### blog.chinaunix.net/uid-20543672-id-3184456.html

### Preface 引言

我在这里用一些篇幅来描述一下arm体系结构下Linux中怎样来初始化中断向量表的，因为这个方法很具有通用性，我把它叫做代码大挪移。您说搬代码谁不会阿，不就是拷贝吗，的确如此，但是拷贝也有技巧。拷贝很简单啦，其实就是memcpy,这不用提，**我在这里想说的是，你怎么把你的代码设计成能随便拷贝的，换句专业的术语，叫与位置无关的代码，拷到哪都能用。**我以前也用过类似的方法作启动，今天拿来说说。

#### Scenario 1 第一场景 copy

我们先看实际复制动作。代码的位置在arch/arm/traps.c中，kernel version: 2.6.27。 这个是初始化部分的代码，setup\_arch()->early\_trap\_init(). 熟悉初始化部分的朋友们可能见到过这段代码。

void \_\_init early\_trap\_init(void)

{

unsigned long vectors = CONFIG\_VECTORS\_BASE;

extern char \_\_stubs\_start[], \_\_stubs\_end[];

extern char \_\_vectors\_start[], \_\_vectors\_end[];

extern char \_\_kuser\_helper\_start[], \_\_kuser\_helper\_end[];

int kuser\_sz = \_\_kuser\_helper\_end - \_\_kuser\_helper\_start;

/\*

\* 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);

memcpy((void \*)vectors + 0x200, \_\_stubs\_start, \_\_stubs\_end - \_\_stubs\_start);

memcpy((void \*)vectors + 0x1000 - kuser\_sz, \_\_kuser\_helper\_start, kuser\_sz);

…

}

实际copy动作一目了然，就是两个memcpy（第三个实际上是拷贝一些别的东西，原理是一样的，这里不提了）. Copy的源是vectors,这个值是CONFIG\_VECTORS\_BASE，一般来讲，是0xffff0000,当然你可以根据硬件的设定自己配制这个值。把什么东西往那copy呢?第一部分是从\_\_vectors\_start到\_\_vectors\_end之间的代码，第二部分是从\_\_stubs\_start到\_\_stubs\_end之间的代码，而第二部分是copy到vectors + 0x200起始的位置。也就是说，两部分之间的距离是0x200,即512个字节。

我们来看\_\_vectors\_start，\_\_vectors\_end，\_\_stubs\_start，\_\_stubs\_end到底是什么东西，只要知道它们在哪里定义的，就知道怎么回事了。

我们来看\_\_vectors\_start，\_\_vectors\_end，\_\_stubs\_start，\_\_stubs\_end到底是什么东西，只要知道它们在哪里定义的，就知道怎么回事了。

#### Scenario 2 第二场景 主角闪亮登场

它们埋伏在arch/arm/kernel/entry-armv.S中，这个文件是arm中各个模式的入口代码，熟悉arm的朋友们知道arm有几种模式，不知道的自己查查，不说了。我们取一个片断，和我们的阐述相关的部分。为了让大家看得更清楚，我删掉了部分代码和注释，把主干凸显出来。有兴趣的朋友可以查看源代码，研究全部，里面还是比较有内涵的。

**.globl \_\_stubs\_start**

**\_\_stubs\_start:**

/\*

\* Interrupt dispatcher

\*/

vector\_stub irq, IRQ\_MODE, 4

// 请注意这里：vector\_stub是一个宏，展开后是一块代码，下面是个跳转表，我们将代码结

// 构展开，大致是这样的结构: (后面的vector\_stub dabt, ABT\_MODE, 8等展开过程全一样，在此略过不提)

// -------------------------------- begin 展开

.align 5

vector\_irq:

sub lr, lr, 4

@ Save r0, lr\_ (parent PC) and spsr\_

@ (parent CPSR)

@

stmia sp, {r0, lr} @ save r0, lr

mrs lr, spsr

str lr, [sp, #8] @ save spsr

@ Prepare for SVC32 mode. IRQs remain disabled.

@

mrs r0, cpsr

eor r0, r0, IRQ\_MODE ^ SVC\_MODE)

msr spsr\_cxsf, r0

@ the branch table must immediately follow this code

@

and lr, lr, #0x0f

mov r0, sp

ldr lr, [pc, lr, lsl #2]

movs pc, lr @ branch to handler in SVC mode

// -------------------------------- end 展开

.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 @ f

/\*

\* Data abort dispatcher

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

\*/

vector\_stub dabt, ABT\_MODE, 8

.long \_\_dabt\_usr @ 0 (USR\_26 / USR\_32)

.long \_\_dabt\_invalid @ 1 (FIQ\_26 / FIQ\_32)

.long \_\_dabt\_invalid @ 2 (IRQ\_26 / IRQ\_32)

.long \_\_dabt\_svc @ 3 (SVC\_26 / SVC\_32)

。。。

.long \_\_dabt\_invalid @ f

/\*

\* Prefetch abort dispatcher

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

\*/

vector\_stub pabt, ABT\_MODE, 4

.long \_\_pabt\_usr @ 0 (USR\_26 / USR\_32)

