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(四)

分类： linux内核编程 PCI-e/PCI 2012-02-25 19:16 356人阅读 评论(0) 收藏 举报

kernel hacker修炼之道之PCI subsystem(四)

作者 李万鹏

首先声明代码的版本是linux3.2.7，平台PowerPC_64

第一步分析PCI core对PCI device，PCI bus的遍历过程。PCI subsystem入口函数pcibios_init，在arch/powerpc/kernel/pci_64.c文件中：

```
static int __init pcibios_init(void)
{
struct pci_controller *hose, *tmp;

printk(KERN_INFO "PCI: Probing PCI hardware\n");

/* For now, override phys_mem_access_prot. If we need it,g
* later, we may move that initialization to each ppc_md
*/
ppc_md.phys_mem_access_prot = pci_phys_mem_access_prot;

if (pci_probe_only)
pci_add_flags(PCI_PROBE_ONLY);

/* On ppc64, we always enable PCI domains and we keep domain 0
* backward compatible in /proc for video cards
*/
pci_add_flags(PCI_ENABLE_PROC_DOMAINS | PCI_COMPAT_DOMAIN_0);

/* Scan all of the recorded PCI controllers. */
list_for_each_entry_safe(hose, tmp, &hose_list, list_node) {
pcibios_scan_phb(hose);
pci_bus_add_devices(hose->bus);
}

/* Call common code to handle resource allocation */
pcibios_resource_survey();

printk(KERN_DEBUG "PCI: Probing PCI hardware done\n");

return 0;
}

```

```
subsys_initcall(pcibios_init);
```

可以看到这个函数是被subsys_initcall调用的，查看其定义：

```
#define subsys_initcall(fn)  __define_initcall("4",fn,4)
#define __define_initcall(level,fn,id) \
        static initcall_t __initcall_##fn##id __used
        __attribute__((__section__(".initcall" level
```

在链接脚本vmlinux.lds中有这么一段：

```
__inicall_start = .;
.initcall.init : AT(ADDR(.initcall.init) – 0xC0000000) {
*(.initcall1.init)
*(.initcall2.init)
*(.initcall3.init)
*(.initcall4.init)
*(.initcall5.init)
*(.initcall6.init)
*(.initcall7.init)
}
__initcall_end = .;
```

所以，_define_initcall把pcibios_init这个函数指针放到了.initc
start_kernel()->rest_init()->kernel_init()->do_basic_set
pci_probe_only是PowerPC_64下的一个全局变量，如果这个全局变量为

```
void __devinit pcibios_scan_phb(struct pci_controller *hose)
{
struct pci_bus *bus;
struct device_node *node = hose->dn;
int mode;

pr_debug("PCI: Scanning PHB %s\n",
node ? node->full_name : "<NO NAME>");

/* Create an empty bus for the toplevel */
bus = pci_create_bus(hose->parent, hose->first_busno, hose->ops, hose);
if (bus == NULL) {
pr_err("Failed to create bus for PCI domain %04x\n",
hose->global_number);
return;
}
bus->secondary = hose->first_busno;
hose->bus = bus;

/* Get some IO space for the new PHB */
pcibios_setup_phb_io_space(hose);

/* Wire up PHB bus resources */
pcibios_setup_phb_resources(hose);

/* Get probe mode and perform scan */
mode = PCI_PROBE_NORMAL;
if (node && ppc_md.pci_probe_mode)
mode = ppc_md.pci_probe_mode(bus);
pr_debug(" probe mode: %d\n", mode);
if (mode == PCI_PROBE_DEVTREE) {
bus->subordinate = hose->last_busno;
of_scan_bus(node, bus);
}

if (mode == PCI_PROBE_NORMAL)
hose->last_busno = bus->subordinate = pci_scan_child_bus(bus);

/* Configure PCI Express settings */
if (bus && !pci_has_flag(PCI_PROBE_ONLY)) {
struct pci_bus *child;
list_for_each_entry(child, &bus->children, node) {
struct pci_dev *self = child->self;
if (!self)
continue;
pcie_bus_configure_settings(child, self->pcie_mpss);
}
}
}
```

