kernel 3.18.7

in arch/arm/kernel/entry-armv.S

该文件中包括的是ARM core产生excption后的entries.即无论是interrupt还是page fault等等，当发生了，软件的入口就都在该文件中。

ARM instruction相比intel instruction更难读，而且gnu assembler的汇编格式相比Microsoft汇编格式也更难读(最起码有点i386汇编阅读史的我是这么认为的)。要看懂

entry-arm.S中的汇编还不如看vmlinux的反汇编码更容易。

```
$ arm-linux-gnueabi-objdump -d vmlinux-3.18.7-yocto-standard > linux-3.18.7.src
```

Disassembly of section .vectors:

下面就是entry-armv.S中的汇编源码被编译成code以后的反汇编code，我看着比源码要容易理解多了。那些gnu assembler的伪码看着就让人头疼。我以前写了2年的Microsoft

格式的x86汇编，从没有对汇编产生过太大的讨厌，但我每当看到gnu assembler格式的汇编就有种厌恶感，再加上是ARM instuction，更是一看就有想吐的感觉。 或许是先入为

主的缘故，但一直不理解gnu那帮人为什么设计出这种format。

==

```
Disassembly of section .vectors:

00000000 <__vectors_start>:
   0:  ea0003ff  b      1004 <vector_rst>
   4:  ea000465        b      11a0 <vector_und>

   8:  e59ffff0    ldr    pc, [pc, #4080]        ; 1000 <__stubs_start>

   c:  ea000443        b      1120 <vector_pabt>

  10: ea000422        b      10a0 <vector_dabt>

  14: ea000481        b      1220 <vector_addrexcptn>

  18: ea000400        b      1020 <vector_irq>

  1c: ea000487        b      1240 <vector_fiq_offset>
```

```
Disassembly of section .stubs:


00001000 <__stubs_start>:

    1000:   c000e760        .word       0xc000e760


00001004 <vector_rst>:

    1004:   ef9f0000  svc  0x009f0000

    1008:   ea000064        b      11a0 <vector_und>

    100c:   e320f000        nop  {0}

    1010:   e320f000        nop  {0}

    1014:   e320f000        nop  {0}

    1018:   e320f000        nop  {0}

    101c:   e320f000        nop  {0}


00001020 <vector_irq>:
```

```
1020:   e24ee004        sub  lr, lr, #4

1024:   e88d4001        stm  sp, {r0, lr}

1028:   e14fe000        mrs  lr, SPSR

102c:   e58de008        str   lr, [sp, #8]

1030:   e10f0000        mrs  r0, CPSR

1034:   e2200001        eor  r0, r0, #1

1038:   e16ff000  msr SPSR_fsxc, r0

103c:   e20ee00f        and  lr, lr, #15

1040:   e1a0000d        mov r0, sp

1044:   e79fe10e        ldr   lr, [pc, lr, lsl #2]

1048:   e1b0f00e        movs       pc, lr

104c:   c00122c0        .word       0xc00122c0

1050:   c0011f60        .word       0xc0011f60

1054:   c0011f60        .word       0xc0011f60

1058:   c0012000        .word       0xc0012000

105c:   c0011f60        .word       0xc0011f60

1060:   c0011f60        .word       0xc0011f60

1064:   c0011f60        .word       0xc0011f60

1068:   c0011f60        .word       0xc0011f60

106c:   c0011f60        .word       0xc0011f60

1070:   c0011f60        .word       0xc0011f60

1074:   c0011f60        .word       0xc0011f60

1078:   c0011f60        .word       0xc0011f60

107c:   c0011f60        .word       0xc0011f60

1080:   c0011f60        .word       0xc0011f60
```

```
    1084:   c0011f60        .word       0xc0011f60

    1088:   c0011f60        .word       0xc0011f60

    108c:   e320f000        nop  {0}

    1090:   e320f000        nop  {0}

    1094:   e320f000        nop  {0}

    1098:   e320f000        nop  {0}

    109c:   e320f000        nop  {0}


000010a0 <vector_dabt>:

