriptor** 可以知道第 3 级页表的起始地址。最后，根据虚拟地址的第 12~20 这 9 个 bit 获取第 3 级页表表项的下标，从而得到第 3 级页表中的 **Page Descriptor**，通过 **Page Descriptor** 可以知道物理页的起始地址，并通过 IA[11:0] 获取该物理页面中的具体偏移。

### 3.1.2 各级描述符的格式

在不同的物理页 (支持 4K, 8K 和 16K 3 种) 大小情况下，描述符格式会略有不同，Linux 默认情况下使用的是 4K 大小的物理页，下面的描述符都是在 4K 物理页大小情况下的。

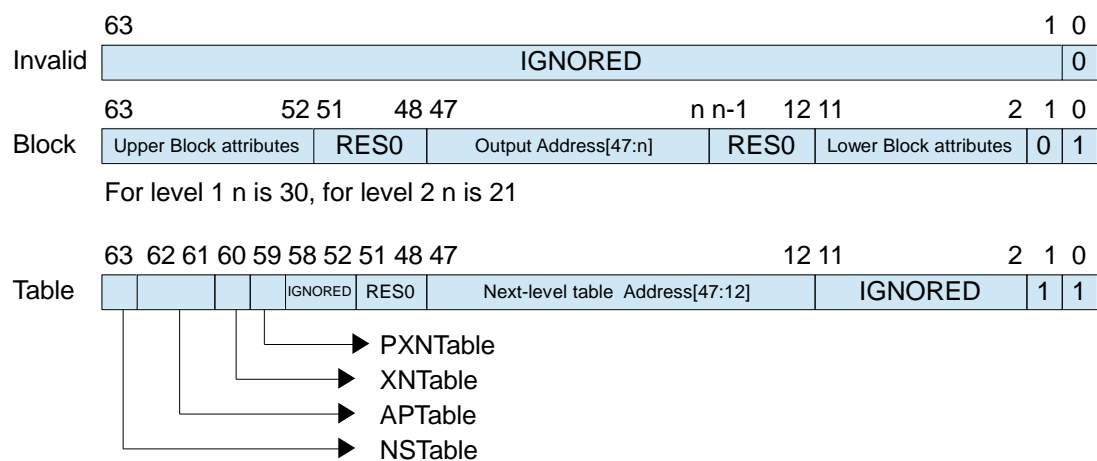


图 3-a. Level 1~2 描述符格式

上图显示的是 level 1~2 的页表描述符，只有在描述符的 bit[0]为 1 时，才是有效的描述符。有效描述符由 Block Descriptor 和 Table Descriptor 构成，通过描述符中的 bit[1]区分，0 为前者，1 为后者。这里的 Output Address 指向 1G 或者 2M 物理地址连续区域的起始地址，Next-level table address 则指向下一级页表的起始地址。另外除了与地址相关的域外，还有很多与属性相关的域，里面牵涉到权限、缓存等等的配置，会在后面章节具体使用到时再解释。

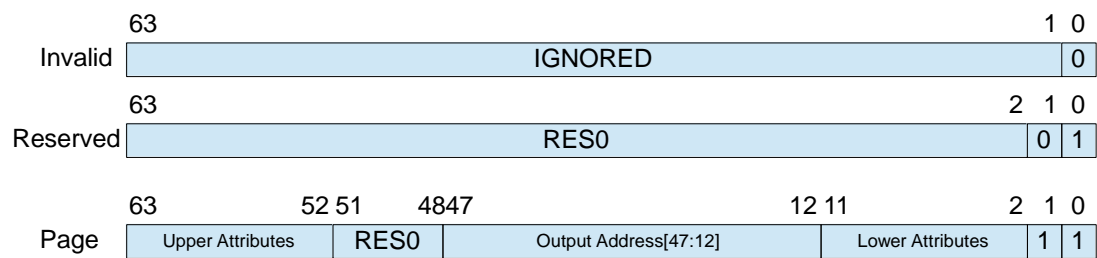


图 3-b. Level 3 描述符格式

上图显示的是 Level 3 的页描述符，只有在 bit[1:0]为 11 时才是有效的 Page Descriptor。同样的 Page Descriptor 中的域由地址和属性域构成。这里的 Output Address 指向 4K 页的起始地址。属性域中具体的项在后续章节使用到时再解释。

## 第四章 ARM64 异常处理

### 4.1 ARM64 异常分类

根据官方文档<sup>AB</sup>的描述，ARM64 AARCH64 模式下支持的异常类型大致如图 4 所示，可以分为同步类型和异步类型的异常。如果当前执行的指令就是触发异常的指令，那么所触发的异常就是同步异常，反之就是异步异常。同步异常种类众多，有 data access 和 instruction fetch 引起的 MMU fault 异常；执行系统调用、调用 secure monitor 和调用 hypervisor 引起的 SVC/SMC/HVC 异常；调试内核触发的 breakpoint 异常、software step 异常和 watchpoint 异常

## Exception trigger 模块说明文档

等等。异步异常由 IRQ, FIQ 和 SError 构成, 最常见的 SError 是异步数据访问异常(asynchronous data abort), 这类异常通常是因为 cache line 中的数据向主存回写引起的。

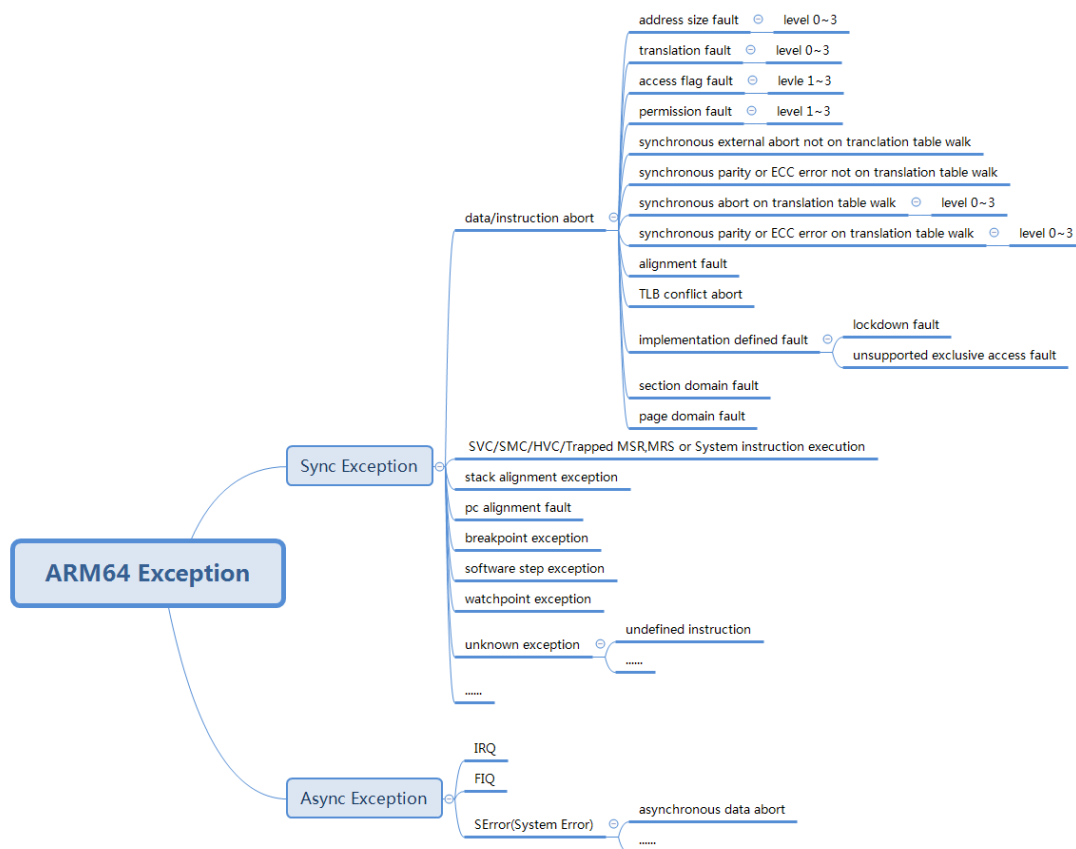


图 4. ARM64 异常类型

## 4.2 Linux 中对 ARM64 异常的处理

Linux ARM64 下发生异常时, 会进入 ARM64 异常向量表中对应的项执行。Linux ARM64 的异常向量表由四组构成, 分别对应不同环境下的异常触发: a. 异常在 EL1 中产生, 交由 EL1 处理, 使用的堆栈是 SP\_ELO; b. 异常在 EL1 中产生, 交由 EL1 处理, 使用的堆栈是 SP\_EL1; c. 异常在 EL0 中产生, 交由 EL1 处理; d. 基本与 c 相同, 区别在于异常发生在 32 位环境下。在每一组中, 又根据不同类型的异常有单独的异常向量: 同步异常, irq, fiq 和 SError。对应具体执行流程如下所示:

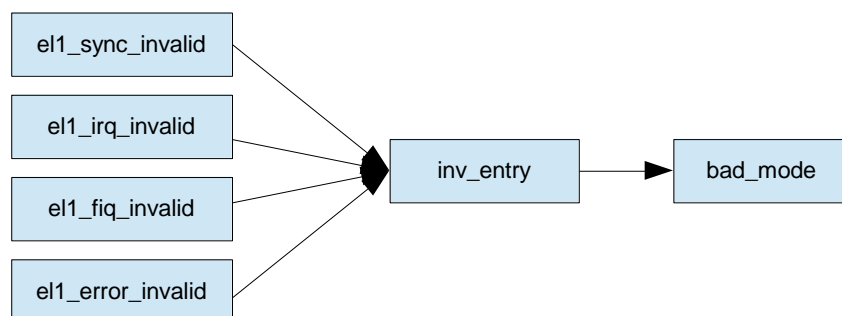


图 5. EL1 中处理 EL1 产生的异常, 且使用 SP\_ELO



## Exception trigger 模块说明文档

上图是异常发生在 EL1，且交由 EL1 处理，并使用 SP\_EL0 情况下，进入异常向量表后的代码执行流程。可以看到对应四类异常的处理函数均带有\_invalid 后缀，表示 Linux ARM64 下不支持对这类异常的处理。这些处理函数都会直接调用 inv\_entry，后者直接调用 bad\_mod，该函数在打印必要的信息后，会触发 panic。对于 bad\_mod 的具体执行流程，会在后续章节具体的异常中描述。

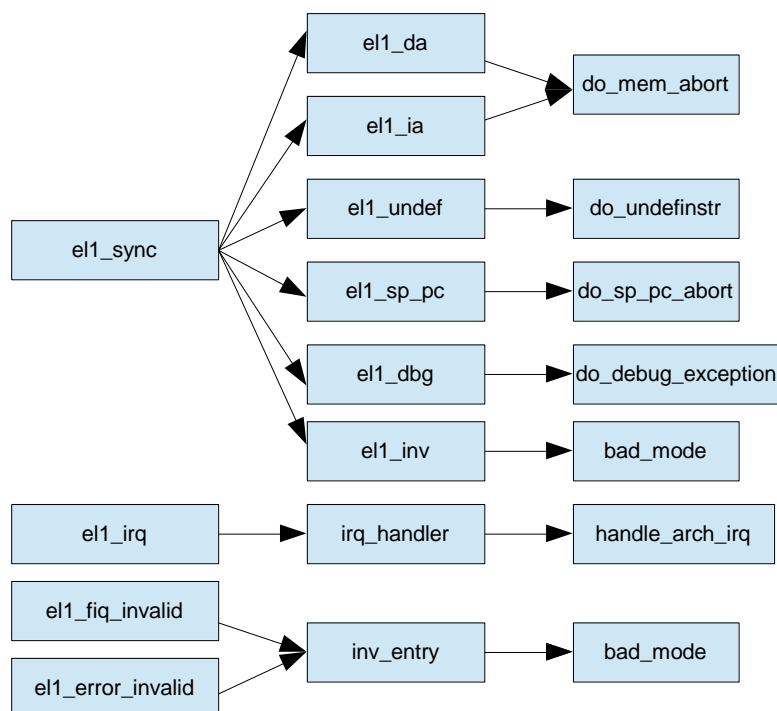


图 6. EL1 中处理 EL1 产生的异常，且使用 SP\_EL1

上图是异常发生在 EL1，且交由 EL1 处理，并使用 SP\_EL1 情况下，进入异常向量表后的代码执行流程。从图中可以看到对于同步类型的异常，会根据具体的同步类型异常有对应的处理函数，当发生 synchronous data/instruction abort 时，会调用 el1\_da 和 el1\_ia，两者最终都会调用 do\_mem\_abort 处理(这一层级的函数的具体执行流程会在后续章节具体的异常中描述，下同)；如果是由于 undefined instruction 触发的同步异常，则会调用 el1\_undef，该函数最终会调用 do\_undefinstr 处理；如果是 sp/pc alignment fault，则会调用 el1\_sp\_pc，该函数最终会调用 do\_sp\_pc\_abort 处理；对于调试类的异常，则会通过 el1\_dbg 调用 do\_debug\_exception 处理；对于其它同步类型异常，均属于不可处理异常，最终会调用 bad\_mode。

如果是中断触发的异常，则会进入 irq\_handler，该函数会调用 handle\_arch\_irq 进入对于的中断处理函数执行。对于 fiq 和 SError 触发的异常，同样是不被 Linux ARM64 支持的。

