kernel hacker 修炼之道——李万鹏

男儿立志出乡关， 学不成名死不还。 埋骨何须桑梓地， 人生无处不青山。 ——西乡隆盛诗

kernel hacker修炼之道之PCI subsystem(五)

分类： linux内核编程 PCI-e/PCI 2012-02-26 16:00 399人阅读 评论(0) 收藏 举报

kernel hacker修炼之道之PCI subsystem(五)

作者 李万鹏

pci_setup_device还会调用一个重要的函数pci_read_bases：

```
244static void pci_read_bases(struct pci_dev *dev, unsigned int howmany, int rom)
245{
246 unsigned int pos, reg;
247
248 for (pos = 0; pos < howmany; pos++) {
249 struct resource *res = &dev->resource[pos];
250 reg = PCI_BASE_ADDRESS_0 + (pos << 2);
251 pos += __pci_read_base(dev, pci_bar_unknown, res, reg);
252 }
253
254 if (rom) {
255 struct resource *res = &dev->resource[PCI_ROM_RESOURCE];
256 dev->rom_base_reg = rom;
257 res->flags = IORESOURCE_MEM | IORESOURCE_PREFETCH |
258 IORESOURCE_READONLY | IORESOURCE_CACHEABLE |
259 IORESOURCE_SIZEALIGN;
260 __pci_read_base(dev, pci_bar_mem32, res, rom);
261 }
262}
```

这个函数从pci device的配置空间总读取BAR里存放的resource的基地址和长度，填充pci_dev的resource数组。这个函数里的for循环以PCI_BASE_ADDRESS_0为基地址，它定义在include/linux/pci_regs.h文件中，其实就是BAR0在PCI device configuration space中的偏移罢了。其中normal pci device(对应htype 0)有BAR0～5，bridge pci device(对应htype 1)只有BAR0～1。

```
84#define PCI_BASE_ADDRESS_0 0x10 /* 32 bits */
85#define PCI_BASE_ADDRESS_1 0x14 /* 32 bits [htype 0,1 only] */
86#define PCI_BASE_ADDRESS_2 0x18 /* 32 bits [htype 0 only] */
87#define PCI_BASE_ADDRESS_3 0x1c /* 32 bits */
88#define PCI_BASE_ADDRESS_4 0x20 /* 32 bits */
89#define PCI_BASE_ADDRESS_5 0x24 /* 32 bits */
```

你可以自己打印出来看看：

```
pci_setup_device 0000:01:00.0: Resource 0: 0000000000008000..0000000000008007 [20101]
pci_setup_device 0000:01:00.0: Resource 1: 0000000000008040..0000000000008043 [20101]
pci_setup_device 0000:01:00.0: Resource 2: 0000000000008200..0000000000008207 [20101]
pci_setup_device 0000:01:00.0: Resource 3: 0000000000008800..0000000000008803 [20101]
pci_setup_device 0000:01:00.0: Resource 4: 0000000000900000..000000000090000f [20101]
pci_setup_device 0000:01:00.0: Resource 5: 0000000000800000..00000000008003ff [20200]
```

struct resource结构定义在include/linux/ioport.h文件中：

```
18struct resource {
19 resource_size_t start;
20 resource_size_t end;
21 const char *name;
22 unsigned long flags;
23 struct resource *parent, *sibling, *child;
24};
```