    10a0:   e24ee008        sub  lr, lr, #8

    10a4:   e88d4001        stm  sp, {r0, lr}

    10a8:   e14fe000        mrs  lr, SPSR

    10ac:   e58de008        str   lr, [sp, #8]

    10b0:   e10f0000        mrs  r0, CPSR

    10b4:   e2200004        eor   r0, r0, #4

    10b8:   e16ff000  msr SPSR_fsxc, r0

    10bc:   e20ee00f        and  lr, lr, #15

    10c0:   e1a0000d        mov r0, sp

    10c4:   e79fe10e        ldr   lr, [pc, lr, lsl #2]

    10c8:   e1b0f00e        movs        pc, lr

    10cc:   c0012280        .word       0xc0012280

    10d0:   c0011f50        .word       0xc0011f50

    10d4:   c0011f50        .word       0xc0011f50

    10d8:   c0011fa0        .word       0xc0011fa0

    10dc:   c0011f50        .word       0xc0011f50
    10e0:   c0011f50        .word      0xc0011f50

    10e4:   c0011f50        .word      0xc0011f50

    10e8:   c0011f50        .word      0xc0011f50

    10ec:   c0011f50        .word      0xc0011f50

    10f0:   c0011f50        .word      0xc0011f50

    10f4:   c0011f50        .word      0xc0011f50

    10f8:   c0011f50        .word      0xc0011f50

    10fc:   c0011f50        .word      0xc0011f50

    1100:   c0011f50        .word      0xc0011f50

    1104:   c0011f50        .word      0xc0011f50

    1108:   c0011f50        .word      0xc0011f50

    110c:   e320f000        nop  {0}

    1110:   e320f000        nop  {0}

    1114:   e320f000        nop  {0}

    1118:   e320f000        nop  {0}

    111c:   e320f000        nop  {0}


00001120 <vector_pabt>:

    1120:   e24ee004        sub  lr, lr, #4

    1124:   e88d4001        stm  sp, {r0, lr}

    1128:   e14fe000        mrs  lr, SPSR

    112c:   e58de008        str   lr, [sp, #8]

    1130:   e10f0000        mrs  r0, CPSR

    1134:   e2200004        eor   r0, r0, #4

    1138:   e16ff000  msr  SPSR_fsxc, r0
```

```
113c:  e20ee00f     and  lr, lr, #15
1140:  e1a0000d     mov r0, sp
1144:  e79fe10e     ldr    lr, [pc, lr, lsl #2]
1148:  e1b0f00e     movs       pc, lr
114c:  c00124e0     .word      0xc00124e0
1150:  c0011f40     .word      0xc0011f40
1154:  c0011f40     .word      0xc0011f40
1158:  c0012120     .word      0xc0012120
115c:  c0011f40     .word      0xc0011f40
1160:  c0011f40     .word      0xc0011f40
1164:  c0011f40     .word      0xc0011f40
1168:  c0011f40     .word      0xc0011f40
116c:  c0011f40     .word      0xc0011f40
1170:  c0011f40     .word      0xc0011f40
1174:  c0011f40     .word      0xc0011f40
1178:  c0011f40     .word      0xc0011f40
117c:  c0011f40     .word      0xc0011f40
1180:  c0011f40     .word      0xc0011f40
1184:  c0011f40     .word      0xc0011f40
1188:  c0011f40     .word      0xc0011f40
118c:  e320f000     nop  {0}
1190:  e320f000     nop  {0}
1194:  e320f000     nop  {0}
1198:  e320f000     nop  {0}
119c:  e320f000     nop  {0}
```

```
000011a0 <vector_und>:
    11a0:   e88d4001        stm  sp, {r0, lr}
    11a4:   e14fe000        mrs  lr, SPSR
    11a8:   e58de008        str   lr, [sp, #8]
    11ac:   e10f0000        mrs  r0, CPSR
    11b0:   e2200008        eor   r0, r0, #8