## Exception trigger 模块说明文档

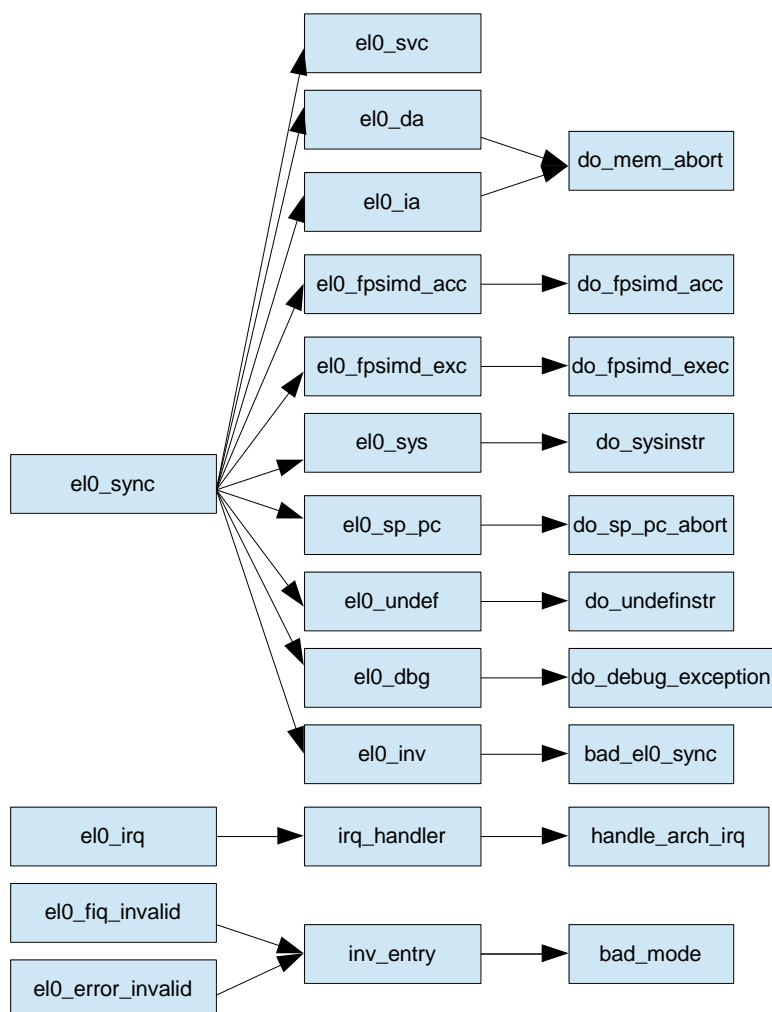


图 7. EL1 中处理 EL0 的异常，且 EL0 运行在 64bit

上图是异常发生在 EL0(运行在 64bit)且交由 EL1 处理的情况下，进入异常向量表后的代码执行流程。同样的对于同步类型的异常，会根据具体的同步类型异常有对应的处理函数。区别于前一小节的例子，这里对于系统调用，SIMD 操作引起的异常和 configurable trap(比如对于在 EL0 下默认没有权限访问的 virtual counter，可以通过添加对应 trap handler 的方式来访问)，有分别的处理函数 el0\_svc，el0\_fpsimd\_acc/exc 和 el0\_sys。

而 irq，fiq 和 SError 类型的异常，则和前一小节完全一致，这里不加详述。

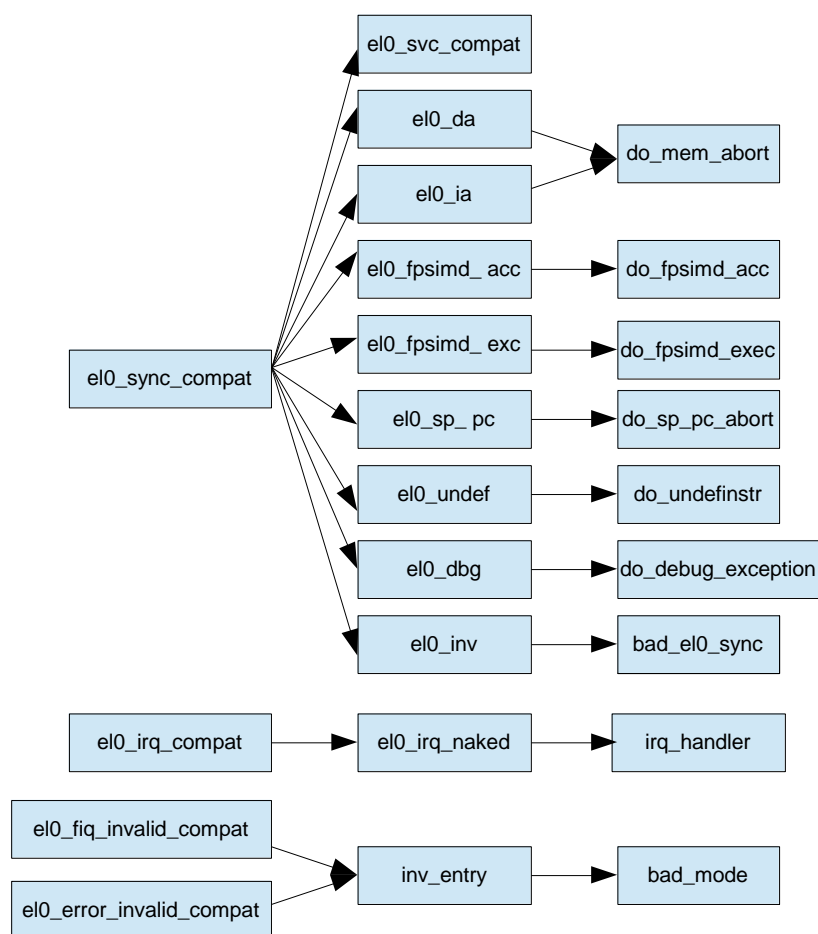


图 8. EL1 中处理 EL0 的异常，且 EL0 运行在 32bit

上图是异常发生在 EL0(运行在 32bit)且交由 EL1 处理的情况下，进入异常向量表后的代码执行流程。这里的处理函数与 64bit EL0 下基本一致，只有系统调用和中断处理有单独的处理函数 `el0_svc_compat` 和 `el0_irq_naked`。

## 第五章 异常触发及原理分析

### 5.1 Synchronous exception

#### 5.1.1 MMU fault

在 ARM Spec<sup>A</sup> 的 D4.5 中，有对 MMU fault 的定义。简单来说就是在当前 translation table lookup 过程中，MMU 的检测机制检测到存在出错的情况。ARM64 下 MMU fault 总共有以下 7 类：

- ※ Alignment fault
- ※ Permission fault
- ※ Translation fault

- ※ Address size fault
- ※ Synchronous external abort on translation table walk
- ※ Access flag fault
- ※ TLB conflict fault

### 5.1.1.1 Address size fault

#### A. 什么是 address size fault

在 ARM Spec<sup>A</sup> 的 D4.5.1 中，有对 address size fault 的准确定义。简单来说就是在遇到下面一种情况时，会触发 address size fault：

- a. 对应 ttbr 中超出 output address size 的 bit 为非 0，触发的是 ttbr address size fault
- b. 某个 level 表项的 output address 域中超出 output address size 的 bit 为非 0，触发的是该 level 的 address size fault
- c. 某个 level 表项的 Next-level table address 域中超出 output address size 的 bit 为非 0，触发的是该 level 的 address size fault

Output address size 是通过 TCR\_EL1.IPS 控制的，目前配置的是 40 bits，因此只要 TTRB、output address 域或者 Next-level table address 域中的 bit[47:40] 中有一个 bit 非 0，就会触发相应的 address size fault。

#### B. Linux 中对 Address size fault 的处理

Linux ARM64(未特别加说明的情况下，均为 Linux ARM64，下同)下，对 address size fault 的处理函数为 do\_bad，该函数直接返回 1，表示 Linux 无法处理该类异常。然后交由 MMU fault 的默认处理程序执行，即 arm64\_notify\_die。

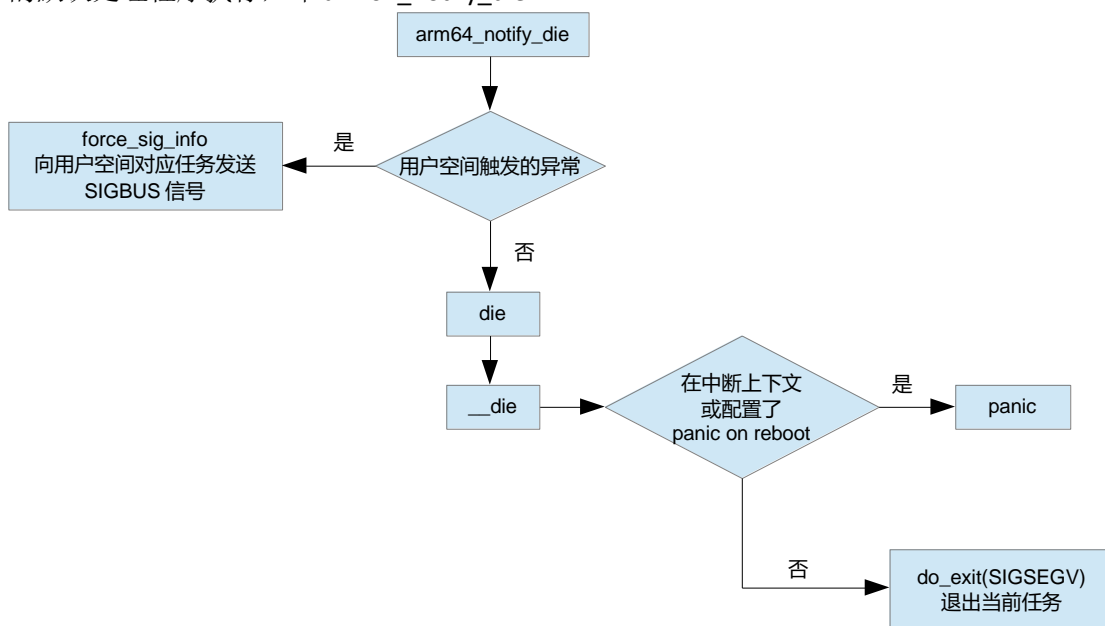


图 9-a. arm\_notify\_die 代码执行流程

## Exception trigger 模块说明文档

Arm\_notify\_die 中会首先判断异常来自用户空间还是内核空间。如果来自用户空间，则直接调用 force\_sig\_info 向当前任务发送 SIGBUS 信号；如果来自内核空间，则会调用 die 函数。该函数会首先调用 \_\_die 函数打印辅助信息，具体见下图，然后根据是否在中断上下文或者配置了 CONFIG\_PANIC\_ON\_OOPS\_VALUE 来决定是触发系统 panic 还是仅仅退出当前任务。

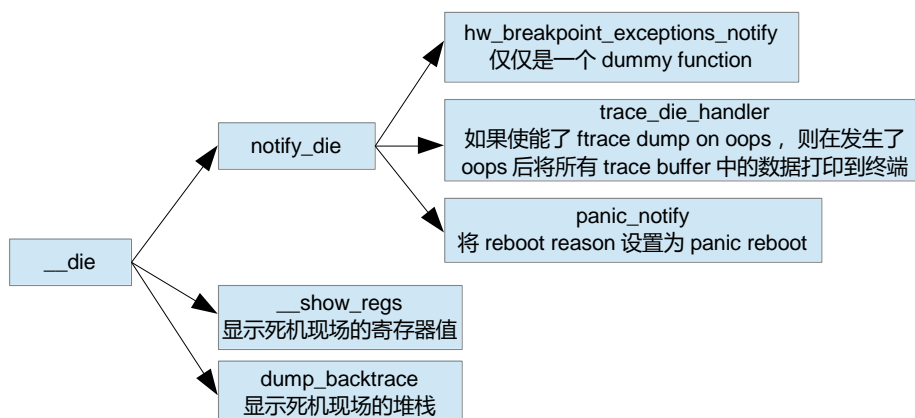


图 9-b. \_\_die 函数代码执行流程

在 `__die` 函数中会打印 trace buffer 中的数据(如果有使能 `ftrace_dump_on_oops`)，crash 现场的寄存器信息以及堆栈，这将有助于分析引起异常的原因。

### C. Exception trigger 如何制造 address size fault

目前已支持触发的 address size fault 为 user/kernel space ttbr、level 2 以及 level 3 address size fault。相关代码位于 `exp_trigger/driver/src/sync_faults/mmu_faults/address_size_fault.c`。

#### user space ttbr address size fault

代码实现如下：

```
ttbr0_el1 = read_sysreg(ttbr0_el1);
ttbr0_el1_tmp = ttbr0_el1 | ((unsigned long)1 << (oas+2));
write_sysreg(ttbr0_el1_tmp, ttbr0_el1);
```

这里的 `oas` 是 `output address size` 的缩写，我们强制修改 `ttbr0` 的 bit 42 为 1。由于用户空间的每个进程有单独的地址空间，因此后续有对该地址空间地址(也就是 `exp_trigger_demo` 的地址空间)的访问，均会触发 `user space ttbr address size fault`。

#### user space level 2 address size fault

代码实现如下：

```
pmd = get_tbl_ent(mm, e_data.addr_in, TABLE_ENTRY_PMD);
pmd->pmd = pmd_val(*pmd) | ((unsigned long)1 << (oas+2));
```

这里 `e_data.addr_in` 是用户空间一个变量的虚拟地址，我们将该虚拟地址对应 PMD table

## Exception trigger 模块说明文档

entry Next-level table address 域的 bit 42 设置为 1。之后用户空间尝试访问该变量或者访问同一地址空间里使用同一 pmd 表项的地址，就会触发 user space level 2 address size fault。

### user space level 3 address size fault

代码实现如下：

```
用户空间：
p->addr_in = (unsigned long)sbrk(4 * 1024);
内核空间：
pte = get_tbl_ent(mm, e_data.addr_in, TABLE_ENTRY_PTE);
pte->pte = pte_val(*pte) | ((unsigned long)1 << (oas+2));
用户空间：
((int *) (p->addr_in))[0] = 0x4321;
```

这里首先在用户空间的堆里分配一个 Page 大小的空间，这样就可以在页表里独自占有一个 PTE page entry，后续异常的触发也可以局限在对该 Page 的访问中，比较独立。接着在内核中，修改该 page 对应 PTE page entry output address 域的 bit 42 为 1。最后，尝试访问该 Page，会触发 user space level 3 address size fault。

### kernel space ttbr address size fault

代码实现如下：

```
ttbr1_el1 = read_sysreg(ttbr1_el1);
ttbr1_el1_tmp = ttbr1_el1 | ((unsigned long)1 << (oas+2));
write_sysreg(ttbr1_el1_tmp, ttbr1_el1);
```

我们强制修改 ttbr1 的 bit 42 为 1。由于 Linux 内核使用同一个地址空间，因此后续对任何内核地址的访问，均会触发 kernel space ttbr address size fault。

### kernel space level 2 address size fault

代码实现如下：

```
p = kmalloc(PTRS_PER_PTE*PAGE_SIZE, GFP_KERNEL);
pmd = get_tbl_ent(mm, (unsigned long)p, TABLE_ENTRY_PMD);
pmd->pmd = pmd_val(*pmd) | ((unsigned long)1 << (oas+2));
p[0] = 0x12;
```

这里首先在内核空间分配一个 2M 大小的 memory block，这样就可以在页表里独自占有一个 PMD block entry，后续异常的触发也可以局限在对该 memory block 的访问中。接着修改该 memory block 对应 PMD block entry output address 域的 bit 42 为 1。最后，尝试访问该 memory block 会触发 kernel space level 2 address size fault。

### kernel space level 3 address size fault

代码实现如下：

## Exception trigger 模块说明文档

```
p = vmalloc(PAGE_SIZE);
pte = get_tbl_ent(mm, (unsigned long)p, TABLE_ENTRY_PTE);
pte->pte = pte_val(*pte) | ((unsigned long)1 << (oas+2));
p[0] = 0x12;
```