可以看到struct resource结构也是组成了一个树状的结构。在pci_setup_device中调用pci_read_bases函数的时候传递的howmany参数是6。这里首先获取resource I/O，MMIO，prefetch MMIO的资源起始地址和长度，然后是ROM的起始地址和长度。

```
133int __pci_read_base(struct pci_dev *dev, enum pci_bar_type type,
134 struct resource *res, unsigned int pos)
135{
136 u32 l, sz, mask;
137 u16 orig_cmd;
138
139 mask = type ? PCI_ROM_ADDRESS_MASK : ~0;
140
141 if (!dev->mmio_always_on) {
142 pci_read_config_word(dev, PCI_COMMAND, &orig_cmd);
143 pci_write_config_word(dev, PCI_COMMAND,
144 orig_cmd & ~(PCI_COMMAND_MEMORY | PCI_COMMAND_IO));
145 }
146
147 res->name = pci_name(dev);
148
149 pci_read_config_dword(dev, pos, &l);
150 pci_write_config_dword(dev, pos, l | mask);
151 pci_read_config_dword(dev, pos, &sz);
152 pci_write_config_dword(dev, pos, l);
153
154 if (!dev->mmio_always_on)
155 pci_write_config_word(dev, PCI_COMMAND, orig_cmd);
156
157 /*
158 * All bits set in sz means the device isn't working properly.
159 * If the BAR isn't implemented, all bits must be 0. If it's a
160 * memory BAR or a ROM, bit 0 must be clear; if it's an io BAR, bit
161 * 1 must be clear.
162 */
163 if (!sz || sz == 0xffffffff)
164 goto fail;
165
166 /*
167 * I don't know how l can have all bits set. Copied from old code.
168 * Maybe it fixes a bug on some ancient platform.
169 */
170 if (l == 0xffffffff)
171 l = 0;
172
173 if (type == pci_bar_unknown) {
174 res->flags = decode_bar(dev, l);
175 res->flags |= IORESOURCE_SIZEALIGN;
176 if (res->flags & IORESOURCE_IO) {
```

```
177 l &= PCI_BASE_ADDRESS_IO_MASK;
178 mask = PCI_BASE_ADDRESS_IO_MASK & (u32) IO_SPACE_LIMIT;
179 } else {
180 l &= PCI_BASE_ADDRESS_MEM_MASK;
181 mask = (u32)PCI_BASE_ADDRESS_MEM_MASK;
182 }
183 } else {
184 res->flags |= (l & IORESOURCE_ROM_ENABLE);
185 l &= PCI_ROM_ADDRESS_MASK;
186 mask = (u32)PCI_ROM_ADDRESS_MASK;
187 }
188
189 if (res->flags & IORESOURCE_MEM_64) {
190 。。。。。。。。。。。。。。
225 } else {
226 sz = pci_size(l, sz, mask);
227
228 if (!sz)
229 goto fail;
230
231 res->start = l;
232 res->end = l + sz;
233
234 dev_printk(KERN_DEBUG, &dev->dev, "reg %x: %pR\n", pos, res);
235 }
236
237 out:
238 return (res->flags & IORESOURCE_MEM_64) ? 1 : 0;
239 fail:
240 res->flags = 0;
241 goto out;
242}
```

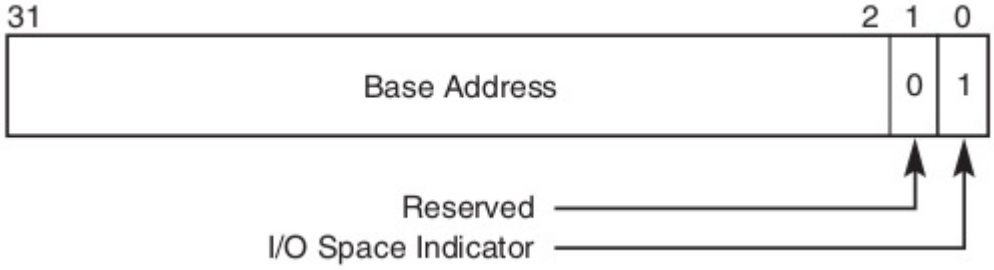
首先分析对起始地址和长度的读取：

```
149 pci_read_config_dword(dev, pos, &l);
150 pci_write_config_dword(dev, pos, l | mask);
151 pci_read_config_dword(dev, pos, &sz);
152 pci_write_config_dword(dev, pos, l);
```

这里读取resource起始地址的时候调用pci_read_config_dword函数，得到的起始地址放入l变量中。然后向BAR写入全1，这样再读一次就可以读取长度，然后将原来的地址写回去。如果读出的长度为0或为0xffffffff说明pci device工作不正常。

在32位PowerPC平台上BAR都是32位的，其中bit 0的read-only，用来指明这是一个IO BAR，还是MMIO BAR。如果BAR的bit 0值为0则是MMIO BAR，否则是IO BAR。

对于IO BAR来说，bit 1被reserved，读取的时候会返回0，其他位用来把device映射到IO space。



对于MMIO BAR来说，bit 2，bit 1用来指明MMIO BAR是映射32 bit wide memory space还是64 bit wide memory space，bit 3指明只否是prefetchable的。

下面来看对resource size的计算，代码里有这么一段comments：

```
78/*
79 * Base addresses specify locations in memory or I/O space.
80 * Decoded size can be determined by writing a value of
81 * 0xffffffff to the register, and reading it back. Only
82 * 1 bits are decoded.
```

83 */也就是说向BAR中写入全1读出的那个长度只有从最低那个1开始才是有效的。在include/linux/pci_regs.h中有：

```
98#define PCI_BASE_ADDRESS_MEM_MASK (~0x0fUL)
99#define PCI_BASE_ADDRESS_IO_MASK (~0x03UL)
```