首先调用函数pci_create_bus创建toplevel bus，也就是每个pci domain，phb直接引出的那个bus。这里把这条toplevel bus所在的phb的下游bus设置为hose->first_busno，其实就是这条bus的number。为新PHB分配一些空间，设置PHB。如果探测模式是PCI_PROBE_NORMAL，那么由PCI core进行遍历。调用pci_scan_child_bus函数扫描这个toplevel bus上的所有设备，以及之下的整个pci sub tree。注意这里的toplevel bus不一定就是0号bus，因为可能会有几个pci domain。

下面来看toplevel bus是如何创建的：

```
struct pci_bus * pci_create_bus(struct device *parent,
int bus, struct pci_ops *ops, void *sysdata)
{
int error;
struct pci_bus *b, *b2;
struct device *dev;

b = pci_alloc_bus();
if (!b)
return NULL;

dev = kzalloc(sizeof(*dev), GFP_KERNEL);
if (!dev){
kfree(b);
return NULL;
}

b->sysdata = sysdata;
b->ops = ops;

b2 = pci_find_bus(pci_domain_nr(b), bus);
if (b2) {
/* If we already got to this bus through a different bridge, ignore it */
dev_dbg(&b2->dev, "bus already known\n");
goto err_out;
}

down_write(&pci_bus_sem);
list_add_tail(&b->node, &pci_root_buses);
up_write(&pci_bus_sem);

dev->parent = parent;
dev->release = pci_release_bus_bridge_dev;
dev_set_name(dev, "pci%04x:%02x", pci_domain_nr(b), bus);
error = device_register(dev);
if (error)
goto dev_reg_err;
b->bridge = get_device(dev);
device_enable_async_suspend(b->bridge);
pci_set_bus_of_node(b);

if (!parent)
set_dev_node(b->bridge, pcibus_to_node(b));

b->dev.class = &pcibus_class;
b->dev.parent = b->bridge;
dev_set_name(&b->dev, "%04x:%02x", pci_domain_nr(b), bus);
error = device_register(&b->dev);
if (error)
goto class_dev_reg_err;

/* Create legacy_io and legacy_mem files for this bus */
pci_create_legacy_files(b);

b->number = b->secondary = bus;
b->resource[0] = &ioport_resource;
b->resource[1] = &iomem_resource;

return b;

class_dev_reg_err:
device_unregister(dev);
dev_reg_err:
down_write(&pci_bus_sem);
list_del(&b->node);
up_write(&pci_bus_sem);
err_out:
kfree(dev);
kfree(b);
return NULL;
}
```

这里的那个bus 是hose->first_busno传过来的，也就是说，已经由firmware分配好的。这里调用pci_find_bus查看一下，如果这个bus已经从不同的bridge分配过了，那么忽略它。否则将这个刚建好的pci_bus添加到pci_root_buses链表上，这个链表是所有toplevel bus的天堂。总线号和总线所在的桥的下游的总线号都是hose->first_buso。toplevel bus的两个指针分别指向了ioport_resousrce和iomem_resource。