这里首先在内核空间分配一个 Page,这样就可以在页表里独自占有一个 PTE block entry,后续异常的触发也可以局限在对该 page 的访问中。接着修改该 page 对应的 PTE page entry output address 域的 bit 42 为 1。最后,尝试访问该 page 会触发 kernel space level 3 address size fault。

### D. 运行结果

这里我们使用提供的用户空间程序 exp\_trigger\_demo 来选择触发不同类型的 address size fault:

#### user space ttbr address size fault

```
[ 28.634359@2] exp_trigger: output address size: 40
[ 28.635809@2] exp_trigger: current task: exp_trigger_dem
[ 28.641073@2] exp_trigger: addr_in: 0x7fca358804
[ 28.645626@2] exp_trigger: try to trigger user space tbr address size fault exception
[ 28.653379@2] exp_trigger: ttbr0_el1: 0x5c989000
[ 28.657957@2] exp_trigger: new ttbr0_el1: 0x4005c989000
[ 28.663135@2] Unhandled fault: ttbr address size fault (0x96000040) at 0x0000007fca358840
[ 28.671230@2] Internal error: : 96000040 [#1] PREEMPT SMP
```

图 10-a. user space ttbr address size fault 异常现场

在上图中,我们可以看到触发异常时访问的地址是 0x7fca358840,与 exp\_trigger\_demo 中尝试访问的地址 0x7fca358804 并不相同。这是因为我们修改了 ttbr0 的内容,对 exp\_trigger\_demo 所在地址空间的访问均会触发 user space ttbr address size fault。在当前的 case 里,应该是在尝试访问 0x7fca358804 前已经先访问了 0x7fca358804。

#### user space level 2 address size fault

```
[ 569.141532@3] exp_trigger: output address size: 40
[ 569.142939@3] exp_trigger: current task: exp_trigger_dem
[ 569.148266@3] exp_trigger: addr_in: 0x7ff72c2c54
[ 569.152835@3] exp_trigger: try to trigger user space level 2 address size fault exception
[ 569.160924@3] exp_trigger: pmd value: 0x5c8d2003
[ 569.165477@3] exp_trigger: new pmd value: 0x4005c8d2003
[ 569.170643@3] Unhandled fault: level 2 address size fault (0x96000042) at 0x0000007ff72c2c90
[ 569.179040@3] Internal error: : 96000042 [#2] PREEMPT SMP
```

图 10-b. user space level 2 address size fault 异常现场

这里触发异常时访问的地址 0x7ff72c2c90 与我们尝试访问的地址 0x7ff72c2c54 不同的原因,和 user space ttbr address size fault 的类似。

## Exception trigger 模块说明文档

### user space level 3 address size fault

```
[ 38.291717@0] exp_trigger: output address size: 40
[ 38.292912@0] exp_trigger: current task: exp_trigger_dem
[ 38.298225@0] exp_trigger: addr_in: 0x353f8000
[ 38.302639@0] exp_trigger: try to trigger user space level 3 address size fault exception
[ 38.310810@0] exp_trigger: pte value: 0xe800000539ef53
[ 38.315804@0] exp_trigger: new pte value: 0xe804000539ef53
try to access 0x353f8000
[ 39.321355@0] Unhandled fault: level 3 address size fault (0x92000043) at 0x00000000353f8000
```

图 10-c. user space level 2 address size fault 异常现场

可以看到在尝试访问对应页描述符 Output address 域的第 42bit 非 0 的用户空间虚拟地址 0x353f8000 后触发了 user space level 3 address size fault。

### kernel space ttbr address size fault

```
[ 16.841095@3] exp_trigger: output address size: 40
[ 16.841130@3] exp_trigger: try to trigger kernel space tbr address size fault exception
[ 16.848283@3] exp_trigger: ttbr1_ell: 0x73000002b1d000
[ 16.853342@3] exp_trigger: new ttbr1_ell: 0x73040002b1d000
[ 16.862659@3] Unhandled fault: ttbr address size fault (0x96000000) at 0xfffff800a7e5048
```

图 10-d. kernel space ttbr address size fault 异常现场

这里的情况与 user space ttbr address size fault 类似。

### kernel space level 2 address size fault

```
[ 23.403426@1] exp_trigger: output address size: 40
[ 23.403460@1] exp_trigger: try to trigger kernel space level 2 address size fault exception
[ 23.410977@1] exp_trigger: pmd value: 0xe800005ce00711
[ 23.416059@1] exp_trigger: new pmd value: 0xe804005ce00711
[ 23.421453@1] exp_trigger: try to access fffffffc05ce0000
[ 23.426800@1] Unhandled fault: level 2 address size fault (0x96000042) at 0xffffffc05ce00000
[ 23.435168@1] Internal error: : 96000042 [#1] PREEMPT SMP
```

图 10-e. kernel space level 2 address size fault 异常现场

可以看到在尝试访问对应块描述符 Output address 域的第 42bit 非 0 的内核空间虚拟地址 0xffffffc05ce00000 后触发了 kernel space level 2 address size fault。

### kernel space level 3 address size fault

```
[ 30.569286@1] exp_trigger: output address size: 40
[ 30.569320@1] exp_trigger: try to trigger kernel space level 3 address size fault exception
[ 30.576861@1] exp_trigger: pte value: 0xe800005cc3a713
[ 30.581896@1] exp_trigger: new pte value: 0xe804005cc3a713
[ 30.587312@1] exp_trigger: try to access fffffff80089ee000
[ 30.592668@1] Unhandled fault: level 3 address size fault (0x96000043) at 0xfffff80089ee0000
[ 30.601027@1] Internal error: : 96000043 [#1] PREEMPT SMP
```

图 10-f. kernel space level 3 address size fault 异常现场



## Exception trigger 模块说明文档

可以看到在尝试访问对应页描述符 Output address 域的第 42bit 非 0 的内核空间虚拟地址 0xffffffff80089ee000 后触发了 kernel space level 3 address size fault

### 5.1.1.2 Translation fault

#### A. 什么是 translation fault

在 ARM Spec<sup>A</sup> D4.5.1 的 Translation fault 小节中有对 translation fault 的准确定义。简单来说就是在遇到下面两种情况时会触发 translation fault:

- a. 输入的虚拟地址没有在对应在 TTBR 的范围内，触发的是 level 0 translation fault
- b. 在查表过程中遇到了无效或保留的描述符，触发的是 level 1~3 translation fault

TTBR0 和 TTBR1 的覆盖范围分别由 TCR\_EL1.T0SZ 和 TCR\_EL1.T1SZ 域设置。两者地址空间覆盖范围的大小分别为  $2^{(64-TCR\_EL1.T0SZ)}$  字节和  $2^{(64-TCR\_EL1.T1SZ)}$  字节。在我选用的 Ampere 平台中，两者的值均为 011001，即 25。因此 TTBR0 的虚拟地址有效地址范围为 0x0000000000000000~0x0000007FFFFFFFFF，TTBR1 的虚拟地址有效覆盖范围为 0xffffffff800000000 ~ 0xffffffff ffffffff。只要输入的用户空间或者内核空间的虚拟地址不在前面两个范围内，即会触发 level 0 translation fault。

对于页表中的描述符，如果其位于 level 0~2，那么只要 bit[0]为 0，即为无效的；如果其位于 level 3，那么只要 bit[1:0]非 11 即为无效或者保留的。在查表过程中只要在任意 level 中遇到了这类描述符，就会触发对应 level 的 translation fault。

#### B. Linux 中对 translation fault 的处理

按触发 translation fault 的行为位于用户空间还是内核空间，其访问的地址位于用户空间还是内核空间，大致可以分为如下 4 类

- ❖ 在用户空间访问用户空间地址
- ❖ 在用户空间访问内核空间地址
- ❖ 在内核空间访问用户空间地址
- ❖ 在内核空间访问内核空间地址

在上述 4 种情况中，只有在用户空间访问用户空间的地址以及内核空间通过 copy\_from\_user 之类的操作，引起的 translation fault 才是正常的缺页异常(不考虑 VMAP\_STACK 的情况下)，其余均为异常访问。

## 访问用户空间地址触发 translation fault

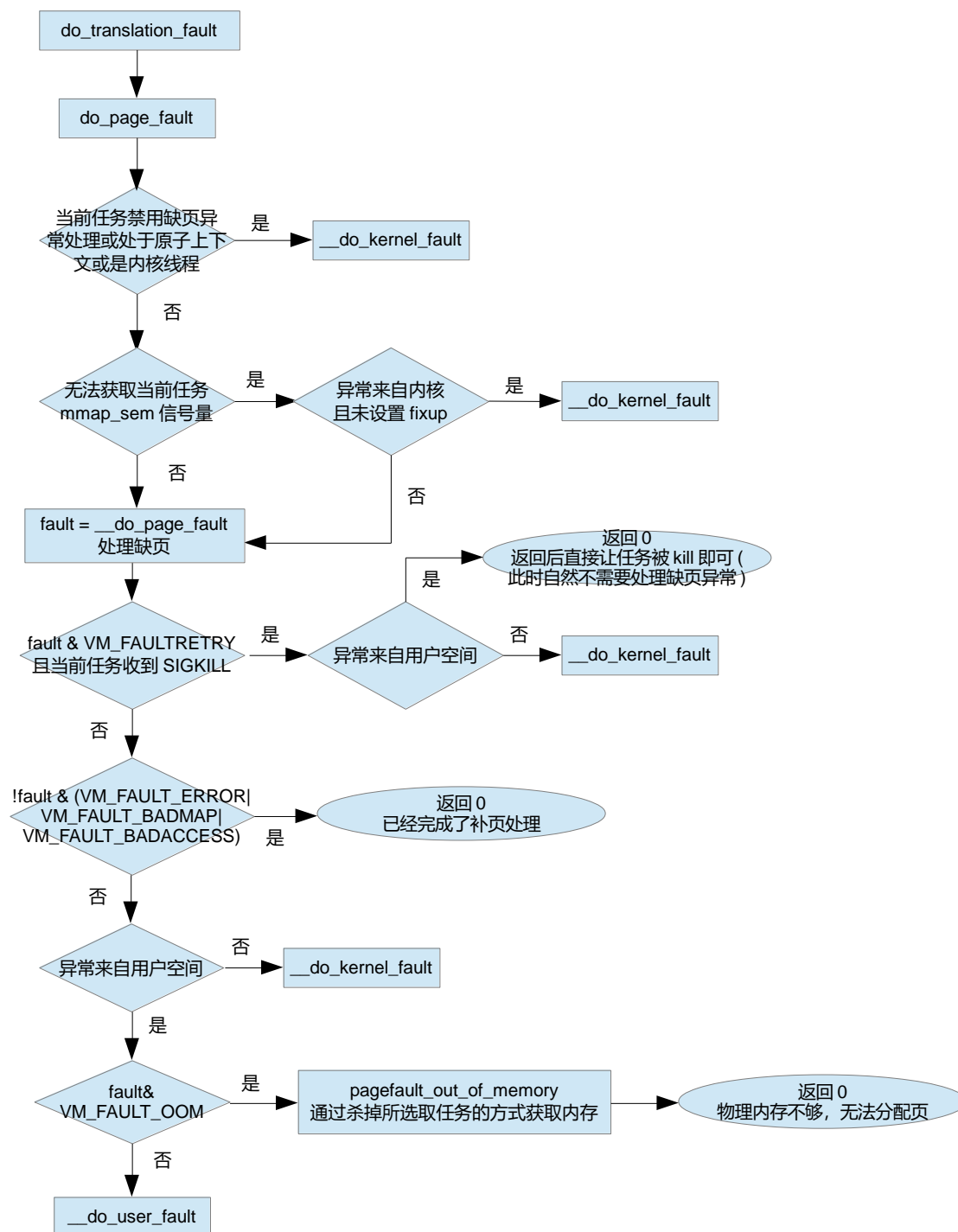


图 11-a. do\_page\_fault 执行流程

## Exception trigger 模块说明文档

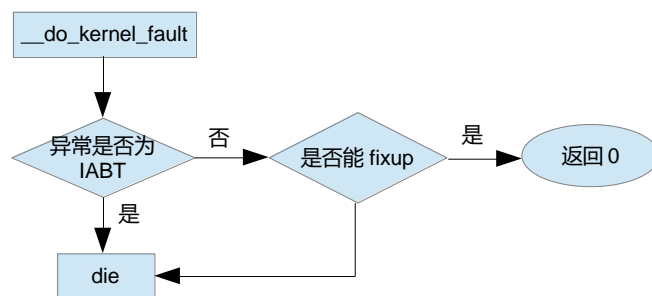


图 11-b. \_\_do\_kernel\_fault 执行流程

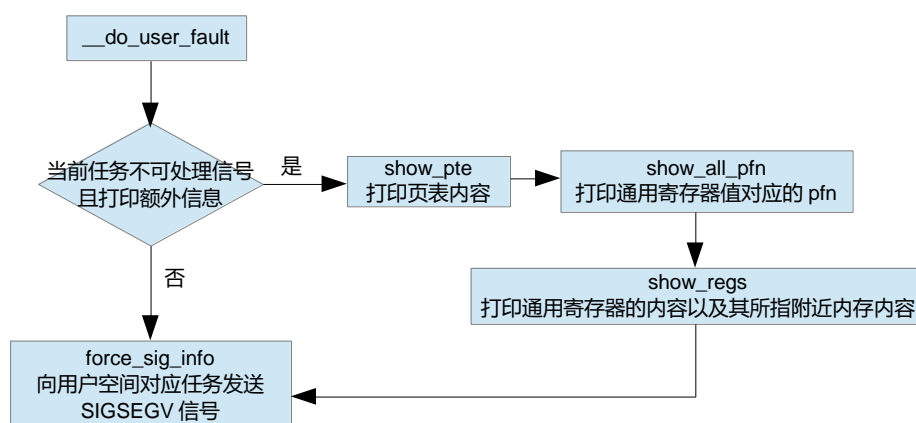


图 11-c. \_\_do\_user\_fault 执行流程

内核中处理各类 translation fault 的总入口为 do\_translation\_fault。在检测到访问的地址位于用户空间后，会调用 do\_page\_fault。该函数首先会进行初步的过滤，对于当前任务明确标记禁止处理缺页异常，当前位于原子上下文或者当前任务是内核线程时(对于第一种情况，有的内核为了验证某个功能是否支持或者为了简化代码，会选择主动触发 translation fault 并告知内核忽略<sup>[附录 1]</sup>，对于第 2 种情况由于原子上下文中不允许休眠，而 ARM 的异常处理函数中是允许休眠的，对于第 3 种情况内核线程访问另一个无关任务的地址空间本身就是不合理的)，判定不是合理的缺页异常，会调用 \_\_do\_kernel\_fault 处理。