.long \_\_pabt\_invalid @ 1 (FIQ\_26 / FIQ\_32)

.long \_\_pabt\_invalid @ 2 (IRQ\_26 / IRQ\_32)

.long \_\_pabt\_svc @ 3 (SVC\_26 / SVC\_32)

。。。

.long \_\_pabt\_invalid @ f

/\*

\* Undef instr entry dispatcher

\* Enter in UND mode, spsr = SVC/USR CPSR, lr = SVC/USR PC

\*/

vector\_stub und, UND\_MODE

.long \_\_und\_usr @ 0 (USR\_26 / USR\_32)

.long \_\_und\_invalid @ 1 (FIQ\_26 / FIQ\_32)

.long \_\_und\_invalid @ 2 (IRQ\_26 / IRQ\_32)

.long \_\_und\_svc @ 3 (SVC\_26 / SVC\_32)

。。。

.long \_\_und\_invalid @ f

.align 5

vector\_fiq:

disable\_fiq

subs pc, lr, #4

vector\_addrexcptn:

b vector\_addrexcptn

/\*

\* We group all the following data together to optimise

\* for CPUs with separate I & D caches.

\*/

.align 5

.LCvswi:

.word vector\_swi

**.globl \_\_stubs\_end**

**\_\_stubs\_end:**

.equ stubs\_offset, \_\_vectors\_start + 0x200 - \_\_stubs\_start

**.globl \_\_vectors\_start**

**\_\_vectors\_start:**

swi SYS\_ERROR0

b vector\_und + stubs\_offset

ldr pc, .LCvswi + stubs\_offset

b vector\_pabt + stubs\_offset

b vector\_dabt + stubs\_offset

b vector\_addrexcptn + stubs\_offset

b vector\_irq + stubs\_offset

b vector\_fiq + stubs\_offset

**.globl \_\_vectors\_end**

**\_\_vectors\_end:**

为了让大家看得更清，我把代码的结构再次简化成这样：

**.globl \_\_stubs\_start**

**\_\_stubs\_start:**

.align 5

vector\_irq:

**[code part] // 展开代码**

**[jump table part] // 地址跳转表**

**。。。**

**.align 5**

**vector\_dabt:**

**[code part]**

**[jump table part]**

**。。。**

**.align 5**

**vector\_ pabt:**

**[code part]**

**[jump table part]**

**。。。**

**.align 5**

**vector\_und:**

**[code part]**

**[jump table part]**

**。。。**

**.align 5**

**vector\_fiq:**

**。。。**

**.globl \_\_stubs\_end**

**\_\_stubs\_end:**

**.globl \_\_vectors\_start**

**\_\_vectors\_start:**

swi SYS\_ERROR0

b vector\_und + stubs\_offset

ldr pc, .LCvswi + stubs\_offset

b vector\_pabt + stubs\_offset

b vector\_dabt + stubs\_offset

b vector\_addrexcptn + stubs\_offset

b vector\_irq + stubs\_offset

b vector\_fiq + stubs\_offset

.globl \_\_vectors\_end

\_\_vectors\_end:

在这里我不花过多的篇幅去解释代码的意思，这不是本文的目的，只要你把结构看清，就达到目的了。但我会花点时间研究一下展开代码部分（蓝色）的特征,这部分代码是与位置无关的代码，我们稍微研究一下，它为什么会这么写。

.align 5

vector\_irq:

[code part] // 展开代码

[jump table part] // 地址跳转表

。。。

首先这部分代码大致都是一样的结构，前面是一些代码，后面跟着一个跳转表。跳转表里面定义了一些地址。我们截取这部分看

。。。

@ the branch table must immediately follow this code

@

**and lr, lr, #0x0f (1) // lr中当前存储了上一个状态寄存器的值，对后几位做与，**

**// 就是取在中断前处在用户态还是核心态，这个值用作跳**

**// 转表的索引**

**mov r0, sp (2) // 用做他用，sp值当第一个参数传给后面函数**

**ldr lr, [pc, lr, lsl #2] (3) // pc是当前执行指令地址加8，即跳转表的基地址，lr是索引**

**// 很好的技巧，取pc找当前地址什么时候都没错**

**mov pc, lr @ branch to handler in SVC mode**

**[jump table]**

.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)

**真正的跳转在最后一句完成，大家都看得很清楚。跳到哪里去了，如果中断以前是svc模式，就会跳到\_\_irq\_svc。我们发现这里不会直接用b(bl,bx等)个，**

**ü 一是b跳转后面是个偏移，而这个偏移是有限制的，不能太大**

**ü 二是b跳转后面的偏移你不知道在代码拷贝后还是不是那个样子，因为我们要搬移代码，所以如果你不能确定搬移后的偏移不变，那你就用绝对地址，而上面的代码前三句就是算出绝对地址来，然后用绝对地址赋值给pc直接完成跳转。**

这些都是一些技巧，总之你要注意的是写位置无关的代码时涉及到跳转部分，用b跳转还是直接赋成绝对地址（通过跳转表实现），如果你不能保证搬移后的偏移一致，写这部分就要注意了，要用一些技巧的。