PCI core会使用Depth first的方式遍历整个子树：

```
unsigned int __devinit pci_scan_child_bus(struct pci_bus *bus)
{
unsigned int devfn, pass, max = bus->secondary;
struct pci_dev *dev;

dev_dbg(&bus->dev, "scanning bus\n");

/* Go find them, Rover! */
for (devfn = 0; devfn < 0x100; devfn += 8)
pci_scan_slot(bus, devfn);

/* Reserve buses for SR-IOV capability. */
max += pci_iov_bus_range(bus);

/*
* After performing arch-dependent fixup of the bus, look behind
* all PCI-to-PCI bridges on this bus.
*/
if (!bus->is_added) {
dev_dbg(&bus->dev, "fixups for bus\n");
pcibios_fixup_bus(bus);
if (pci_is_root_bus(bus))
bus->is_added = 1;
}

for (pass=0; pass < 2; pass++)
list_for_each_entry(dev, &bus->devices, bus_list) {
if (dev->hdr_type == PCI_HEADER_TYPE_BRIDGE ||
dev->hdr_type == PCI_HEADER_TYPE_CARDBUS)
max = pci_scan_bridge(bus, dev, max, pass);
}

/*
* We've scanned the bus and so we know all about what's on
* the other side of any bridges that may be on this bus plus
* any devices.
*
* Return how far we've got finding sub-buses.
*/
dev_dbg(&bus->dev, "bus scan returning with max=%02x\n", max);
return max;
}
```

由于一条总线上可以有32个slot，256个logical device，所以这里先扫描这个bus上的所有logical device，如果读取他们的configuration space，填充pci_dev的各个域。然后遍历这个bus上的所有桥，遍历两遍：

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；

这里的具体细节会在下一篇blog中介绍，下面来看负责扫描slot的函数pci_scan_slot：

```
int pci_scan_slot(struct pci_bus *bus, int devfn)
{
unsigned fn, nr = 0;
struct pci_dev *dev;
unsigned (*next_fn)(struct pci_dev *, unsigned) = no_next_fn;

if (only_one_child(bus) && (devfn > 0))
return 0; /* Already scanned the entire slot */

dev = pci_scan_single_device(bus, devfn);
if (!dev)
return 0;
if (!dev->is_added)
nr++;

if (pci_ari_enabled(bus))
next_fn = next_ari_fn;
else if (dev->multifunction)
next_fn = next_trad_fn;

for (fn = next_fn(dev, 0); fn > 0; fn = next_fn(dev, fn)) {
dev = pci_scan_single_device(bus, devfn + fn);
if (dev) {
if (!dev->is_added)
nr++;
dev->multifunction = 1;
}
}

/* only one slot has pcie device */
if (bus->self && nr)
pcie_aspm_init_link_state(bus->self);

return nr;
}
```

这里调用pci_scan_single_device扫描单个设备：

```
struct pci_dev *__ref pci_scan_single_device(struct pci_bus *bus, int devfn)
{
struct pci_dev *dev;

dev = pci_get_slot(bus, devfn);
if (dev) {
pci_dev_put(dev);
return dev;
}

dev = pci_scan_device(bus, devfn);
if (!dev)
return NULL;

pci_device_add(dev, bus);

return dev;
}
```

通过bus和device:function号获得pci_dev，然后扫描这个logical device，主要就是获得厂商ID和设备ID，然后通过device的bus_list bit field把pci_dev添加到bus的devices链表上。

```
static struct pci_dev *pci_scan_device(struct pci_bus *bus, int devfn)
{
struct pci_dev *dev;
u32 l;
int delay = 1;

if (pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, &l))
return NULL;

/* some broken boards return 0 or ~0 if a slot is empty: */
if (l == 0xffffffff || l == 0x00000000 ||
l == 0x0000ffff || l == 0xffff0000)
return NULL;

/* Configuration request Retry Status */
while (l == 0xffff0001) {
msleep(delay);
delay *= 2;
if (pci_bus_read_config_dword(bus, devfn, PCI_VENDOR_ID, &l))
return NULL;
/* Card hasn't responded in 60 seconds? Must be stuck. */
if (delay > 60 * 1000) {
printk(KERN_WARNING "pci %04x:%02x:%02x.%d: not "
"responding\n", pci_domain_nr(bus),
bus->number, PCI_SLOT(devfn),
PCI_FUNC(devfn));
return NULL;
}
}

dev = alloc_pci_dev();
if (!dev)
return NULL;

dev->bus = bus;
dev->devfn = devfn;
dev->vendor = l & 0xffff;
dev->device = (l >> 16) & 0xffff;

pci_set_of_node(dev);

if (pci_setup_device(dev)) {
kfree(dev);
return NULL;
}

return dev;
}
```