\_\_do\_kernel\_fault 会调用 fixup\_exception 去查询被访问的地址是否在 exception table 中，如果是，则将 regs->pc 替换为 exception table 中被访问地址对应的 pc 值，直接退出 translation fault 处理并跳转到 regs->pc 中继续运行；否则直接调用 die，对于配置了 panic on oops 的情况会触发 panic，否则触发 Oops 并向当前任务发送 SIGSEGV 信号。

如果通过了初步过滤，则会尝试去获取当前任务的 mmap\_sem 信号量，如果能获取到则直接进入缺页异常的处理流程(\_\_do\_page\_fault)，否则先检查是否是内核空间触发的异常且有设置 fixup(对于系统调用进入内核，并在内核中使用 copy\_from\_user 之类的函数访问用户地址空间就是这种情况，copy\_from\_user 会主动将被访问的用户空间地址加入 exception tables 中)或者是来自用户空间的异常，如果有会冒着发生信号量死锁的可能调用 down\_read(有可能进入 uninterruptible sleep 状态)再次尝试获取 mmap\_sem 信号量，成功后进入缺页异常处理，否则直接调用 \_\_do\_kernel\_fault。

\_\_do\_page\_fault 的具体实现后面分析，先来看看对其返回值的处理。首先如果其返回值为 VM\_FAULT\_RETRY，但是当前任务收到了 SIGKILL 信号，如果异常来自用户空间则直接退出 do\_translation\_fault(说明触发 translation fault 的任务本身收到了 SIGKILL 信号，直接让该

## Exception trigger 模块说明文档

任务被 kill 即可), 如果异常来自内核则直接调用 `__do_kernel_fault`(内核空间尝试访问的一个收到了 kill 信号的任务的地址, 本身不合理)。其它情况下只要返回值不是 `VM_FAULT_ERROR`, `VM_FAULT_ERROR` 或者 `VM_FAULT_BADACCESS` 中的一个, 均表示补页成功直接返回 0。其它情况下, 如果异常来自内核空间, 则调用 `__do_kernel_fault`, 否则先检查是否返回值为 `VM_FAULT_OOM`(表示物理内存不够), 如果是则调用 `pagefault_out_of_memory` 通过 kill 掉部分任务的方法获取物理内存, 然后返回。默认情况下最后会调用 `__do_user_fault`。

`__do_user_fault` 在向当前任务发送对应信号前, 会先检查是否需要打印不可处理信号的额外信息。如果是, 则先打印如页表内容, 通用寄存器值对应的 PFN 以及通用寄存器的内容及其所指附近内存内容等辅助信息。最后会向用户空间的任務发送对应的信号。

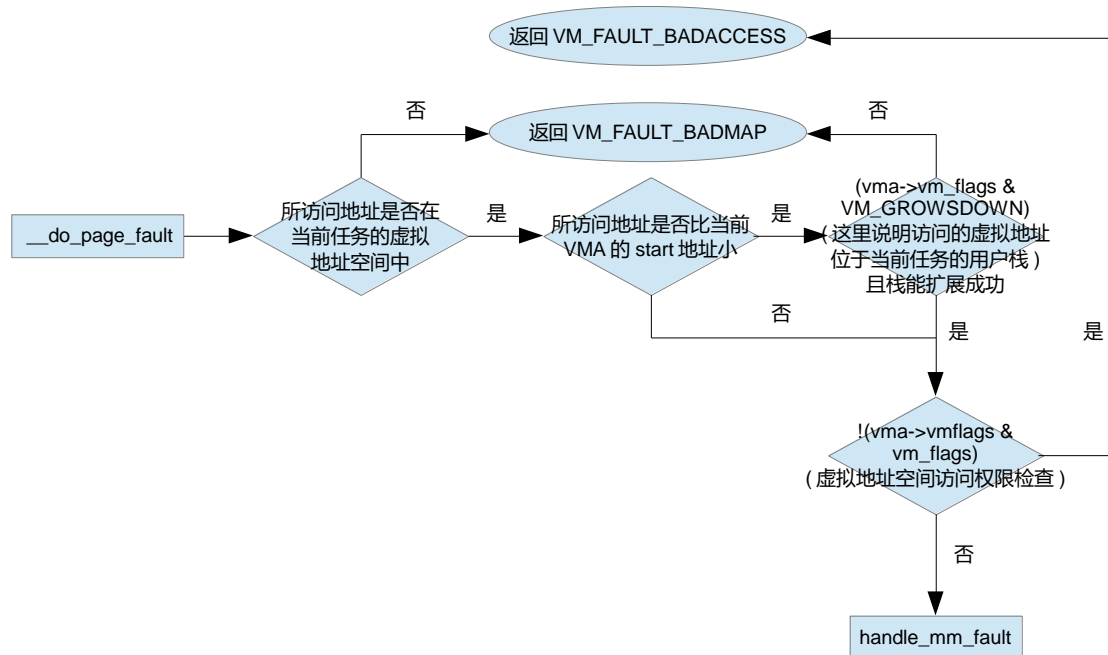


图 11-d. `__do_page_fault` 执行流程

处理缺页异常的核心函数是 `__do_page_fault`, 该函数会作一些初步的检查, 然后调用 `handle_mm_fault` 去处理各种不同类型的缺页。这里的过滤检查包括 3 个方面, 首先是通过检查访问的地址是否位于当前任务的虚拟地址空间, 来判断该地址是否是合理的虚拟地址; 其次是当该地址比所在虚拟地址空间的起始地址小时, 需要判断该虚拟地址空间是否设置了 `VM_GROWNSDOWN`(有该标记说明所处的虚拟地址空间是用户栈), 且是否能成功扩展; 最后会将对地址尝试的访问与所处虚拟地址空间允许的访问作比较, 查看是否进行了非法的访问。只有通过了这些检查, 才会调用 `handle_mm_fault` 处理缺页, 否则返回对应的错误。

## Exception trigger 模块说明文档

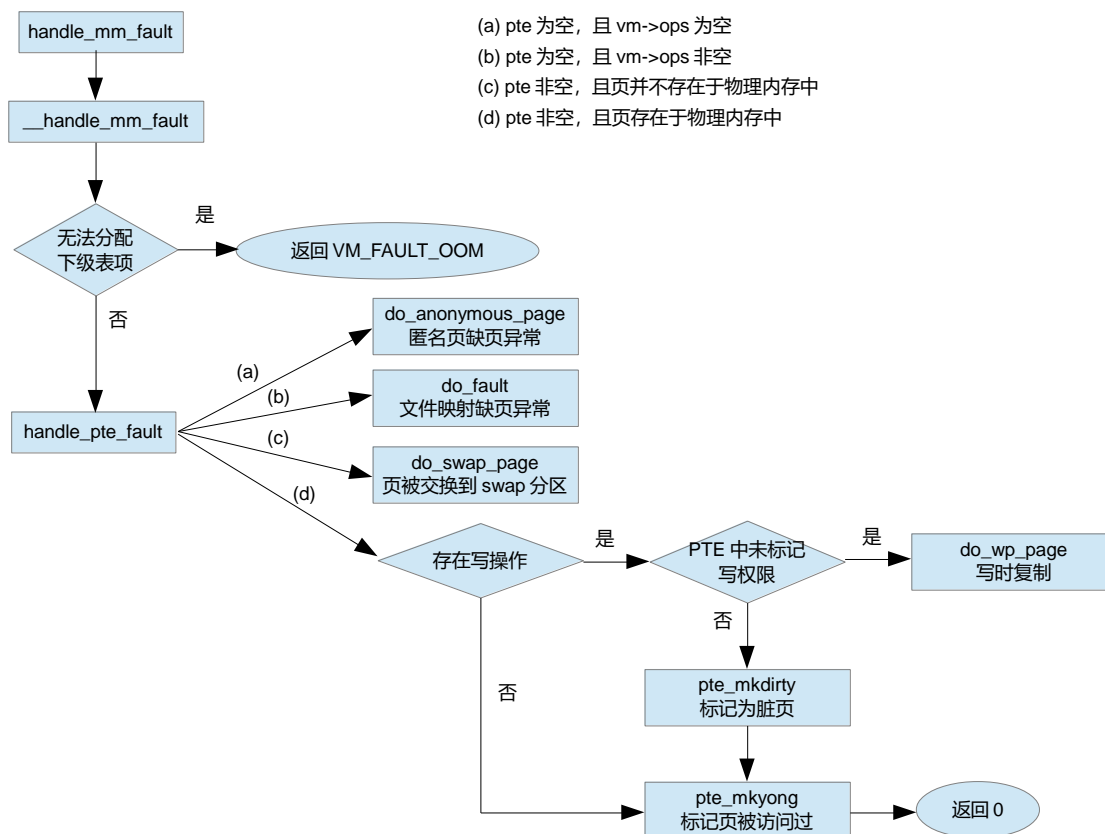


图 11-e. handle\_mm\_fault 执行流程

handle\_mm\_fault 是处理缺页的核心函数, 上图的执行流程中暂时忽略了大页, 透明巨页以及 NUMA balancing(使任务与其使用的内存位于一个 NUMA 节点上, 达到最高的执行效率)(Amlogic 目前使用的内核上, 均未使能这 3 个对应的配置)。该函数在调用 handle\_pte\_fault 处理缺页前, 会先检查是否能分配下级页表项, 如果不能则说明物理内存已经不够需返回错误 VM\_FAULT\_OOM, 否则调用 handle\_pte\_fault 处理不同类型的缺页。

如果对应的 PTE 为空, 且所在虚拟地址空间的文件操作集合为空, 说明是匿名页缺页异常, 调用 do\_anonymous\_page 处理; 如果对应的 PTE 为空, 且所在的虚拟地址空间的文件操作集合非空, 说明是文件映射缺页异常, 调用 do\_fault 处理; 如果对应的 PTE 非空, 且页不存在于物理内存中, 说明页被交换到了 swap 分区, 调用 do\_swap\_page 处理; 如果对应的 PTE 非空, 且页存在于物理内存中, 说明在进行写时复制, 调用 do\_wp\_page 处理。其余情况下, 会调用 pte\_mkyoung 标记页被访问过, 同时如果对页进行写操作会调用 pte\_mkdirty 标记为脏页。

### 访问内核空间地址触发 translation fault

在不考虑 VMAP STACK(任务的内核栈通过 vmalloc 分配, 且初始时只分配 1 个 page, 在访问超过 1 个 page 的内核栈时, 会触发缺页异常, 其实现非内核原生, 暂时不考虑)的情况下, 访问内核空间地址触发的 translation fault 均为真正的访问错误。对应的处理函数是 do\_bad\_area, 该函数会判断异常来自用户空间还是内核空间, 如果是前者则调用 \_\_do\_user\_fault, 否则调用 \_\_do\_kernel\_fault。

## Exception trigger 模块说明文档

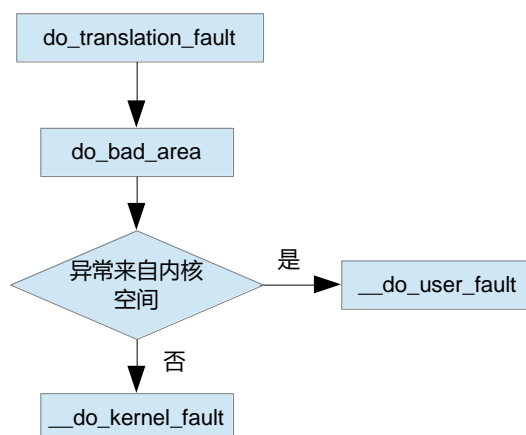


图 11-f. `do_bad_area` 执行流程

### C. Exception trigger 如何制造 translation fault

目前已经支持触发的 translation fault 为 user/kernel space level 0~3 translation fault 这 8 种 translation fault。对于 ttbr translation fault，可基于查询到的 ttbr 范围来生成一个不在该范围内的地址，并强制去访问它来触发。对于其余各级 translation fault，可以通过遍历各级页表找到不存在映射的项，倒推出对应的虚拟地址，并强制去访问该地址来触发。相关代码位于 `exp_trigger/driver/src/sync_faults/mmu_faults/translation_fault.c` 以及 `exp_trigger/driver/src/sync_faults/mmu_faults/translation_table_helper.c`。

#### user space level 0 translation fault

代码实现如下：

```
get_ttbr_range(0, &start, &end)
fault_trigger_address = end + 0x1000;
e_data.addr_out = fault_trigger_address;
copy_to_user(data, &e_data, sizeof(e_data));
```

这里首先通过 `get_ttbr_range` 获取 user space 的 ttbr 区间，并将获得区间的最大值再加上 `0x1000` 作为提供给用户空间访问的地址，用户空间访问该地址后，即可触发 user space level 0 translation fault。

#### user space level 1 translation fault

代码实现如下：

```
pud = first_cond_pud(mm, 0, &ig);
fault_trigger_address = 0x0000000000000000 |
    (((unsigned long)ig.pud_i) << PUD_SHIFT);
e_data.addr_out = fault_trigger_address;
copy_to_user(data, &e_data, sizeof(e_data));
```

这里首先通过 `first_cond_pud` 获取第一个无效的 pud 表项，然后计算出一个为该 pud 表项的用户空间虚拟地址(这里截取的是 3 级页表情况的下的计算方法，实际的代码实现中会

## Exception trigger 模块说明文档

考虑多种情况，下同)，并将该地址传给用户空间。用户空间访问该地址后，即可触发 user space level 1 translation fault。

### user space level 2 translation fault

代码实现如下：

```
pmd = first_cond_pmd(mm, 0, &ig);
fault_trigger_address = 0x0000000000000000 |
    (((unsigned long)ig.pgd_i) << PGDIR_SHIFT) |
    (((unsigned long)ig.pmd_i) << PMD_SHIFT);
e_data.addr_out = fault_trigger_address;
copy_to_user(data, &e_data, sizeof(e_data));
```

这里首先通过 first\_cond\_pmd 获取第一个无效的 pmd 表项，然后计算出一个为该 pmd 表项的用户空间虚拟地址，并将该地址传给用户空间。用户空间访问该地址后，即可触发 user space level 2 translation fault。

### user space level 3 translation fault

代码实现如下：

```
pte = first_cond_pte(mm, 0, &ig);
fault_trigger_address = 0x0000000000000000 |
    (((unsigned long)ig.pgd_i) << PGDIR_SHIFT) |
    (((unsigned long)ig.pmd_i) << PMD_SHIFT) |
    (((unsigned long)ig.pte_i) << PAGE_SHIFT);
e_data.addr_out = fault_trigger_address;
copy_to_user(data, &e_data, sizeof(e_data));
```