可以看到从MMIO BAR中读取的时候是屏蔽掉低4位，从IO BAR中读取的时候是屏蔽掉低2位：

```
71static u64 pci_size(u64 base, u64 maxbase, u64 mask)
72{
73 u64 size = mask & maxbase; /* Find the significant bits */
74 if (!size)
75 return 0;
76
77 /* Get the lowest of them to find the decode size, and
78 from that the extent. */
79 size = (size & ~(size-1)) - 1;
80
81 /* base == maxbase can be valid only if the BAR has
82 already been programmed with all 1s. */
83 if (base == maxbase && ((base | size) & mask) != mask)
84 return 0;
85
86 return size;
```

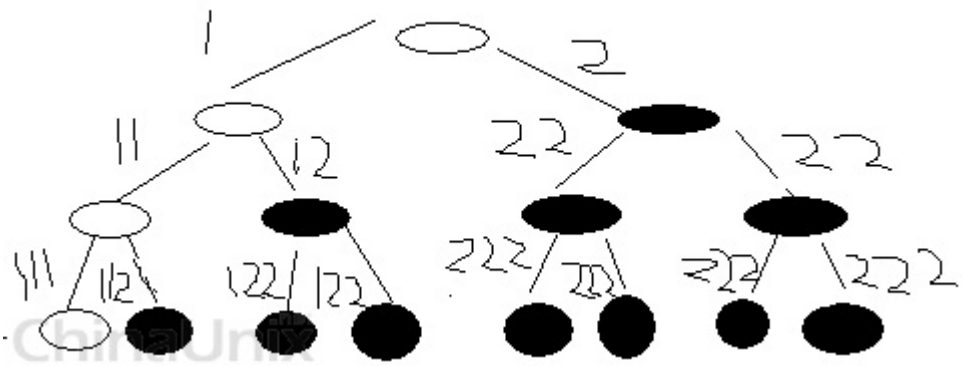
87}比如把最低的4位或2位屏蔽掉后得到的size为0xffff0100，则(size-1)为0xffff00ff，而~(size-1)为0x0000ff00，最后得到size=0xffff0100&0x0000ff00=0x100，这样就把size的二进制中最低那个1提取出来了。

初始状态：PCI桥的树中，BIOS已经初始化了部分PCI桥，分配了总线号；另一部分，BIOS未初始化，需kernel来分配总线号。

那么，我们需要做的，是两次深度优先遍历PCI桥的树。
第一次： 一旦遇到BIOS未初始化的PCI桥，既向上返回一层，目的是统计BIOS分配到的最大总线号；
第二次： 一旦遇到BIOS已初始化的PCI桥，既向下深入一层，目的是继续分配总线号；

既两次遍历，但每个节点只处理一次，要么在第一次中被统计，要么在第二次中被分配总线号。
但是kernel不是这么做的，它只是一次遍历PCI桥的树。
pci_do_scan_bus中的两次遍历只是个假象，因为pass是个局部变量，只有当子节点的两遍都完成后才向上返回父节点。