只要不是全1或全0就算有效的厂商ID，设备ID，如果读出的是0xffff0001，则重复读，如果超过60s，不再尝试。否则调用pci_setup_device函数，这个函数主要读取configuration space，然后设置device的相应域。

```
int pci_setup_device(struct pci_dev *dev)
{
u32 class;
u8 hdr_type;
struct pci_slot *slot;
int pos = 0;

if (pci_read_config_byte(dev, PCI_HEADER_TYPE, &hdr_type))
return -EIO;

dev->sysdata = dev->bus->sysdata;
dev->dev.parent = dev->bus->bridge;
dev->dev.bus = &pci_bus_type;
dev->hdr_type = hdr_type & 0x7f;
dev->multifunction = !!(hdr_type & 0x80);
dev->error_state = pci_channel_io_normal;
set_pcie_port_type(dev);

list_for_each_entry(slot, &dev->bus->slots, list)
if (PCI_SLOT(dev->devfn) == slot->number)
dev->slot = slot;

/* Assume 32-bit PCI; let 64-bit PCI cards (which are far rarer)
set this higher, assuming the system even supports it. */
dev->dma_mask = 0xffffffff;

dev_set_name(&dev->dev, "%04x:%02x:%02x.%d", pci_domain_nr(dev->bus),
dev->bus->number, PCI_SLOT(dev->devfn),
PCI_FUNC(dev->devfn));

pci_read_config_dword(dev, PCI_CLASS_REVISION, &class);
dev->revision = class & 0xff;
```

```
class >>= 8; /* upper 3 bytes */
dev->class = class;
class >>= 8;

dev_printk(KERN_DEBUG, &dev->dev, "[%04x:%04x] type %d class %#08x\n",
dev->vendor, dev->device, dev->hdr_type, class);

/* need to have dev->class ready */
dev->cfg_size = pci_cfg_space_size(dev);

/* "Unknown power state" */
dev->current_state = PCI_UNKNOWN;

/* Early fixups, before probing the BARs */
pci_fixup_device(pci_fixup_early, dev);
/* device class may be changed after fixup */
class = dev->class >> 8;

switch (dev->hdr_type) { /* header type */
case PCI_HEADER_TYPE_NORMAL: /* standard header */
if (class == PCI_CLASS_BRIDGE_PCI)
goto bad;
pci_read_irq(dev);
pci_read_bases(dev, 6, PCI_ROM_ADDRESS);
pci_read_config_word(dev, PCI_SUBSYSTEM_VENDOR_ID, &dev->subsystem_vendor);
pci_read_config_word(dev, PCI_SUBSYSTEM_ID, &dev->subsystem_device);

/*
* Do the ugly legacy mode stuff here rather than broken chip
* quirk code. Legacy mode ATA controllers have fixed
* addresses. These are not always echoed in BAR0-3, and
* BAR0-3 in a few cases contain junk!
*/
if (class == PCI_CLASS_STORAGE_IDE) {
u8 progif;
pci_read_config_byte(dev, PCI_CLASS_PROG, &progif);
if ((progif & 1) == 0) {
dev->resource[0].start = 0x1F0;
dev->resource[0].end = 0x1F7;
dev->resource[0].flags = LEGACY_IO_RESOURCE;
dev->resource[1].start = 0x3F6;
dev->resource[1].end = 0x3F6;
dev->resource[1].flags = LEGACY_IO_RESOURCE;
}
if ((progif & 4) == 0) {
dev->resource[2].start = 0x170;
dev->resource[2].end = 0x177;
dev->resource[2].flags = LEGACY_IO_RESOURCE;
dev->resource[3].start = 0x376;
dev->resource[3].end = 0x376;
dev->resource[3].flags = LEGACY_IO_RESOURCE;
}
}
break;

case PCI_HEADER_TYPE_BRIDGE: /* bridge header */
if (class != PCI_CLASS_BRIDGE_PCI)
goto bad;
/* The PCI-to-PCI bridge spec requires that subtractive
decoding (i.e. transparent) bridge must have programming
interface code of 0x01. */
pci_read_irq(dev);
dev->transparent = ((dev->class & 0xff) == 1);
pci_read_bases(dev, 2, PCI_ROM_ADDRESS1);
set_pcie_hotplug_bridge(dev);
pos = pci_find_capability(dev, PCI_CAP_ID_SSVID);
if (pos) {
pci_read_config_word(dev, pos + PCI_SSVID_VENDOR_ID, &dev->subsystem_vendor);
pci_read_config_word(dev, pos + PCI_SSVID_DEVICE_ID, &dev->subsystem_device);
}
break;

case PCI_HEADER_TYPE_CARDBUS: /* CardBus bridge header */
if (class != PCI_CLASS_BRIDGE_CARDBUS)
goto bad;
pci_read_irq(dev);
pci_read_bases(dev, 1, 0);
pci_read_config_word(dev, PCI_CB_SUBSYSTEM_VENDOR_ID, &dev->subsystem_vendor);
pci_read_config_word(dev, PCI_CB_SUBSYSTEM_ID, &dev->subsystem_device);
break;

default: /* unknown header */
dev_err(&dev->dev, "unknown header type %02x, "
"ignoring device\n", dev->hdr_type);
return -EIO;

bad:
dev_err(&dev->dev, "ignoring class %02x (doesn't match header "
"type %02x)\n", class, dev->hdr_type);
dev->class = PCI_CLASS_NOT_DEFINED;
}

/* We found a fine healthy device, go go go... */
return 0;
}
```