这里首先通过 first\_cond\_pte 获取第一个无效的 pte 表项，然后计算出一个为该 pte 表项的用户空间虚拟地址，并将该地址传给用户空间。用户空间访问该地址后，即可触发 user space level 3 translation fault。

### kernel space level 0 translation fault

代码实现如下：

```
get_ttbr_range(1, &start, &end)
fault_trigger_address = start - 0x1000;
*((unsigned long *)fault_trigger_address) = 0x1234;
```

这里首先通过 get\_ttbr\_range 获取 kernel space 的 ttbr 区间，并将获得区间的最小值减去 0x1000 作为给内核空间访问的地址，内核空间访问该地址后，就会触发 kernel space level 0 translation fault。

### kernel space level 1 translation fault

代码实现如下：

## Exception trigger 模块说明文档

```
pud = first_cond_pud(&init_mm, 0, &ig);
fault_trigger_address = 0xffffffff8000000000 |
                        (((unsigned long)ig.pud_i) << PUD_SHIFT);
*((unsigned long *)fault_trigger_address) = 0x1234;
```

这里首先通过 `first_cond_pud` 获取第一个无效的 `pud` 表项，然后计算出一个为该 `pud` 表项的内核空间虚拟地址，内核空间访问该地址后，即可触发 `kernel space level 1 translation fault`。

### kernel space level 2 translation fault

代码实现如下：

```
pmd = first_cond_pmd(&init_mm, 0, &ig);
fault_trigger_address = 0xffffffff8000000000 |
                        (((unsigned long)ig.pgd_i) << PGDIR_SHIFT) |
                        (((unsigned long)ig.pmd_i) << PMD_SHIFT);
*((unsigned long *)fault_trigger_address) = 0x1234;
```

这里首先通过 `first_cond_pmd` 获取第一个无效的 `pmd` 表项，然后计算出一个为该 `pmd` 表项的内核空间虚拟地址。用户空间访问该地址后，即可触发 `kernel space level 2 translation fault`。

### kernel space level 3 translation fault

代码实现如下：

```
pte = first_cond_pte(&init_mm, 0, &ig);
fault_trigger_address = 0xffffffff8000000000 |
                        (((unsigned long)ig.pgd_i) << PGDIR_SHIFT) |
                        (((unsigned long)ig.pmd_i) << PMD_SHIFT) |
                        (((unsigned long)ig.pte_i) << PAGE_SHIFT);
*((unsigned long *)fault_trigger_address) = 0x1234;
```

这里首先通过 `first_cond_pte` 获取第一个无效的 `pte` 表项，然后计算出一个为该 `pte` 表项的内核空间虚拟地址。内核空间访问该地址后，即可触发 `kernel space level 3 translation fault`。

## D. 运行结果

### user space level 0 translation fault

```
[ 21.375768@0] exp_trigger: try to trigger user space level 0 translation fault
[ 21.377381@0] exp_trigger: start: 0x0, end: 0x7fffffffff
[ 21.382750@0] exp_trigger: fault trigger address: 0x8000000fff
fault_trigger_address: 0x8000000fff
[ 22.388585@0] exp_trigger_dem[2268]: unhandled level 0 translation fault (11) at 0x8000000fff, esr 0x92000044
[ 22.392884@0] pgd = fffffffc05cd2c000
[ 22.396419@0] [8000000fff] *pgd=000000005cd40003, *pud=000000005cd40003, *pmd=0000000000000000
```

图 12-a. user space level 0 translation fault 异常现场



## Exception trigger 模块说明文档

可以看到在用户空间访问了不属于 ttbr0 区间的地址 0x8000000fff 后,触发了 user space level 0 translation fault。

### user space level 1 translation fault

```
[ 17.124366@03] exp_trigger: try to trigger user space level 1 translation fault
[ 17.125981@03] pgd[0]: 0x5cd4f003
[ 17.129236@03] pgd[1]: 0x0
[ 17.131811@03] exp_trigger: fault trigger address: 0x40000000
fault_trigger_address: 0x40000000
[ 18.137523@03] exp_trigger_dem[2262]: unhandled level 1 translation fault (11) at 0x40000000, esr 0x92000045
[ 18.141646@03] pgd = fffffffc05cc6f000
[ 18.145533@03] [40000000] *pgd=0000000000000000, *pud=0000000000000000
```

图 12-b. user space level 1 translation fault 异常现场

可以看到在访问了第 1 级页表描述符为空的用户空间地址 0x40000000 后,触发了 user space level 1 translation fault。

### user space level 2 translation fault

```
[ 25.371969@00] exp_trigger: try to trigger user space level 2 translation fault
[ 25.373624@00] pgd[0]: 0x5cd5b003
[ 25.376820@00] pmd[0]: 0x0
[ 25.379389@00] exp_trigger: fault trigger address: 0x0
fault_trigger_address: 0x0
[ 26.384508@00] exp_trigger_dem[2278]: unhandled level 2 translation fault (11) at 0x00000000, esr 0x92000046
[ 26.388609@00] pgd = fffffffc05cd6c000
[ 26.392375@00] [00000000] *pgd=000000005cd5b003, *pud=000000005cd5b003, *pmd=0000000000000000
```

图 12-c. user space level 2 translation fault 异常现场

可以看到在访问了第 2 级页表描述符为空的用户空间地址 0x00 后,触发了 user space level 2 translation fault。

### user space level 3 translation fault

```
[ 25.940096@00] exp_trigger: try to trigger user space level 3 translation fault
[ 25.941711@00] pgd[0]: 0x5cd00003
[ 25.944969@00] pmd[0]: 0x0
[ 25.947526@00] pmd[1]: 0x0
[ 25.950070@00] pmd[2]: 0x5cc2f003
[ 25.953298@00] pte[0]: 0x200000748ebfd3
[ 25.957090@00] pte[1]: 0x200000748b4fd3
[ 25.961392@00] pte[2]: 0x200000748b5fd3
[ 25.964519@00] pte[3]: 0x200000748b6fd3
[ 25.968142@00] pte[4]: 0x0
[ 25.970718@00] exp_trigger: fault trigger address: 0x404000
fault_trigger_address: 0x404000
[ 26.976273@00] exp_trigger_dem[2273]: unhandled level 3 translation fault (11) at 0x00404000, esr 0x92000047
[ 26.980457@00] pgd = fffffffc05ccb9000
[ 26.983942@00] [00404000] *pgd=000000005cd00003, *pud=000000005cd00003, *pmd=000000005cc2f003, *pte=0000000000000000
```

图 12-d. user space level 3 translation fault 异常现场

可以看到在访问了第 3 级页表描述符为空的用户空间地址 0x404000 后,触发了 user space level 3 translation fault。

## Exception trigger 模块说明文档

### kernel space level 0 translation fault

```
[ 17.531355@3] exp_trigger: try to trigger kernel space level 0 translation fault
[ 17.533143@3] exp_trigger: start: 0xffffffff8000000000, end: 0xffffffffffffffff
[ 17.540288@3] exp_trigger: fault trigger address: 0xffffffff7ffffff000
[ 17.546550@3] exp_trigger: kernel level 0 translation fault
[ 17.552060@3] Unable to handle kernel paging request at virtual address fffffff7ffffff000
[ 17.560076@3] pgd = fffffffc05cd66000
[ 17.563626@3] [ffffff7ffffff000] *pgd=000000005cd76003, *pud=000000005cd76003, *pmd=0000000000000000
```

图 12-e. kernel space level 0 translation fault 异常现场

可以看到在内核空间访问了不属于 ttbr1 区间的地址 0xffffffff7ffffff000 后，触发了 kernel space level 0 translation fault。

### kernel space level 1 translation fault

```
[ 190.588313@0] exp_trigger: try to trigger kernel space level 1 translation fault
[ 190.590138@0] pgd[0]: 0x743fe003
[ 190.593353@0] pgd[1]: 0x0
[ 190.596139@0] exp_trigger: fault trigger address: 0xffffffff8040000000
[ 190.602251@0] exp_trigger: kernel level 1 translation fault
[ 190.607734@0] Unable to handle kernel paging request at virtual address ffffff8040000000
[ 190.615856@0] pgd = fffffffc05cc60000
[ 190.619290@0] [fffff80400000000] *pgd=0000000000000000, *pud=0000000000000000
```

图 12-f. kernel space level 1 translation fault 异常现场

可以看到在访问了第 1 级页表描述符为空的内核空间地址 0xffffffff8040000000 后，触发了 kernel space level 1 translation fault。

### kernel space level 2 translation fault

```
[ 2569.191202@2] exp_trigger: try to trigger kernel space level 2 translation fault
[ 2569.192990@2] pgd[0]: 0x743fe003
[ 2569.196228@2] pmd[0]: 0x0
[ 2569.199553@2] exp_trigger: fault trigger address: 0xffffffff8000000000
[ 2569.207471@2] exp_trigger: kernel level 2 translation fault
[ 2569.211223@2] Unable to handle kernel paging request at virtual address ffffff8000000000
[ 2569.219230@3] pgd = fffffffc05ccac000
[ 2569.222200@3] [fffff80000000000] *pgd=000000005ccb4003, *pud=000000005ccb4003, *pmd=0000000000000000
```

图 12-g. kernel space level 2 translation fault 异常现场

可以看到在访问了第 2 级页表描述符为空的内核空间地址 0xffffffff8000000000 后，触发了 kernel space level 2 translation fault。

## Exception trigger 模块说明文档

### kernel space level 3 translation fault

```
[ 2181.676392@1] exp_trigger: try to trigger kernel space level 3 translation fault
[ 2181.678208@1] pgd[0]: 0x743fe003
[ 2181.681369@1] pmd[0]: 0x0
[ 2181.684218@1] pmd[11]: 0x0
[ 2181.684952@1] pmd[63]: 0x0
[ 2181.851652@1] pmd[64]: 0x128003
[ 2181.854797@1] pte[0]: 0xe80000c4301707
[ 2181.858467@1] pte[1]: 0x0
[ 2181.861030@1] exp_trigger: fault trigger address: 0xfffff8008001000
[ 2181.867359@1] exp_trigger: kernel level 3 translation fault
[ 2181.872867@1] Unable to handle kernel paging request at virtual address ffffff8008001000
[ 2181.880892@1] pgd = ffffffc05cdaf000
[ 2181.884423@1] [fffff8008001000] *pgd=000000005cda3003, *pud=000000005cda3003, *pmd=0000000000000000
```

图 12-g. kernel space level 3 translation fault 异常现场

可以看到在访问了第 3 级页表描述符为空的内核空间地址 0xfffff8008001000 后，触发了 kernel space level 3 translation fault。

### 5.1.1.3 Access flag fault

#### A. 什么是 access flag fault

在 ARM Spec<sup>A</sup> D4.5.1 的 Access flag fault 小节中有对 access flag fault 的准确定义。简单来说就是访问的地址属于“OLD”页或内存块的，就会触发 access flag fault。一个“OLD”的页或内存块表明它还从未被访问过。另外，仅当其页描述符或块描述符的 Lower attributes 中的 AF(Access Flag)位为 0 时，才表示它是从未被访问过的。

AF 位的值会影响页的交换策略，当 AF 为 1 的页被“swapped out”的概率更低。

#### B. Linux 中对 access flag fault 的处理

在 Linux 下除了用户空间访问内核空间地址(会触发 permission fault，这类异常在 5.1.1.5 中有详细介绍)的情况外，其余 3 种访问情况均可能触发 access flag fault。另外访问用户空间地址引起的 access flag fault 才可能是正常的 access flag fault，在触发异常后会将对应的页描述符或块描述符中的 AF 位设置为 1，标识该内存块或页被访问过。

Access flag fault 在 Linux 中的处理函数为 do\_page\_fault，该函数的执行流程已经在上一节中有过详细描述。下面给出的是该函数中与 access flag fault 相关的执行流程：

## Exception trigger 模块说明文档

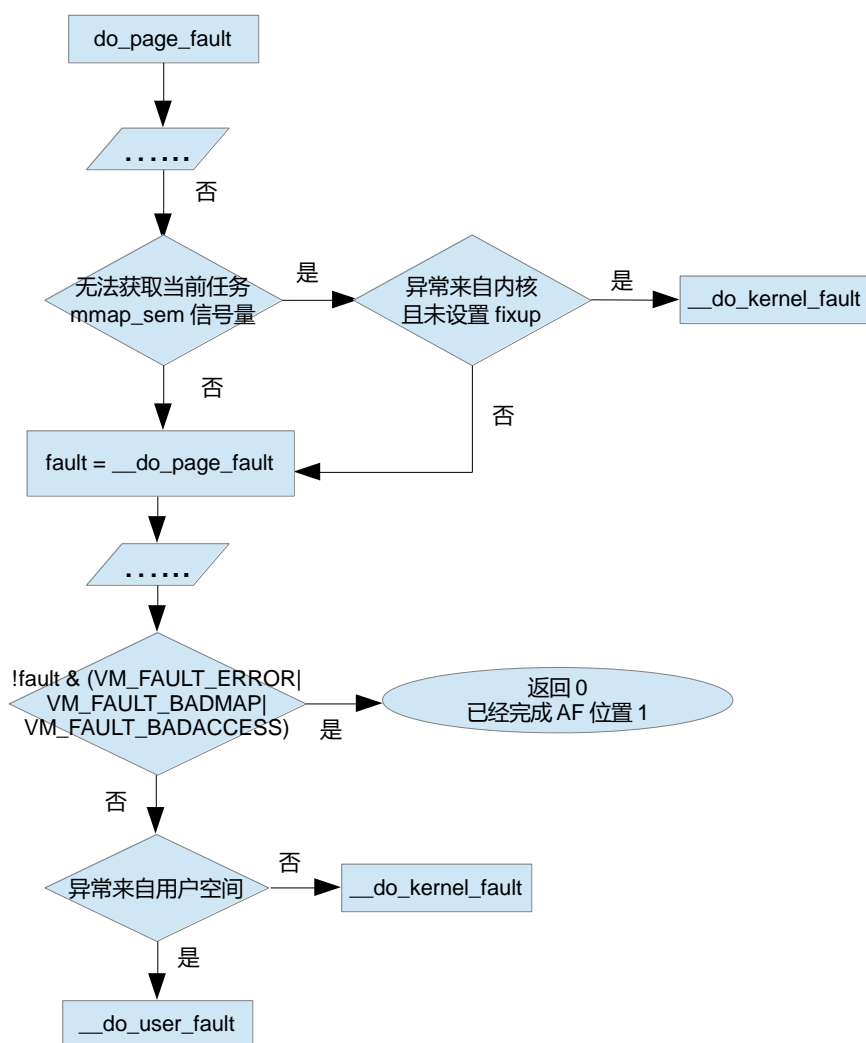


图 13. `do_page_fault` 中与 access flag fault 相关的执行流程

值得注意的是,如果是访问用户空间地址触发的 access flag fault,会通过 `__do_page_fault` 一路调用到 `pte_mkyoung` 将对应页的 AF 位设置为 1; 而访问内核空间地址触发的 access flag fault, 如果无法获取到 `mmap_sem` 信号量会直接调用 `__do_kernel_fault`, 否则, 也会由于内核的虚拟地址并不属于 VMA 管理, 会在 `__do_page_fault` 的 `find_vma` 中出错并返回 `VM_FAULT_BADMAP` 错误, 最后调用也会调用 `__do_kernel_fault`。