这里有一张图：



圆圈表示BIOS已初始化的PCI桥；
黑蛋表示BIOS未初始化，需要kernel来初始化的PCI桥；
数字表示递归的各层中pass的值，如“112”表示递归进行到了第三层。并且第一层中pass==1，第二层中pass==1，第三层中pass==2。
kernel遍历的过程是这样的：1(统计) -->11(统计) -->111(统计) --> 112(分配) -->12(分配) -->122(分配) --> 122(分配) --> 2(分配).....
这棵树只被遍历了一次，而且112,12,122,122时，为PCI桥分配总线号都是错误的。因为在我画的图中，右子树全部是黑蛋，但是实际情况中，右子树完全可能都是圆圈，既还有BIOS已分配的总线号未统计上来。这样，在统计完右子树之前，就为左子树中的黑蛋分配了总线号。完全可能造成总线号重复的情况。
只有在2,22,222时，既递归的每层中，pass都等于2时，才能证明已经遍历了所有圆圈，统计出来黑蛋应该使用那些总线号。
内核中是怎么处理这个问题的呢，我们来看代码，下面来分析在上一篇blog中介绍的pci_scan_child_bus会调用到的pci_scan_bridge函数：

```
636 */
637int __devinit pci_scan_bridge(struct pci_bus *bus, struct pci_dev *dev, int max, int pass)
638{
```

```
639 struct pci_bus *child;
640 int is_cardbus = (dev->hdr_type == PCI_HEADER_TYPE_CARDBUS);
641 u32 buses, i, j = 0;
642 u16 bctl;
643 u8 primary, secondary, subordinate;
644 int broken = 0;
645
646 pci_read_config_dword(dev, PCI_PRIMARY_BUS, &buses);
647 primary = buses & 0xFF;
648 secondary = (buses >> 8) & 0xFF;
649 subordinate = (buses >> 16) & 0xFF;
650
651 dev_dbg(&dev->dev, "scanning [bus %02x-%02x] behind bridge, pass %d\n",
652 secondary, subordinate, pass);
653
654 /* Check if setup is sensible at all */
655 if (!pass &&
656 (primary != bus->number || secondary <= bus->number)) {
657 dev_dbg(&dev->dev, "bus configuration invalid, reconfiguring\n");
658 broken = 1;
659 }
660
661 /* Disable MasterAbortMode during probing to avoid reporting
662 of bus errors (in some architectures) */
663 pci_read_config_word(dev, PCI_BRIDGE_CONTROL, &bctl);
664 pci_write_config_word(dev, PCI_BRIDGE_CONTROL,
665 bctl & ~PCI_BRIDGE_CTL_MASTER_ABORT);
666
667 if ((secondary || subordinate) && !pcibios_assign_all_busses() &&
668 !is_cardbus && !broken) {
669 unsigned int cmax;
670 /*
671 * Bus already configured by firmware, process it in the first
672 * pass and just note the configuration.
673 */
674 if (pass)
675 goto out;
676
677 /*
678 * If we already got to this bus through a different bridge,
679 * don't re-add it. This can happen with the i450NX chipset.
680 *
681 * However, we continue to descend down the hierarchy and
682 * scan remaining child buses.
683 */
684 child = pci_find_bus(pci_domain_nr(bus), secondary);
685 if (!child) {
686 child = pci_add_new_bus(bus, dev, secondary);
687 if (!child)
688 goto out;
689 child->primary = primary;
690 child->subordinate = subordinate;
691 child->bridge_ctl = bctl;
692 }
693
694 cmax = pci_scan_child_bus(child);
695 if (cmax > max)
696 max = cmax;
697 if (child->subordinate > max)
698 max = child->subordinate;
```

首先读取primary bus number，secondary bus number，subordinate bus number，注意这里的pci_read_config_dword是读取长字，所以一次都取出来了，下图是configure space的头的一部分：

Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Pr Bus

然后这里有个if-else，if那部分是在pass 0执行的，主要是用来获得最大的总线号，else那部分是在pass 1的时候执行的，主要是为BIOS未分配总线号的bus分配总信号。这里要注意if里的判断条件pcibios_assign_all_busses()，这个是什么意思呢？在arch/powerpc/include/asm/pci.h中有定义：

```
46#define pcibios_assign_all_busses() \
47 (pci_has_flag(PCI_REASSIGN_ALL_BUS))
```

其实也就是看pci_flags的PCI_REASSIGN_ALL_BUS标志位有没有设置，那这个标志代表什么意思呢？在include/asm-generic/pci-bridge.h中定义：

```
12enum {
13 /* Force re-assigning all resources (ignore firmware
14 * setup completely)
15 */
16 PCI_REASSIGN_ALL_RSRC = 0x00000001,
17
18 /* Re-assign all bus numbers */
19 PCI_REASSIGN_ALL_BUS = 0x00000002,
20
21 /* Do not try to assign, just use existing setup */
22 PCI_PROBE_ONLY = 0x00000004,
23
24 /* Don't bother with ISA alignment unless the bridge has
25 * ISA forwarding enabled
26 */
27 PCI_CAN_SKIP_ISA_ALIGN = 0x00000008,
28
29 /* Enable domain numbers in /proc */
30 PCI_ENABLE_PROC_DOMAINS = 0x00000010,
31 /* ... except for domain 0 */
32 PCI_COMPAT_DOMAIN_0 = 0x00000020,
33};
```

那么这个标志可以在哪里设置呢，查看一下平台相关的函数find_and_init_phbs在arch/powerpc/kernel/rtas_pci.c文件中：

```
277
278 /*
279 * pci_probe_only and pci_assign_all_buses can be set via properties
280 * in chosen.
281 */
282 if (of_chosen) {
283 const int *prop;
284
285 prop = of_get_property(of_chosen,
286 "linux,pci-probe-only", NULL);
287 if (prop)
288 pci_probe_only = *prop;
289
290#ifdef CONFIG_PPC32 /* Will be made generic soon */
291 prop = of_get_property(of_chosen,
292 "linux,pci-assign-all-buses", NULL);
293 if (prop && *prop)
294 pci_add_flags(PCI_REASSIGN_ALL_BUS);
295#endif /* CONFIG_PPC32 */
296 }
```