    11b4:   e16ff000  msr  SPSR_fsxc, r0
    11b8:   e20ee00f        and  lr, lr, #15
    11bc:   e1a0000d        mov r0, sp
    11c0:   e79fe10e        ldr   lr, [pc, lr, lsl #2]
    11c4:   e1b0f00e        movs       pc, lr
    11c8:   c0012320        .word       0xc0012320
    11cc:   c0011f70        .word       0xc0011f70
    11d0:   c0011f70        .word       0xc0011f70
    11d4:   c00120a0        .word       0xc00120a0
    11d8:   c0011f70        .word       0xc0011f70
    11dc:   c0011f70        .word       0xc0011f70
    11e0:   c0011f70        .word       0xc0011f70
    11e4:   c0011f70        .word       0xc0011f70
    11e8:   c0011f70        .word       0xc0011f70
    11ec:   c0011f70        .word       0xc0011f70
    11f0:   c0011f70        .word       0xc0011f70
    11f4:   c0011f70        .word       0xc0011f70
    11f8:   c0011f70        .word       0xc0011f70
    11fc:   c0011f70        .word       0xc0011f70

    1200:   c0011f70        .word       0xc0011f70

    1204:   c0011f70        .word       0xc0011f70

    1208:   e320f000        nop  {0}

    120c:   e320f000        nop  {0}

    1210:   e320f000        nop  {0}

    1214:   e320f000        nop  {0}

    1218:   e320f000        nop  {0}

    121c:   e320f000        nop  {0}


00001220 <vector_addrexcptn>:

    1220:   eafffffe     b      1220 <vector_addrexcptn>

    1224:   e320f000        nop  {0}

    1228:   e320f000        nop  {0}

    122c:   e320f000        nop  {0}

    1230:   e320f000        nop  {0}

    1234:   e320f000        nop  {0}

    1238:   e320f000        nop  {0}

    123c:   e320f000        nop  {0}


00001240 <vector_fiq_offset>:

    1240:   e24ee004        sub  lr, lr, #4

    1244:   e88d4001        stm  sp, {r0, lr}

    1248:   e14fe000        mrs  lr, SPSR

    124c:   e58de008        str   lr, [sp, #8]
```

```
1250:  e10f0000       mrs  r0, CPSR
1254:  e2200002       eor   r0, r0, #2
1258:  e16ff000  msr SPSR_fsxc, r0
125c:  e20ee00f       and  lr, lr, #15
1260:  e1a0000d       mov r0, sp
1264:  e79fe10e       ldr    lr, [pc, lr, lsl #2]
1268:  e1b0f00e       movs       pc, lr
126c:  c0012540       .word       0xc0012540
1270:  c0012180       .word       0xc0012180
1274:  c0012180       .word       0xc0012180
1278:  c0012180       .word       0xc0012180
127c:  c0012180       .word       0xc0012180
1280:  c0012180       .word       0xc0012180
1284:  c0012180       .word       0xc0012180
1288:  c0012200       .word       0xc0012200
128c:  c0012180       .word       0xc0012180
1290:  c0012180       .word       0xc0012180
1294:  c0012180       .word       0xc0012180
1298:  c0012180       .word       0xc0012180
129c:  c0012180       .word       0xc0012180
12a0:  c0012180       .word       0xc0012180
12a4:  c0012180       .word       0xc0012180
12a8:  c0012180       .word       0xc0012180
12ac:  e320f000       nop  {0}
12b0:  e320f000       nop  {0}
```

```
    12b4:   e320f000        nop  {0}

    12b8:   e320f000        nop  {0}

    12bc:   e320f000        nop  {0}
```