### C. Exception trigger 如何制造 access flag fault

Exp\_trigger 目前支持 user space level 3 access flag fault 和 kernel space level 2~3 access flag fault。为了不影响其它代码, 我们通过分别为用户空间和内核空间分配物理地址连续的页或者内存块, 修改其页描述符或块描述符中 AF 位为 1, 并强制访问其中任意一个地址即可触发对应级别的 access flag fault。相关代码位于 `exp_trigger/driver/src/sync_faults/mmu_faults/access_flag_fault.c`。

## Exception trigger 模块说明文档

### user space level 3 access flag fault

代码实现如下：

```
pte = get_tbl_ent(mm, e_data.addr_in, TABLE_ENTRY_PTE);  
set_pte(pte, pte_mkold(*pte));
```

这里的 `e_data.addr_in` 是来自用户空间并属于用户空间物理地址连续的一个页中的一个地址,通过 `get_tbl_ent` 获取改地址所属页的页描述符 `pte`。通过 `pte_mkold` 将 AF 位设置为 0,之后用户空间访问 `e_data.addr_in` 后即可触发 user space level 3 access flag fault。

### kernel space level 2 access flag fault

代码实现如下：

```
p = kmalloc(PTRS_PER_PTE*PAGE_SIZE, GFP_KERNEL);  
pmd = get_tbl_ent(mm, (unsigned long)p, TABLE_ENTRY_PMD);  
set_pmd(pmd, pmd_mkold(*pmd));  
p[0] = 0x12;
```

这里首先调用 `kmalloc` 分配一个物理地址 2M 连续的内存块,然后通过 `get_tbl_ent` 获取该内存块对应的的块描述符 `pmd`,最后调用 `pmd_mkold` 将该块描述符中的 AF 位设置为 0。之后内核空间访问位于该内存块中任意的地址均会触发 kernel space level 2 access flag fault。

### kernel space level 3 access flag fault

代码实现如下：

```
p = vmalloc(PAGE_SIZE);  
pte = get_tbl_ent(mm, (unsigned long)p, TABLE_ENTRY_PTE);  
set_pte(pte, pte_mkold(*pte));  
p[0] = 0x12;
```

这里首先调用 `vmalloc` 分配一个物理地址 4K 连续的页,然后通过 `get_tbl_ent` 获取该内存块对应的的页描述符 `pte`,最后调用 `pte_mkold` 将该页描述符中的 AF 位设置为 0。之后内核空间访问位于该页中任意的地址均会触发 kernel space level 3 access flag fault。

## D. 运行结果

### user space level 3 access flag fault

```
[ 2764.093144@0] exp_trigger: current task: exp_trigger_dem  
[ 2764.094897@0] exp_trigger: addr_in: 0x29c9e000  
[ 2764.099293@0] exp_trigger: try to trigger user space level 3 access flag fault exception  
[ 2764.107319@0] exp_trigger: pte value: 0xe8000005388f53  
[ 2764.112489@0] exp_trigger: new pte value: 0xe8000005388b53  
try to access 0x29c9e000  
[ 2765.117958@0] [exp_trigger] esr: 0x9200004b  
[ 2765.117992@0] [exp_trigger] user level 3 access flag fault @ 0x29c9e000
```

图 14-a. user space level 3 access flag fault 异常现场

## Exception trigger 模块说明文档

可以看到在访问了所对应页描述符 AF 位为 0 的用户空间虚拟地址 0x29c9e000 后触发了 user space level 3 access flag fault。

### kernel space level 2 access flag fault

```
[ 3489.436009@0] exp_trigger: try to trigger user kernel level 2 access flag fault exception
[ 3489.438635@0] exp_trigger: pmd value: 0xe800005c800711
[ 3489.443716@0] exp_trigger: The start address of allocated block: fffffffc05c800000
[ 3489.451122@0] [exp_trigger] kernel level 2 access flag fault
[ 3489.456819@0] Unable to handle kernel paging request at virtual address fffffffc05c800000
[ 3489.464732@0] pgd = fffffffc05ccdb000
[ 3489.468263@0] [fffffffc05c800000] *pgd=0000000000000000, *pud=0000000000000000
```

图 14-b. kernel space level 2 access flag fault 异常现场

可以看到在访问了所对应块描述符 AF 位为 0 的内核空间虚拟地址 0xfffffc0c800000 后触发了 kernel space level 2 access flag fault。

### kernel space level 3 access flag fault

```
[ 4172.034132@1] exp_trigger: try to trigger user kernel level 3 access flag fault exception
[ 4172.036697@1] exp_trigger: pte value: 0xe800005ce27713
[ 4172.041801@1] exp_trigger: The start address of allocated page: ffffff80081d9000
[ 4172.049134@1] [exp_trigger] kernel level 3 access flag fault
[ 4172.054743@1] Unable to handle kernel paging request at virtual address ffffff80081d9000
[ 4172.062749@1] pgd = fffffffc05ce06000
[ 4172.066275@1] [fffff80081d9000] *pgd=000000005cdce003, *pud=000000005cdce003, *pmd=0000000000000000
```

图 14-c. kernel space level 3 access flag fault 异常现场

可以看到在访问了所对应页描述符 AF 位为 0 的内核空间虚拟地址 0xfffff80081d9000 后触发了 kernel space level 2 access flag fault。

## 5.1.1.4 Alignment fault

### A. 什么是 alignment fault

在 ARM Spec<sup>A</sup> 的 B2.4 Alignment support, D1.8.1 PC alignment checking 和 D1.8.2 Stack pointer alignment checking 这些小节中, 有对 alignment fault 的说明。在访问数据, 指令以及堆栈时, 只要没有满足 ARM64 的对齐访问要求, 均会触发 alignment fault。为了便于理解, 特将 ARM64 下 alignment fault 的分类及触发条件整理如下:

## Exception trigger 模块说明文档

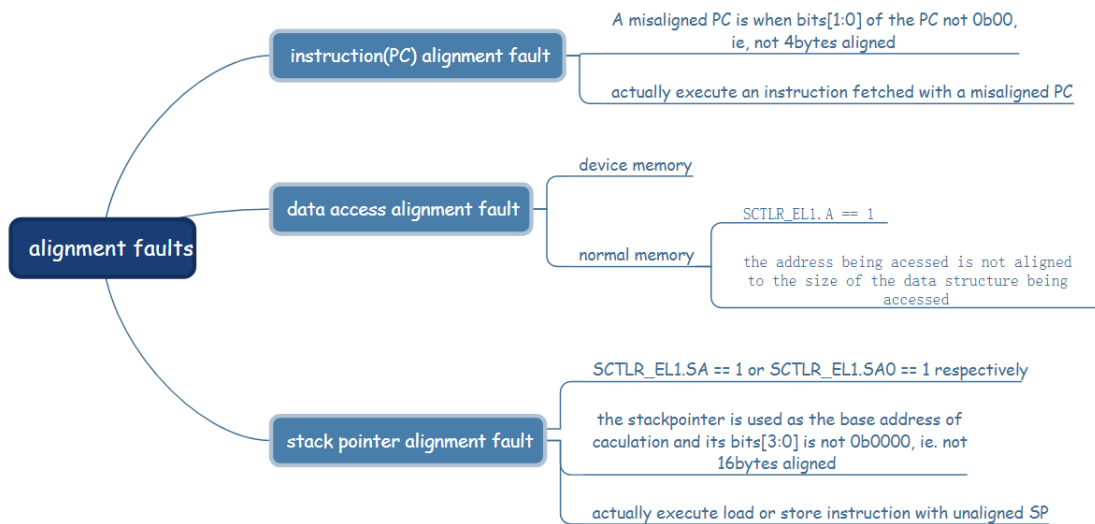


图 15. ARM 64 alignment fault 的分类及触发条件

## B. Linux 中对 alignment fault 的处理

在 Linux 中, data alignment fault 和 pc/sp alignment fault 的处理函数不同, 前者是 `do_alignment_fault`, 后两者是 `do_sp_pc_abort`。这三类 alignmet fault 在 Linux ARM64 下均属于不可处理异常, 会触发 panic 或者 oops。

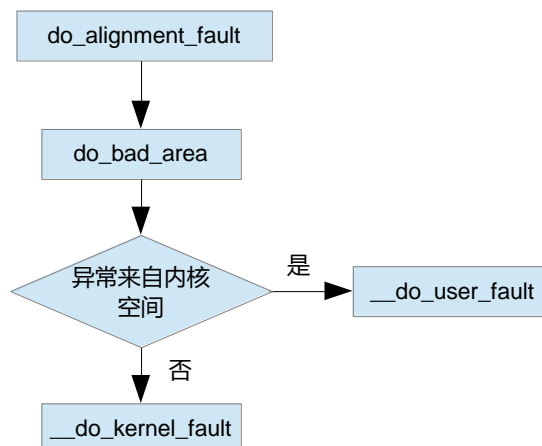


图 16-a. do\_alignment\_fault 代码执行流程

`do_alignment_fault` 函数会直接调用 `do_bad_area`, 该函数会判断异常来自用户空间还是内核空间, 如果是前者则调用 `__do_user_fault`, 否则调用 `__do_kernel_fault`。

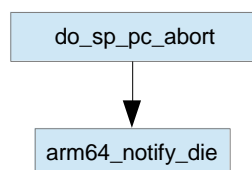


图 16-b. do\_sp\_pc\_abort 代码执行流程

## Exception trigger 模块说明文档

在不考虑防范 aliasing attack 防护的情况下(CONFIG\_HARDEN\_BRANCH\_PREDICTOR 本身也没有使能), do\_sp\_pc\_abort 会直接调用 arm64\_notify\_die。

### C. Exception trigger 如何制造 alignment fault

Exp\_trigger 目前支持触发 kernel space data/pc/sp alignment fault。所有非对齐 data sp 和 pc 的地址均通过对齐的地址加上一定的偏移获得, 然后访问这些地址就可触发对应的 alignment fault 异常。相关代码位于 exp\_trigger/driver/src/sync\_faults/mm\_faults/sp\_alignment\_fault.S 以及 exp\_trigger/driver/src/sync\_faults/mm\_faults/alignment\_fault.c。

#### kernel space data alignment fault

代码实现如下:

```
d_mem = host->clksrc_base;
unalign_d_mem = (unsigned char __iomem *)d_mem + 1;
readl(unalign_d_mem);
```

这里 d\_mem 执行一段被 ioremap 过的内存, 而 ioremap 后的内存都是 device 类型的内存。而 unalign\_d\_mem 则在 d\_mem 的基础上加上偏移 1, 迫使其 1 字节对齐。最后通过 readl 尝试从 unalign\_d\_mem 指向的内存空间读取 4 字节的数据, 即可触发 data alignment fault。

#### kernel space pc alignment fault

代码实现如下:

```
void *test_addr = test_func;
unaligned_pc = (unsigned char *)test_addr + 0x10 + 0x1;
asm("blr %0:::r"(unaligned_pc));
```

这里的 test\_func 指向一段可以正常执行的代码, 而 unaligned\_pc 则在 test\_func 的基础上加上一定的偏移, 使其不 word aligned。最后通过 blr 指令直接跳转到 unaligned\_pc 执行, 即可触发 pc alignment fault。

#### kernel space sp alignment fault

代码实现如下:

```
mov    x10, sp
add    x10, x10, #0x1
mov    sp, x10
ldr    x11, [sp]
```

这里首先获取对齐且正常的 sp 地址, 接着在该 sp 基础上加 1 使其不对齐, 最后通过 ldr 指令以该不对齐的 sp 为 base 取数据, 从而触发 sp alignment fault。



## D. 运行结果

## kernel space data alignment fault

```
[ 77.331159@3] exp_trigger: sctlr_el1: 0x34d5d91d
[ 77.331194@3] exp_trigger: data access alignment checking for normal memory is disabled
[ 77.338088@3] exp_trigger: SP alignment checking for EL1 is enabled
[ 77.344185@3] exp_trigger: SP alignment checking for EL0 is enabled
[ 77.350322@3] exp_trigger: try to trigger kernel space Data Access alignment fault exception
[ 77.358684@3] exp_trigger: d_mem: fffffff800853e000
[ 77.363412@3] exp_trigger: unalign_d_mem: fffffff800853e001
[ 77.368853@3] exp_trigger: kernel alignment fault
[ 77.373503@3] Unable to handle kernel paging request at virtual address fffffff800853e001
[ 77.381528@3] pgd = fffffffc05ccfe000
[ 77.385059@3] [ffffff800853e001] *pgd=000000005ccf6003, *pud=000000005ccf6003, *pmd=0000000000000000
```

图 16-a. kernel space data alignment fault

可以看到在以非 word aligned 的方式访问且指向 device memory 的内核空间虚拟地址 0xffffffff800853e001 后，触发了 kernel space data alignment fault。

## kernel space pc alignment fault

```
[ 360.384171@1] exp_trigger: sctlr_el1: 0x34d5d91d
[ 360.384204@1] exp_trigger: data access alignment checking for normal memory is disabled
[ 360.391111@1] exp_trigger: SP alignment checking for EL1 is enabled
[ 360.397242@1] exp_trigger: SP alignment checking for EL0 is enabled
[ 360.403350@1] exp_trigger: try to trigger kernel space PC alignment fault exception
[ 360.410930@1] exp_trigger: test_addr: fffffff8009a1c2e0
[ 360.416020@1] exp_trigger: unalign_pc: fffffff8009a1c2f1
[ 360.421193@1] exp_trigger_dem[2956]: PC Alignment exception: pc=ffffff8009a1c2f1 sp=ffffff800a850b88
[ 360.430218@1] Internal error: Oops - SP/PC alignment exception: 8a000000 [#2] PREEMPT SMP
```

图 16-b. kernel space pc alignment fault

可以看到在尝试跳转到非 word aligned 的内核空间虚拟地址 0xffffffff800a850b88 执行后，触发了 kernel space pc alignment fault。