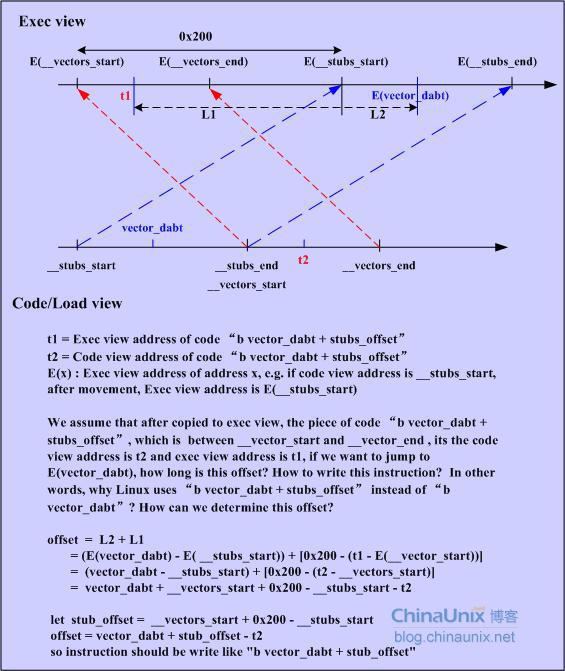
大家可以去用gcc 的-fPIC和-S选项汇编一个小的函数看看，fPIC就是与位置无关选项，相信编译过动态库的人都熟悉，看看它是怎么做的。你会发现异曲同工。

#### Scenario 3 第三场景 大搬移

我用一个章节来介绍大搬移的过程，以及一些在搬移中Linux出现的问题及解决方案。我把整个的搬移过程做成一张图里，然后讨论了一些技术细节。我们看到这是一个巨大无比的图，我们这章节的所阐述的内容都在图里。

我们将搬移前的代码组织称为Code/Load 视图，因为这是代码中的（或image中的）组织情况，把搬移后的代码组织称为Exec视图，反映的是代码执行时代码在内存中的情况。我刚才讲过了第一场景的情况，忘了的回到第一场景中去看，两个memcpy的执行过程在图中也有表示，就是蓝色和红色的带箭头的虚线，这就是代码从code view到exec view的拷贝过程，一目了然，不用多说。

现在出现了一个问题，就是我们发现在\_\_vector\_start和\_\_vector\_end之间的代码有点怪异，我们再次摘到这里来看：



.equ stubs\_offset, \_\_vectors\_start + 0x200 - \_\_stubs\_start

**.globl \_\_vectors\_start**

**\_\_vectors\_start:**

swi SYS\_ERROR0

b vector\_und + stubs\_offset

ldr pc, .LCvswi + stubs\_offset

b vector\_pabt + stubs\_offset

b vector\_dabt + stubs\_offset

b vector\_addrexcptn + stubs\_offset

b vector\_irq + stubs\_offset

b vector\_fiq + stubs\_offset

**.globl \_\_vectors\_end**

**\_\_vectors\_end:**

在第二个场景中我们说过，这叫做位置无关的代码，因为要拷贝到别的地方。而且里面都是跳转指令。我们发现了除了第三个行代码用了绝对地址进行了跳转，其它都是用的b跳转。举个例子，b vector\_dabt + stubs\_offset，(vector\_dabt在\_\_stubs\_start和\_\_stubs\_end之间)，如果你用b vector\_dabt, 这肯定是有问题的，因为copy之后exec view的组织（map）是不一样的，所以b后这个偏移就不对了。这里面，我们就要对这个偏移进行一次调整。Stubs\_offset就是这个调整值，是可以计算出来的，具体的计算过程在图中讲得比较清楚，这里不提了。大家可以在图中看到详细的推导过程。

其实尽管ldr pc, .LCvswi + stubs\_offset这条指令用的是绝对地址跳转，用得跳转表的方法，但找地址的过程也用到了这个技术。我们看到

.align 5

.LCvswi:

.word vector\_swi

.LCvswi这个位置存储的是一个地址，就是要跳到这个地方。.align 5的意思是32字节对齐，这个是保证cache line对齐的，不提了。在exec view中找这个地址，就得加上个offset.原理是一样的，因为.LCvswi在\_\_stubs\_start和\_\_stubs\_end之间，这个区域被搬移了，不能直接用这个标签地址了，vector\_swi没有被搬移，所以可以直接用。

总结一下。我觉得我要讲的东西虽然是Linux中的技术细节，描述的确是代码搬移过程原理和注意事项。其实更重要的是，我们如何把这一个过程倒过来，即在涉及到代码搬移的场合中如何进行设计，如何运用这些技术实现这一设计过程。你可以遵循这样的指导步骤：

1． 画出那个大图来，按自己的要求确定Code view和Exec view, 设计搬移区段和设计搬移的位置

2． 写出要搬移的代码，运用位置无关的技术（上面提到的）进行编码和检验

3． 用类似memcpy的代码进行搬移