===

上面的code分别位于".vectors"和".stubs" section中。这个在entry-armv.S源码中反映了。

```
        .section .vectors, "ax", %progbits    ①

__vectors_start:

        W(b)      vector_rst

        W(b)      vector_und

        W(ldr)    pc, __vectors_start + 0x1000

        W(b)      vector_pabt

        W(b)      vector_dabt

        W(b)      vector_addrexcptn

        W(b)      vector_irq

        W(b)      vector_fiq
```

这是".vector" section

```
        .section .stubs, "ax", %progbits    ②

__stubs_start:

        @ This must be the first word

        .word     vector_swi
```

```
vector_rst:
 ARM(      swi  SYS_ERROR0       )
 THUMB( svc  #0           )
 THUMB( nop                  )
        b      vector_und


/*
 * Interrupt dispatcher
 */
        vector_stub     irq, IRQ_MODE, 4


        .long__irq_usr                @  0  (USR_26 / USR_32)
        .long__irq_invalid                 @  1  (FIQ_26 / FIQ_32)
        .long__irq_invalid                 @  2  (IRQ_26 / IRQ_32)
        .long__irq_svc                @  3  (SVC_26 / SVC_32)
        .long__irq_invalid                 @  4
        .long__irq_invalid                 @  5
        .long__irq_invalid                 @  6
        .long__irq_invalid                 @  7
        .long__irq_invalid                 @  8
        .long__irq_invalid                 @  9
        .long__irq_invalid                 @  a
        .long__irq_invalid                 @  b
        .long__irq_invalid                 @  c
        .long__irq_invalid                 @  d
        .long __irq_invalid                 @  e

        .long __irq_invalid                 @  f


/*

 * Data abort dispatcher

 * Enter in ABT mode, spsr = USR CPSR, lr = USR PC

 */

        vector_stub     dabt, ABT_MODE, 8



        ......
```

①

.vector是section name,"ax" 表示是relocatable and executable

②

.stubs是section name,"ax" 表示是relocatable and executable

".vectors" section被安排在0地址（page 0 in virtual space），而".stubs" section则被安排在0x1000(page 1 in virtual space).

从反汇编码看，这一目了然，而从源码上是看不出的，要从vmlinux.lds的链接脚本上才能看出。

in vmlinux.lds

```
 /*
```

```
        * The vectors and stubs are relocatable code, and the
        * only thing that matters is their relative offsets
        */
 __vectors_start = .;
 .vectors 0 : AT(__vectors_start) {
  *(.vectors)
 }
 . = __vectors_start + SIZEOF(.vectors);
 __vectors_end = .;
 __stubs_start = .;
 .stubs 0x1000 : AT(__stubs_start) {
  *(.stubs)
 }
 . = __stubs_start + SIZEOF(.stubs);
 __stubs_end = .;
```

从build kernel后生成的system.map文件也能看出端倪。

in config-3.18.7-yocto-standard

```
00000000 t __vectors_start
00000024 A cpu_v7_suspend_size
0000002c A cpu_ca9mp_suspend_size
00001000 t __stubs_start
00001004 t vector_rst
```

```
00001020 t vector_irq

000010a0 t vector_dabt

00001120 t vector_pabt

000011a0 t vector_und

00001220 t vector_addrexcptn

00001240 t vector_fiq

00001240 T vector_fiq_offset
```

位于0地址的".vectors" section只是包含了ARM core的中断向量表。依次为

1. RESET	
2. UNDEF	(undefined instruction, a.k.a. invalid instruction)
3. SWI	(SoftWare Interrupt)
4. PABT	(Prefetch instruction Abort)
5. DABT	(Data Abort)
6. -	?
7. IRQ	hardware interrupt
8. FIQ	Fast Interrupt

第6项在“ARM System Developer's Guide: Designing and Optimizing System Software” book中被标为Reserved。这本书只讲了RAMv5，在ARMv7-A manual中标为：

0x14 Not used

其中SWI (SoftWare Interrupt)有点特殊。

```
        W(ldr)      pc, __vectors_start + 0x1000
```

        反汇编后为

```
        ldr    pc, [pc, #4080]        ; 1000 <__stubs_start>
```

        反汇编码比源码更易理解。

        pc, #4080 指向__stubs_start的开始。

```
__stubs_start:
        @ This must be the first word
        .word       vector_swi
```