## kernel space sp alignment fault

```
[ 567.990485@2] exp_trigger: sctlr_el1: 0x34d5d91d
[ 567.990519@2] exp_trigger: data access alignment checking for normal memory is disabled
[ 567.997400@2] exp_trigger: SP alignment checking for EL1 is enabled
[ 568.003884@2] exp_trigger: SP alignment checking for EL0 is enabled
[ 568.009624@2] exp_trigger: try to trigger kernel space SP alignment fault exception
[ 568.017242@2] exp_trigger: !!! -----attention----- !!!
[ 568.022655@2] exp_trigger: when SP alignment happens, if CONFIG_AMLOGIC_VMAP is not
[ 568.030404@2] exp_trigger: enabled, the exception entry still uses the original stack,
[ 568.038259@2] exp_trigger: Which means the unaligned sp is still used as the base sp
[ 568.045929@2] exp_trigger: for calculation, causing iteration of SP alignment faults.
[ 568.053695@2] exp_trigger: And the OS will never have the chance to execute do_sp_pc_abort,
[ 568.061968@2] exp_trigger: which means we cannot see any meaningful logs shows up at terminal.
[ 568.070510@2] exp_trigger: If you really wanna to verify do_kernel_SP_alignment_fault, please use ICE :-))
[ 568.080039@2] exp_trigger: !!! -----attention----- !!!
```

图 16-c. kernel space sp alignment fault

可以看到此时虽然触发了异常，但无任何异常信息，尤其没有看到 SP/PC alignment

## Exception trigger 模块说明文档

exception 的字眼。这里其实已经触发了 `sp alignment fault`，之所没有相关打印，是因为在 `sp alignment fault` 的异常处理程序的入口处，继续以非对齐的 `sp` 作为 `base` 计算，导致递归的 `sp alignment fault`。这里为了验证确实触发了该异常，可以连接 ICE 确认<sup>[附录 2]</sup>。

### 5.1.1.5 Permission fault

#### A. 什么是 permission fault

在 ARM Spec<sup>A</sup> 的 D4.5.1 中有对 `permission fault` 的详细定义。简单来说在进行数据访问和代码执行时，如果违反了权限设置就会触发对应级别的 `permission fault`。其中数据访问的权限通过页描述符或块描述符中的 `AP`(Access Permission)域控制，代码执行的权限通过页描述符或块描述符中的 `UXN`(Unprivilege eXecute Never)/`XN`(eXecute Never)位，`PXN`(Privilege Execute Never)位以及 `SCTLR_ELx` 中的 `WXN`(Writable eXecute Never)位控制。

这里 `AP` 域位于页描述符或块描述符的 `bit[7:6]`，这里的 `AP[2]`用于控制 `read only` 还是 `read/write`；`AP[1]`用于设置对来自 `EL1` 还是 `EL0` 的数据访问的控制。这里的 `UXN/XN`(两者是同一个 bit)和 `PXN` 分别是页描述符或块描述符的 `bit[54]`和 `bit[53]`，分别用于控制 `EL0` 和 `EL1` 下是否有某段代码的执行权限，在 `EL2` 或 `EL3` 下则由 `XN` 控制。另外 `SCTLR_ELx.WXN` 则用于控制在某段内存有可写权限时就不能执行，且可覆盖 `UXN/XN` 和 `PXN` 的值。`AP` 域，`UXN/XN`，`PXN` 以及 `SCTLR_ELx.WXN` 的各种组合值对读写及可执行权限的影响在 ARM Spec<sup>A</sup> 的 Table D4-33 中有详细描述。

#### B. Linux 中对 permission fault 的处理

用户空间或内核空间对用户空间地址的访问，以及用户空间或内核空间对内核空间地址的访问，都可能触发相应的 `permission fault`。Linux 中对 `permission fault` 的处理函数同样是 `do_page_fault`。其执行流程图下图所示，同样只列出与 `permission fault` 相关部分。

## Exception trigger 模块说明文档

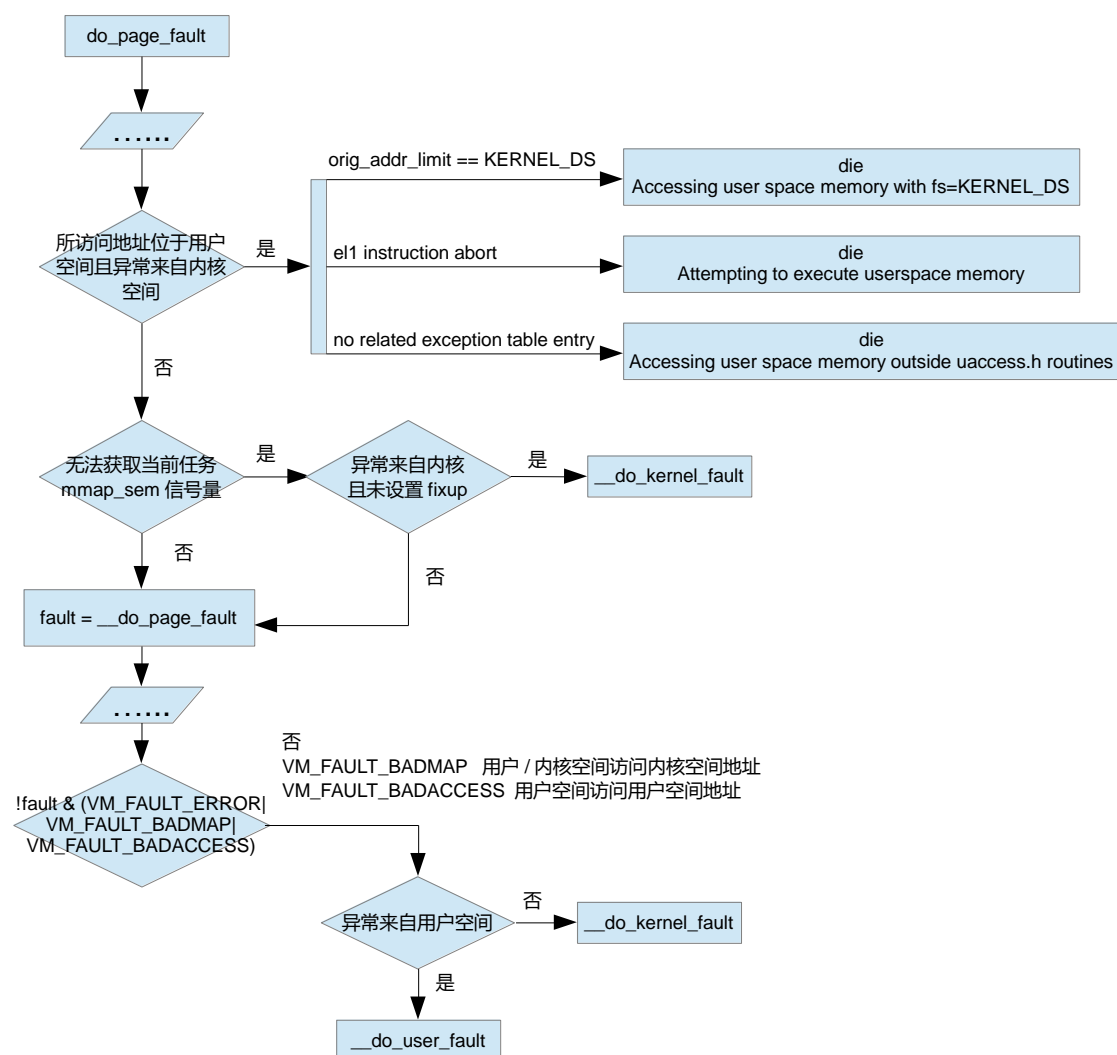


图 17. `do_page_fault` 中与 permission fault 相关的执行流程

对于内核空间访问用户地址触发的 `permission fault`，会直接调用 `die`，并打印出详细的原因。对于其它情况，会进入 `__do_page_fault` 中执行。如果是访问内核空间地址触发的 `permission fault`，则会因为地址不在 VMA 中，`__do_page_fault` 会返回 `VM_FAULT_BADMAP`；如果是访问用户空间地址触发的 `permission fault`，则会因为所执行的操作，不在对应 VMA 的权限属性允许范围内，`__do_page_fault` 返回 `VM_FAULT_BADACCESS`。之后，根据异常来自用户空间还是内核空间，会分别调用 `__do_user_fault` 和 `__do_kernel_fault`。

### C. Exception trigger 如何制造 permission fault

目前 `exp_trigger` 支持触发 `kernel space level 3 data accessing permission fault` 和 `kernel space level 2~3 instruction fetching permission fault` 3 类 `permission fault` 异常。通过控制页描述符或块描述符中的 `AP`，`PXN` 位，构造只读或者没有执行权限的内存区域，在尝试写这类区域或者执行这类区域时，就可以触发对应的异常了。相关的代码位于 `exp_trigger/driver/src/sync_faults/mmu_faults/permission_fault.c`。

## Exception trigger 模块说明文档

### Kernel space level 3 data accessing permission fault

代码实现如下:

```
p = vmalloc(PAGE_SIZE);
pte = get_tbl_ent(mm, (unsigned long)p, TABLE_ENTRY_PTE);
new_pte.pte = pte_val(*pte) | PTE_RDONLY;
set_pte(pte, new_pte);
p[0] = 0x12;
```

这里首先通过 `vmalloc` 分配一个物理地址连续的页, 默认情况下对应页描述符的 AP 域为 00, 表示内核对该页有可读可写的权限。然后 AP 域设置为 10, 表示内核对该页只有读的权限。最后尝试向该页写数据, 即可触发 kernel space level 3 data accessing permission fault。

### Kernel space level 2 instruction fetching permission fault

代码实现如下:

```
p = kmalloc(PTRS_PER_PTE*PAGE_SIZE, GFP_KERNEL);
pmd = get_tbl_ent(mm, (unsigned long)p, TABLE_ENTRY_PMD);
memcpy(p, test_func, sizeof(test_func));
test_val = ((int (*)(void))p)();
```

这里首先通过 `kmalloc` 分配一个物理地址 2M 连续的内存块, 然后通过 `memcpy` 将一段合理的代码拷贝到该内存块中。默认情况下该内存块没有执行权限, 因此尝试执行该数据后, 就会触发 kernel space level 2 instruction fetching permission fault。

### Kernel space level 3 instruction fetching permission fault

代码实现如下:

```
p = vmalloc_exec(PAGE_SIZE);
pte = get_tbl_ent(mm, (unsigned long)p, TABLE_ENTRY_PTE);
memcpy(p, test_func, sizeof(test_func));
new_pte.pte = pte_val(*pte) | PTE_PXN;
set_pte(pte, new_pte);
test_val = ((int (*)(void))p)();
```

这里首先通过 `vmalloc_exec` 分配一个带执行权限的物理地址连续的页(其实可以直接通过 `vmalloc` 分配一个不带执行权限的页, 这里之所以使用 `vmalloc_exec`, 只是想用用这个没有用过的 API), 然后通过 `memcpy` 将一段合理的代码拷贝到该内存块中。默认情况下, 这个页对于 `privilege` 是有可执行权限的, 因此首先设置 PXN 使 `privilege` 没有可执行权限。最后尝试执行该数据, 即可触发 kernel space level 3 instruction fetching permission fault。

## D. 运行结果

## Kernel space level 3 data accessing permission fault

```
[ 372.760264@3] exp_trigger: sctlr_ell: 0x34d5d91d
[ 372.760301@3] exp_trigger: triggering source is data accessing
[ 372.765125@3] exp_trigger: try to trigger kernel space level 3 data access permission fault exception
[ 372.774227@3] exp_trigger: pte value: 0xe800005ccc5713
[ 372.779257@3] exp_trigger: new pte value: 0xe800005ccc5793
[ 372.784682@3] exp_trigger: The start address of allocated page: fffffff800907d000
[ 372.792007@3] [exp_trigger] kernel level 3 permission fault
[ 372.797524@3] Unable to handle kernel paging request at virtual address fffffff800907d000
[ 372.805542@3] pgd = fffffffc05ccab000
[ 372.809075@3] [ffffff800907d000] *pgd=000000005d762003, *pud=000000005d762003, *pmd=0000000000000000
```

图 18-a. kernel space level 3 data accessing permission fault

可以看到这里在尝试访问由内核空间虚拟地址 0xfffff800907d000 指向的没有权限访问内存空间后，触发了 kernel space level 3 data accessing permission fault 异常。

## Kernel space level 2 instruction fetching permission fault

```
[ 197.591556@3] exp_trigger: sctlr_ell: 0x34d5d91d
[ 197.591593@3] exp_trigger: triggering source is instruction fetching
[ 197.597211@3] exp_trigger: try to trigger kernel space level 2 instruction fetch permission fault exception
[ 197.606678@3] exp_trigger: pmd value: 0xe800005ce00711
[ 197.612057@3] exp_trigger: execute memory block starting at: fffffffc05ce00000
[ 197.618675@3] [exp_trigger] kernel level 2 permission fault
[ 197.624170@3] Unable to handle kernel paging request at virtual address fffffffc05ce00000
[ 197.632175@3] pgd = fffffffc05d7bc000
[ 197.635706@3] [ffffffc05ce00000] *pgd=0000000000000000, *pud=0000000000000000
```

图 18-a. kernel space level 2 instruction fetching permission fault

可以看到在尝试执行 0xfffffc05ce00000 指向的没有执行权限的代码后，触发了 kernel space level 2 instruction fetching permission fault。

## Kernel space level 3 instruction fetching permission fault

```
[ 35.262206@0] exp_trigger: sctlr_ell: 0x34d5d91d
[ 35.262244@0] exp_trigger: triggering source is instruction fetching
[ 35.267858@0] exp_trigger: try to trigger kernel space level 3 instruction fetch permission fault exception
[ 35.277368@0] exp_trigger: pte value: 0xc800005cd02713
[ 35.282361@0] exp_trigger: new pte value: 0xe800005cd02713
[ 35.287657@0] exp_trigger: execute executable page starting at: fffffff8008b9a000
[ 35.294973@0] [exp_trigger] kernel level 3 permission fault
[ 35.300618@0] Unable to handle kernel paging request at virtual address fffffff8008b9a000
[ 35.308526@0] pgd = fffffffc05d78f000
[ 35.312060@0] [ffffff8008b9a000] *pgd=000000005cd1c003, *pud=000000005cd1c003, *pmd=0000000000000000
```

图 18-c. kernel space level 3 instruction fetching permission fault

可以看到在尝试执行 0xfffff8008b9a000 指向的没有执行权限的代码后，触发了 kernel space level 3 instruction fetching permission fault。