这里注意Header Type Register是8位的，低7位指明header type，最高1位指明这是多功能设备还是单功能设备，当最高位为1时是多功能设备，当最高位为0时是单功能设备。所以这里dev->hdr_type=hdr_type & 0x7f获得低7位，dev->multifunction=!!(hdr_type & 08)获得最高位。目前header type头主要有三种，可以从include/linux/pci_regs.h：

```
#define PCI_HEADER_TYPE_NORMAL 0
#define PCI_HEADER_TYPE_BRIDGE 1
#define PCI_HEADER_TYPE_CARDBUS 2
```

Revision ID register是8位的寄存器，Class Code Register是24位的寄存器，这里一次读取了长字，其中最低的8位是revision id，高24位是class code，这24位又分成3部分，class：sub-class：programming interface。假如读出的数是0x060401，那么06表示class(bridge device)，04表示sub-class(PCI-to-PCI bridge)，01是programming interface。其中head type 0 对应normal device，head type 1对应pci-to-pci bridge。注意如果这个不匹配回答因警告信息：

```
dev_err(&dev->dev, "ignoring class %02x (doesn't match header "
"type %02x)\n", class, dev->hdr_type);
```

跟桥有关的类和子类如下，这些都是在文件include/linux/pci_ids.h中定义的：

```
#define PCI_BASE_CLASS_BRIDGE 0x06
#define PCI_CLASS_BRIDGE_HOST 0x0600
#define PCI_CLASS_BRIDGE_ISA 0x0601
#define PCI_CLASS_BRIDGE_EISA 0x0602
#define PCI_CLASS_BRIDGE_MC 0x0603
#define PCI_CLASS_BRIDGE_PCI 0x0604
#define PCI_CLASS_BRIDGE_PCMCIA 0x0605
#define PCI_CLASS_BRIDGE_NUBUS 0x0606
#define PCI_CLASS_BRIDGE_CARDBUS 0x0607
#define PCI_CLASS_BRIDGE_RACEWAY 0x0608
#define PCI_CLASS_BRIDGE_OTHER 0x0680
```

注意上边的type1与class: pci-to-pci bridge

匹配，与其他的不匹配，不要看错了。