__stubs_start的头四个bytes是vector_swi的address，所以

ldr   pc, [pc, #4080] 就是把vector_swi赋值给pc，也就是跳转到vector_swi执行。

位于".stubs" section的都是第一层exception handler。

".vectors" section所在的page实际上就0x20(32)个bytes是有意义的，而page 1所在的".stubs" section包含的first level exception handlers code不过0x2e0

in config-3.18.7-yocto-standard

```
c05a9000 T __vectors_start

c05a9020 T __stubs_start

c05a9020 T __vectors_end

c05a92e0 t __mmap_switched

c05a92e0 T __stubs_end
```

0xc05a92e0 - 0xc05a9020 = 0x2e0

1. exception vector table

```
   0:  ea0003ff  b      1004 <vector_rst>

   4:  ea000465         b      11a0 <vector_und>

   8:  e59ffff0    ldr    pc, [pc, #4080]        ; 1000 <__stubs_start>

   c:  ea000443         b      1120 <vector_pabt>

  10: ea000422         b      10a0 <vector_dabt>

  14: ea000481         b      1220 <vector_addrexcptn>

  18: ea000400         b      1020 <vector_irq>

  1c: ea000487         b      1240 <vector_fiq_offset>
```

当出现exception，ARM core就会载入对应的entry执行。比如reset ARM core,core将跳转到0地址(当vector在0地址的情况下)，也就是这里执行

```
b      1004 <vector_rst>
```

2. vector stub code

比如当hardware interrupt发生时，core将执行

```
  18: ea000400        b      1020 <vector_irq>
```

跳转到下面的位于vector stub code中的irq entry

```
00001020 <vector_irq>:
    1020:   e24ee004        sub  lr, lr, #4            ①
    1024:   e88d4001        stm  sp, {r0, lr}        ②
    1028:   e14fe000        mrs  lr, SPSR
    102c:   e58de008        str    lr, [sp, #8]
    1030:   e10f0000        mrs  r0, CPSR                          ③
    1034:   e2200001        eor   r0, r0, #1
    1038:   e16ff000  msr SPSR_fsxc, r0
    103c:   e20ee00f        and  lr, lr, #15
    1040:   e1a0000d        mov r0, sp                                  ④
    1044:   e79fe10e        ldr    lr, [pc, lr, lsl #2]
    1048:   e1b0f00e        movs       pc, lr
    104c:   c00122c0        .word       0xc00122c0
    1050:   c0011f60        .word       0xc0011f60
    1054:   c0011f60        .word       0xc0011f60
    1058:   c0012000        .word       0xc0012000
    105c:   c0011f60        .word       0xc0011f60
    1060:   c0011f60        .word       0xc0011f60
    1064:   c0011f60        .word       0xc0011f60
    1068:   c0011f60        .word       0xc0011f60
    106c:   c0011f60        .word       0xc0011f60
    1070:   c0011f60        .word       0xc0011f60
    1074:   c0011f60        .word       0xc0011f60
    1078:   c0011f60        .word       0xc0011f60
    107c:   c0011f60        .word       0xc0011f60
    1080:   c0011f60        .word       0xc0011f60
    1084:   c0011f60        .word       0xc0011f60
    1088:   c0011f60        .word       0xc0011f60
```

上面的反汇编码对应的是下面的源码

```
/*
 * Interrupt dispatcher
 */
        vector_stub     irq, IRQ_MODE, 4
```

```
    .long__irq_usr              @  0  (USR_26 / USR_32)
    .long__irq_invalid          @  1  (FIQ_26 / FIQ_32)
    .long__irq_invalid          @  2  (IRQ_26 / IRQ_32)
    .long__irq_svc              @  3  (SVC_26 / SVC_32)
    .long__irq_invalid          @  4
    .long__irq_invalid          @  5
    .long__irq_invalid          @  6
    .long__irq_invalid          @  7
    .long__irq_invalid          @  8
    .long__irq_invalid          @  9
    .long__irq_invalid          @  a
    .long__irq_invalid          @  b
    .long__irq_invalid          @  c
    .long__irq_invalid          @  d
    .long__irq_invalid          @  e
    .long__irq_invalid          @  f
```

vector_stub macro下面是不同mode下的excetion handler，这里就是不同mode下的interrupt handler。

由于Linux只工作在user mode和svc mode，所以上面只有user mode与svc mode对应的handler才assign有意义的handler。