### 5.1.1.6 synchronous external abort on translation table walk

#### A. 什么是 synchronous external abort on translation table walk

在 ARM Spec<sup>A</sup> 的 D4.5.1 中, 有对 synchronous external abort on translation table walk 的准确定义, 简单来说就是在查表过程中发生了同步类型的外部异常。

#### B. Linux 中对 synchronous external abort on translation table walk 的处理

与 access flag fault 一样, synchronous external abort on translation table walk 在 Linux 下也是不可处理的异常。其异常的处理入口为 do\_bad, 相关执行流程已在 access flag fault 中有详述。

#### C. Exp\_trigger 如何制造 synchronous external abort on translation table walk

修改某一虚拟地址对应的 PMD 表项的内容, 使其 Next-level table address 域的内容为指向某个 secure register 或者 secure memory 的地址。这样修改后, 如果尝试去访问该地址, 那么在查表过程中, 就会迫使 CPU 去访问 secure register 或者 secure memory。由于 kernel 下没有访问 secure register 与 secure memory 的权限, 因此会触发异常。同时由于是直接访问的物理地址(不会因为 cache 问题引起操作的异步), 而且 secure register 和 secure memory 相对于 MMU 都是外部的, 因此是 synchronous external abort 的异常。最后, 因为是在查表过程中触发的该异常, 最终触发的是 synchronous external abort on translation table walk 类的异常。

相关代码位于 exp\_trigger/driver/src/sync\_faults/mm\_faults/sync\_ext\_ab\_on\_tbl\_walk.c, 目前只实现了 kernel space 下 synchronous external abort on translation table walk 类型异常。其核心代码如下所示:

```
#define SEC_AO_RTI_STATUS_REG0 (0xda100000 + (0x00 << 2))

addr = (unsigned long)(meson_reg_map[IO_VAPB_BUS_BASE] + 0x470);
pmd = get_ttbl_ent(mm, addr, TABLE_ENTRY_PMD);
new_pmd.pmd = (pmd_val(*pmd) & 0xFFFF000000000000) | SEC_AO_RTI_STATUS_REG0;
set_pmd(pmd, new_pmd);
val = readl((void __iomem *)addr);
```

这里 SEC\_AO\_RTI\_STATUS\_REG0 是 GXL 系列 CPU 下的一个 secure register。我们选定访问的地址是一个普通的 VCBUS 总线地址, 在替换该地址对应 PMD 的 Next-level table address 域为 SEC\_AO\_RTI\_STATUS\_REG0 后, 尝试用 readl 去访问该地址, 即可触发对应异常。

## D. 运行结果

```
[ 576.643875@0] try to trigger synchronous external abort on translation table walk exception
[ 576.663157@0] CPU1: shutdown
[ 576.663194@0] psci: CPU1 killed.
[ 576.706924@0] CPU2: shutdown
[ 576.706971@0] psci: CPU2 killed.
[ 576.754855@0] CPU3: shutdown
[ 576.754904@0] psci: CPU3 killed.
[ 576.775526@0] pmd value: 0x5dc8a003
[ 576.775555@0] new pmd value: 0xda100003
[ 576.777188@0] try to read data from addr: 0xffffffff8008600470
[ 576.782798@0] Unhandled fault: synchronous abort (translation table walk) (0x96000217) at 0xffffffff8008600470
[ 576.792542@0] Internal error: : 96000217 [#6] PREEMPT SMP
```

图 19. synchronous external abort on translation table walk 异常现场

可以看到在尝试访问地址对应页表描述符地址域指向 secure register 的虚拟地址 0xffffffff8008600470 后，触发了 synchronous external abort on translation walk。

## 5.1.2 Synchronous external abort

### 5.1.2.1 什么是 synchronous external abort

外部访问触发的同步类型异常就属于 synchronous external abort。这里的外部是相对于 MMU 来说的，secure memory 就是典型的外部区域。

### 5.1.2.2 Linux 下如何处理 synchronous external abort

对于 Linux 来说，synchronous external abort 属于不可处理异常，对应的处理函数为 do\_bad。该函数的具体执行流程已经在 5.1.1.1 节中描述。

### 5.1.2.3 Exp\_trigger 如何触发 synchronous external abort

按照 5.1.2.1 中对定义，需要满足三个条件才能触发 synchronous external abort：外部，同步以及异常访问。这里我们选择经过映射的 secure memory(内核无权访问 secure memory) 作为被访问的区域，由于这段区域之前从未被访问过，访问操作会直接尝试从 secure memory 中获取数据，因此这样的访问是同步的外部异常访问。相关代码位于 exp\_trigger/driver/src/sync\_faults/mm\_faults/others/sync\_ext\_abort.c。

代码实现如下：

```
secmem_address = (void __iomem *) (secmon_start_virt +
                                   ((secmon_end_virt - secmon_start_virt)/3)*2);
readl(secmem_address);
```

这里的 secmon\_start\_virt 和 secmon\_end\_virt 分别指向一段已经被映射为 cacheable 的 secure memory 的起始和结束地址。secmem\_address 基于这两个地址计算了一个中间值。最后调用 readl 尝试读取该地址，从而触发 synchronous external abort。



### 5.1.2.4 运行结果

```
[ 34.873348@3] exp_trigger: try to trigger synchronous external abort exception
[ 34.874990@3] exp_trigger: try to read secure memory address: fffffffc00520000
[ 34.882258@3] Unhandled fault: synchronous external abort (0x96000010) at 0xffffffffc00520000
[ 34.890629@3] Internal error: : 96000010 [#1] PREEMPT SMP
```

图 20. Synchronous external abort 异常现场

可以看到尝试读取由虚拟地址 0xffffffffc00520000 指向的 secure memory 区域后，触发了 synchronous external abort

## 5.2 SError

### 5.2.1 什么是 SError

在文档<sup>[2]</sup>中有对 SError(System Error)的定义，简单来说就是除 FIQ 和 IRQ 外的异步外部 abort 就属于 SError。SError 的特点是异常触发时执行的指令并非引起异常的指令，特别的文档<sup>[2]</sup>中给出了一个常见的 SError 的例子：由于 cache line write-back 触发的异常。

### 5.2.2 Linux 下如何处理 SError

在 Linux 下 SError 属于不可处理异常，其异常处理函数为 bad\_mode，该函数在打印一些辅助信息后直接 panic，其具体执行流程如下：

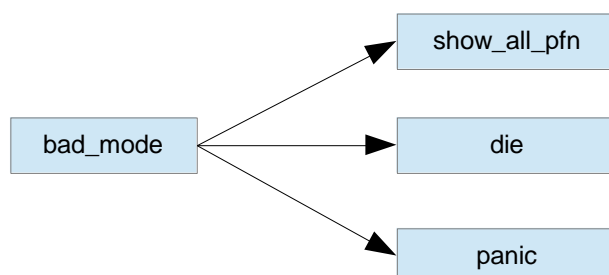


图 21. bad\_mode 代码执行流程

### 5.2.3 Exp\_trigger 如何触发 SError

按照 5.2.1 中对 SError 的定义，需要满足三个条件才能触发 SError：外部，异步以及异常访问。这里的外部是相对于 MMU 的，在我们的平台中 secure memory 和外设寄存器均属于外部的。因此直接写 cacheable secure memory 就可以触发 SError。相关代码位于 exp\_trigger/driver/src/serror/serror.c。

代码实现如下：



## Exception trigger 模块说明文档

```
secmem_address = (void __iomem *) (secmon_start_virt +  
    ((secmon_end_virt - secmon_start_virt) * 2) / 3);  
writel(value, secmem_address);
```

这里的 `secmon_start_virt` 和 `secmon_end_virt` 分别指向一段已经被映射为 `cacheable` 的 `secure memory` 的起始和结束地址。`secmem_address` 基于这两个地址计算了一个中间值。最后调用 `writel` 尝试写该地址，从而触发 `SError`。

### 5.2.4 运行结果

```
[ 15.678246@3] exp_trigger: try to trigger error exception  
[ 15.678286@3] exp_trigger: try to write 0x99 to secure memory address: fffffffc00520000  
[ 15.686472@3] Bad mode in Error handler detected on CPU3, code 0xbf000000 -- SError
```

图 22. `SError` 异常现场

可以看到尝试向虚拟地址 `0xfffffc0052000000` 指向的 `secure memory` 区域写 `0x99` 后，触发了 `SError`。

## 参考文档

A. <<ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile>>

B. <<AArch64 Exception and Interrupt Handling Version 1.0>>

## 附录

### 1. Fixup exception 的例子

在 Linux 中有些异常是可以被当前代码安全处理的，不需要交由异常处理函数处理。为了满足这个需求，Linux 中提供了 `exception fixup` 机制。在发生异常时，`exception fixup` 会尝试遍历 `exception table`，如果其中正好有一项的指令地址和当前触发异常的指令地址一致，就会退出异常处理程序并跳转到指定的地址执行。`Exception table` 表项的结构非常简单，由两个成员变量构成，如下所示：

```
struct exception_table_entry  
{  
    int insn, fixup;  
};
```

这里 `insn` 和 `fixup` 分别用 `offset` 的方式表示触发异常的指令地址以及对应的跳转地址。因此只需要在 `exception table` 中添加对应的 `exception table entry` 即可绕过异常处理函数，到指定的地址执行。

`futex(fast userspace mutex)` 中，函数 `futex_detect_cmpxchg` 中检测是否实现了函数 `futex_atomic_cmpxchg_inatomic`，就用到了 `exception fixup`。相关代码如下：

## Exception trigger 模块说明文档

```
static void __init futex_detect_cmpxchg(void)
{
    if (cmpxchg_futex_value_locked(&curval, NULL, 0, 0) == -EFAULT)
        futex_cmpxchg_enabled = 1;
}
```

这里首先通过检查 `cmpxchg_futex_value_locked` 的返回值是否为 `-EFAULT` 来判断是否实现了函数 `futex_atomic_cmpxchg_inatomic`。特别需要注意的是这里传入的第二个参数是 `NULL`。

```
static int cmpxchg_futex_value_locked(u32 *curval, u32 __user *uaddr,
                                      u32 uval, u32 newval)
{
    int ret;
    pagefault_disable();
    ret = futex_atomic_cmpxchg_inatomic(curval, uaddr, uval, newval);
    pagefault_enable();
    return ret;
}
```

然后直接调用了函数 `cmpxchg_futex_cmpxchg_inatomic`。

```
static inline int futex_atomic_cmpxchg_inatomic(u32 *uval, u32 __user *uaddr,
                                                u32 oldval, u32 newval)
{
    int ret = 0;
    u32 val, tmp;
    u32 __user *uaddr;

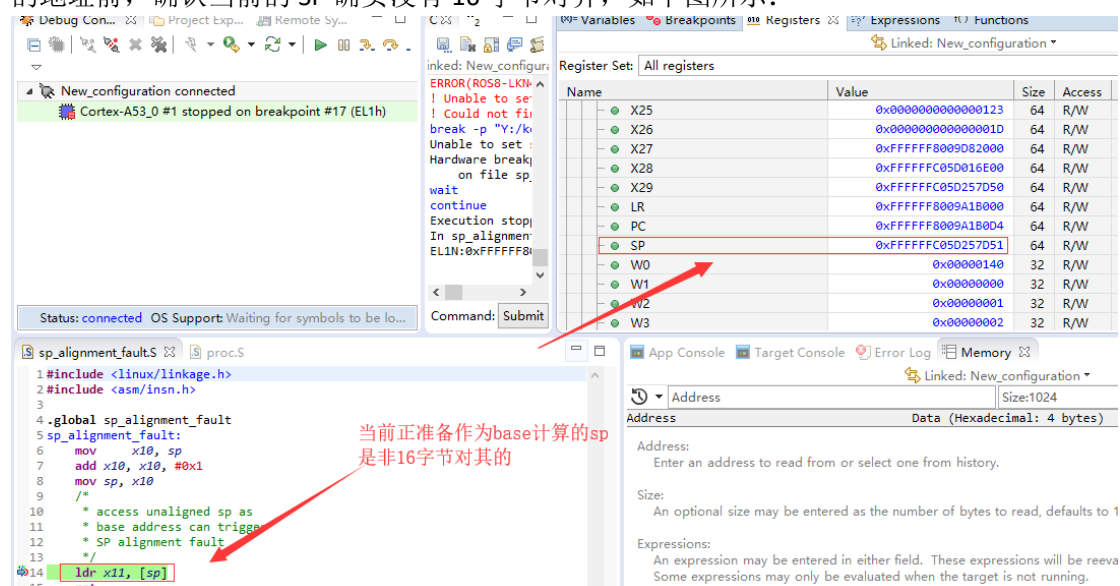
    asm volatile("// futex_atomic_cmpxchg_inatomic\n"
        " prfmpstl1strm, %2\n"
        "1: ldxr %w1, %2\n"
        " sub %w3, %w1, %w4\n"
        " cbnz %w3, 3f\n"
        "2: stlxx %w3, %w5, %2\n"
        " cbnz %w3, 1b\n"
        " dmb ish\n"
        "3:\n"
        " .pushsection .fixup, \"ax\"\n"
        "4: mov %w0, %w6\n"
        " b 3b\n"
        " .popsection\n"
        _ASM_EXTABLE(1b, 4b)
        _ASM_EXTABLE(2b, 4b)
        : "+r" (ret), "=&r" (val), "+Q" (*uaddr), "=&r" (tmp)
        : "r" (oldval), "r" (newval), "Ir" (-EFAULT)
        : "memory");
    return ret;
}
```

## Exception trigger 模块说明文档

最后在函数 `futex_atomic_cmpxchg_inatomic` 中会通过 `_ASM_EXTABLE(1b, 4b)` 向 `exception table` 中添加一个 `exception table entry`，这个 `exception table entry` 的 `insn` 对应标签 1 处地址，`fixup` 对应标签 4 处地址。由于所传入的 `_uaddr` 为 `NULL`，在执行到标签 1 处尝试解引用 `_uaddr` 时，会触发 `translation fault`。由于存在对应的 `exception table entry`，会直接通过 `exception fixup` 跳转到标签 4 处执行。在标签 4 处将 `-EFAULT` 赋值给 `ret`，并将 `ret` 作为该函数的返回值。因此最开始的 `futex_detect_cmpxchg` 中会因为检测到返回值为 `-EFAULT`，判断出内核已经实现了函数 `futex_atomic_cmpxchg_inatomic`。

## 2.ICE 查看 Sp alignment fault 的现场

接上 ICE(ICE 的使用不在本文范围内)后,首先在尝试访问以非 16 字节对齐的 SP 为 base 的地址前,确认当前的 SP 确实没有 16 字节对齐,如下图所示:



单步执行后，通过 ICE 可以确认当前已经进入了异常处理程序，且当前的异常确实是 `sp alignment fault` 异常，如下图所示：