也就是说如果OF中的属性设置pci-assign-all-buses则PCI core重新分配所有的PCI bus number，忽略firmware的设置。

在pass 0的时候，如果没有发生什么冲突，则调用pci_add_new_bus将这个pci_bus链入pci_bus的tree中。

```
598struct pci_bus *__ref pci_add_new_bus(struct pci_bus *parent, struct pci_dev *dev, int busnr)
599{
600 struct pci_bus *child;
601
602 child = pci_alloc_child_bus(parent, dev, busnr);
603 if (child) {
604 down_write(&pci_bus_sem);
605 list_add_tail(&child->node, &parent->children);
606 up_write(&pci_bus_sem);
607 }
608 return child;
609}
```

如果PCI_REASSIGN_ALL_BUS被设置，则pcibios_assign_all_busses()函数的值为1，在pass 0的时候，对已分配的不会执行if那段，而会执行else这段。这里会调用pci_write_config_dword把bus number等清零。也就是说如果PCI_REASSIGN_ALL_BUS这个标志被设置，遍历所有的bus的时候都只执行else这部分中if(!pass)这段，所以所有的bus无论是BIOS分配bus number号与否，都会清除configuration space中相应的值。而在pass 1的时候，也只会执行else这段，会为所有bus分配number，并写到configuration space，然后将pci_bus添加到pci_bus的tree中。对于PCI_REASSIGN_ALL_BUS为设置的情况，pass 0的时候执行if这段，这样就可以获得max bus number，但是这不是整个PCI tree的已被BIOS分配的number，而仅是这个sub tree的最大number。在pass 1的时候会执行else这段，从++max开始分配新的设备号。

```
699 } else {
700 /*
701 * We need to assign a number to this bus which we always
702 * do in the second pass.
703 */
704 if (!pass) {
705 if (pcibios_assign_all_busses() || broken)
706 /* Temporarily disable forwarding of the
707 configuration cycles on all bridges in
708 this bus segment to avoid possible
709 conflicts in the second pass between two
710 bridges programmed with overlapping
711 bus ranges. */
712 pci_write_config_dword(dev, PCI_PRIMARY_BUS,
713 buses & ~0xffffff);
714 goto out;
715 }
716
717 /* Clear errors */
718 pci_write_config_word(dev, PCI_STATUS, 0xffff);
719
720 /* Prevent assigning a bus number that already exists.
721 * This can happen when a bridge is hot-plugged, so in
722 * this case we only re-scan this bus. */
723 child = pci_find_bus(pci_domain_nr(bus), max+1);
724 if (!child) {
725 child = pci_add_new_bus(bus, dev, ++max);
726 if (!child)
727 goto out;
```

```
728 }
729 buses = (buses & 0xff000000)
730 | ((unsigned int)(child->primary) << 0)
731 | ((unsigned int)(child->secondary) << 8)
732 | ((unsigned int)(child->subordinate) << 16);
733
734 /*
735 * yenta.c forces a secondary latency timer of 176.
736 * Copy that behaviour here.
737 */
738 if (is_cardbus) {
739 buses &= ~0xff000000;
740 buses |= CARDBUS_LATENCY_TIMER << 24;
741 }
742
743 /*
744 * We need to blast all three values with a single write.
745 */
746 pci_write_config_dword(dev, PCI_PRIMARY_BUS, buses);
747
748 if (!is_cardbus) {
749 child->bridge_ctl = bctl;
750 /*
751 * Adjust subordinate busnr in parent buses.
752 * We do this before scanning for children because
753 * some devices may not be detected if the bios
754 * was lazy.
755 */
756 pci_fixup_parent_subordinate_busnr(child, max);
757 /* Now we can scan all subordinate buses... */
758 max = pci_scan_child_bus(child);
759 /*
760 * now fix it up again since we have found
761 * the real value of max.
762 */
763 pci_fixup_parent_subordinate_busnr(child, max);
764 } else {
765 。。。。。。。。。。。。。。。。。。。
795 }
796 /*
797 * Set the subordinate bus number to its real value.
798 */
799 child->subordinate = max;
800 pci_write_config_byte(dev, PCI_SUBORDINATE_BUS, max);
801 }
802
。。。。。。。。。。。。。。。。。。。。。。。。。
825
826out:
827 pci_write_config_word(dev, PCI_BRIDGE_CONTROL, bctl);
828
829 return max;
830}
```