ARM只定义了7种处理器mode，由core的cpsr register的最低5位(bit 0 to bit 4)指示CPU mode in arch/arm/include/uapi/asm/ptrace.h

```
#define USR_MODE         0x00000010
#define SVC_MODE         0x00000013

#define FIQ_MODE    0x00000011
#define IRQ_MODE         0x00000012
#define ABT_MODE         0x00000017
#define HYP_MODE         0x0000001a
#define UND_MODE         0x0000001b
#define SYSTEM_MODE         0x0000001f
```

由于指示mode的bit 4总是1（从上面的定义可知），所以实际上用bit 0 to bit 3可以标示which mode。

上面的mode exception handler共定义了16项（所有mode全用上也不过7项，何况Linux只使用了2项），大部分都填的是__irq_invalid。

user mode和svc mode的在cpsr register中的mode bit pattern是0x10和0x13,去掉bit 4的“1”,值是0x00和0x03,他们正好是mode exception handler table中的

第0项和第3项。

①

vector_stub   irq, IRQ_MODE, 4

这里的4是对lr regsiter的调整。由于RISC CPU的特性，pc register总是先于executing的instruction. 在ARM上是先于2条指令，即8 bytes的offset。每条指令4个bytes.

instruction 1

instruction 2   <--- executing， interrupt occur

instruction 3

instruction 4   <--- pc 指向

instruction 5

instruction 6

由于是在执行instruction 2时发生interrupt,当core切换入interrupt handler时，instruction 2必须执行完(hardware interrupt本身是不能中断正在执行的instruction的，它只能在本指令执行完后再打断)，所以instruction 3是在interrupt handler完成以后应该执行的instruction，即hardware interrupt handler在instruction 2和instruction 3间执行。lr用于记录interrupt handler处理完后要回复到原执行流程的address。而instruction 3的地址正好是pc - 4。这就是这里4的由来。

vector_stub   dabt, ABT_MODE, 8

这里的8则是对产生data abort exception后对lr的调整。它调整不同的值与interrupt是由于下面这样的特性。

interrupt是在执行完当前正在executing的instruction后进入interrupt handler，即当前executing instruction的效果已经结束，当interrupt handler结束，就执行下一条instruction，不能再执行上面那条指令了，否则就错了。而data abort exception之所以产生，就是因为当前executing的instruction没法运行下去（当然就不存在该指令的执行效果的问题），所以当exception handler返回，必须再次执行原来产生exception的指令。

instruction 1

instruction 2   <--- executing, data abort exception occur

instruction 3

instruction 4   <--- pc 指向

instruction 5

instruction 6

假设core在执行instruction 2时发生exception,比如该page不可写。则进入exception handler后，kernel的责任是fix该exception，使得exception handler返回后依然

执行instruction 2.而instruction 2的地址就是pc - 8。

从上可知interrupt是异步的，而exception是同步的。这个在intel CPU manual中有分别，但在ARM manual中好像都统称为"exception"。

②

保存ro,lr和spsr regsiters to stack

③

disable IRQ and enter svc mode

④

根据发生exception发生时的mode跳转到对应的handler。

比如如果发生interrupt时,core运行在user mode，则跳转到__irq_usr handler，如果是svc mode，则跳转到__irq_svc。

由于Linux只工作在这两个mode，所以其它的任意mode,都作出错处理(__irq_invalid).

当kernel起来后,真正的exception vector table和first level exception handler并不在0地址开始，而是在0xffff,f000。

in in config-3.18.7-yocto-standard

CONFIG_VECTORS_BASE=0xffff0000

整个move exception vector table和first level exception handler的代码在early_trap_init() in arch/arm/kernel/traps.c.

```
start_kernel()

       |

       |

      \|/

setup_arch() in arch/arm/kernel/setup.c

       |

       |

      \|/

paging_init() in arch/arm/mm/mmu.c

       |

       |

      \|/

devicemaps_init() in arch/arm/mm/mmu.c

       |

       |

      \|/

early_trap_init() in arch/arm/mm/mmu.c
```

in devicemaps_init()

```
	/*
	 * Allocate the vector page early.
	 */
	vectors = early_alloc(PAGE_SIZE * 2);  ①

	early_trap_init(vectors);
```

①

由于kernel的memory manager(比如buddy allocator, slab allocator，etc)都没初始化，这里分配2 pages的memory用的是memblock module(当kernel起来后就丢弃不用了)

```
void __init early_trap_init(void *vectors_base)
{
#ifndef CONFIG_CPU_V7M                                              ②
	unsigned long vectors = (unsigned long)vectors_base;
	extern char __stubs_start[], __stubs_end[];                ③
	extern char __vectors_start[], __vectors_end[];
	unsigned i;

	vectors_page = vectors_base;

	/*
```

```
	 * Poison the vectors page with an undefined instruction.  This
	 * instruction is chosen to be undefined for both ARM and Thumb
	 * ISAs.  The Thumb version is an undefined instruction with a
	 * branch back to the undefined instruction.
	 */
	for (i = 0; i < PAGE_SIZE / sizeof(u32); i++)                    ④
		((u32 *)vectors_base)[i] = 0xe7fddef1;

	/*
	 * Copy the vectors, stubs and kuser helpers (in entry-armv.S)
	 * into the vector page, mapped at 0xffff0000, and ensure these
	 * are visible to the instruction stream.
	 */
	memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);        ⑤
	memcpy((void *)vectors + 0x1000, __stubs_start, __stubs_end - __stubs_start);    ⑥

	kuser_init(vectors_base);                                        ⑦

	flush_icache_range(vectors, vectors + PAGE_SIZE * 2);        ⑧
	modify_domain(DOMAIN_USER, DOMAIN_CLIENT);
#else /* ifndef CONFIG_CPU_V7M */
	/*
	 * on V7-M there is no need to copy the vector table to a dedicated
	 * memory area. The address is configurable and so a table in the kernel
	 * image can be used.
```

```
         */
#endif
}
```

②

CONFIG_CPU_V7M is for Cortext-M，当然没有定义该macro。

③

这些extern的global variable都定义在vmlinux.lds中

```
 . = __vectors_start + SIZEOF(.vectors);
 __vectors_end = .;
 __stubs_start = .;
 .stubs 0x1000 : AT(__stubs_start) {
  *(.stubs)
 }
 . = __stubs_start + SIZEOF(.stubs);
 __stubs_end = .;
```

④

从comments可知，是把存放exception vector table的page先填满invalid instruction.

⑤

把exception vector table复制过去，这样除了头上32个字节是exception vector table，该page的其他内容都是invalid instruction。

⑥

把first level exception handler的code复制到第2页.

⑦

CONFIG_KUSER_HELPERS=y

in arch/arm/kernel/traps.c

```
#ifdef CONFIG_KUSER_HELPERS
static void __init kuser_init(void *vectors)
{
        extern char __kuser_helper_start[], __kuser_helper_end[];
        int kuser_sz = __kuser_helper_end - __kuser_helper_start;

        memcpy(vectors + 0x1000 - kuser_sz, __kuser_helper_start, kuser_sz);

        /*
         * vectors + 0xfe0 = __kuser_get_tls
         * vectors + 0xfe8 = hardware TLS instruction at 0xffff0fe8
         */
        if (tls_emu || has_tls_reg)
                memcpy(vectors + 0xfe0, vectors + 0xfe8, 4);
}
#else
```

```
static inline void __init kuser_init(void *vectors)

{

}

#endif
```

__kuser_helper_start and __kuser_helper_end 定义在arch/arm/kernel/entry-armv.S

in system.map

```
c05acb80 t __kuser_cmpxchg64

c05acb80 T __kuser_helper_start

c05acbc0 t __kuser_memory_barrier

c05acbe0 t __kuser_cmpxchg

c05acc00 t __kuser_get_tls

c05acc1c t __kuser_helper_version

c05acc20 T __kuser_helper_end
```

c05acc20 - c05acb80 = 0xa0 (160 bytes)

kuser_init()把从__kuser_helper_start to __kuser_helper_end的code复制到vector page的尾部。

这样vector page的头上是32个bytes的exception vectors(8 entries),尾部是160bytes的kuser helper code，而中间被填充了invalid instruction(0xe7fddef1)

具体kuser helper的分析，见《about kuser helper》笔记。

由于无论是vector page中的code(exception table中的跳转指令或是kuser helper code),还是vector stub page中的code都生成在0地址开始的virtual address,

但都被复制到0xffff,f000开始的地址去运行，所以这些code必须是relocatable code，即这些code本身间的跳转不能有绝对地址的跳转，只能有相对地址的跳转。当然对这些relocatable code本身外的跳转是完全可以使用绝对地址的。

mapping vector page to virtual address **0xffff,0000**

and vector stub page to **0xffff,1000**

```
static void __init devicemaps_init(const struct machine_desc *mdesc)

{

        struct map_desc map;

        unsigned long addr;

        void *vectors;



        /*

         * Allocate the vector page early.

         */

        vectors = early_alloc(PAGE_SIZE * 2);



        early_trap_init(vectors);



        for (addr = VMALLOC_START; addr; addr += PMD_SIZE)

                pmd_clear(pmd_off_k(addr));



        ......
```

```
	/*
	 * Create a mapping for the machine vectors at the high-vectors
	 * location (0xffff0000).  If we aren't using high-vectors, also
	 * create a mapping at the low-vectors virtual address.
	 */
	map.pfn = __phys_to_pfn(virt_to_phys(vectors));            ①
	map.virtual = 0xffff0000;
	map.length = PAGE_SIZE;
#ifdef CONFIG_KUSER_HELPERS
	map.type = MT_HIGH_VECTORS;
#else
	map.type = MT_LOW_VECTORS;
#endif
	create_mapping(&map);

	if (!vectors_high()) {                                              ②
		map.virtual = 0;
		map.length = PAGE_SIZE * 2;
		map.type = MT_LOW_VECTORS;
		create_mapping(&map);
	}

	/* Now create a kernel read-only mapping */
	map.pfn += 1;                                                          ③
```

```
        map.virtual = 0xffff0000 + PAGE_SIZE;

        map.length = PAGE_SIZE;

        map.type = MT_LOW_VECTORS;

        create_mapping(&map);


        ......

}
```

①

mapping vector page to 0xffff0000,建立virtual page v.s. physical page之间的page table.

②

```
#if __LINUX_ARM_ARCH__ >= 4

#define vectors_high()    (get_cr() & CR_V)

#else

#define vectors_high()    (0)

#endif


#define CR_V (1 << 13) /* Vectors relocated to 0xffff0000    */
```

In ARMv7-A manual,

If the Security Extensions are not implemented there is a single exception base address. This is controlled by the SCTLR.V bit:

V == 0 Exception base address = 0x00000000. This setting is referred to as normal vectors, or as low

vectors.

V == 1 Exception base address = 0xFFFF0000. This setting is referred to as high vectors, or Hivecs.

Note

Use of the Hivecs setting, V == 1, is deprecated in ARMv7-R. ARM recommends that Hivecs is used only

in ARMv7-A implementations.

```
static inline unsigned long get_cr(void)

{

        unsigned long val;

        asm("mrc p15, 0, %0, c1, c0, 0      @ get CR" : "=r" (val) : : "cc");

        return val;

}
```

mapping vector page and vector stub page 2 virtual address 0 and 0x1000

③

mapping vector stub page to 0xffff0000 + 0x1000