邮箱：zhunxun@gmail.com

< 2020年4月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
3	4	5	6	7	8	9

搜索

找找看

谷歌搜索

PostCategories

C语言(2)
IO Virtualization(3)
KVM虚拟化技术(26)
linux 内核源码分析(61)
Linux日常应用(3)
linux时间子系统(3)
qemu(10)
seLinux(1)
windows内核(5)
调试技巧(2)
内存管理(8)
日常技能(3)
容器技术(2)
生活杂谈(1)
网络(5)
文件系统(4)
硬件(4)

PostArchives

2018/4(1)
2018/2(1)
2018/1(3)
2017/12(2)
2017/11(4)
2017/9(3)
2017/8(1)
2017/7(8)
2017/6(6)
2017/5(9)
2017/4(15)
2017/3(5)
2017/2(1)
2016/12(1)
2016/11(11)
2016/10(8)
2016/9(13)

ArticleCategories

时态分析(1)

Recent Comments

1. Re:virtio前端驱动详解
我看了下，Linux-4.18.2中的vp_notify()函数。bool vp_notify(struct virtqueue *vq){ /* we write the queue's selec...
--Linux-inside
2. Re:virtIO之VHOST工作原理简析

PCI 设备详解二

上篇文章主要从硬件的角度分析了PCI设备的特性以及各种寄存器，那么本节就结合LInux源代码分析下内核中PCI设备的各种数据结构以及相互之间的联系和工作机制

2016-10-09

注：一下代码参考LInux3.11.1内核

基本的数据结构：

struct pci_bus

```
struct pci_bus {
    struct list_head node;        /* node in list of buses */
    struct pci_bus    *parent;    /* parent bus this bridge is on */
    struct list_head children;    /* list of child buses */
    struct list_head devices;    /* list of devices on this bus */
    struct pci_dev    *self;        /* bridge device as seen by parent */
    struct list_head slots;        /* list of slots on this bus */
    struct resource *resource[PCI_BRIDGE_RESOURCE_NUM];
    struct list_head resources;    /* address space routed to this bus */
    struct resource busn_res;    /* bus numbers routed to this bus */

    struct pci_ops    *ops;        /* configuration access functions */
    void        *sysdata;    /* hook for sys-specific extension */
    struct proc_dir_entry *procdir;    /* directory entry in /proc/bus/pci */

    unsigned char    number;        /* bus number */
    unsigned char    primary;    /* number of primary bridge */
    unsigned char    max_bus_speed;    /* enum pci_bus_speed */
    unsigned char    cur_bus_speed;    /* enum pci_bus_speed */

    char        name[48];

    unsigned short  bridge_ctl;    /* manage NO_ISA/FBB/et al behaviors */
    pci_bus_flags_t bus_flags;    /* Inherited by child busses */
    struct device        *bridge;
    struct device        dev;
    struct bin_attribute    *legacy_io; /* legacy I/O for this bus */
    struct bin_attribute    *legacy_mem; /* legacy mem */
    unsigned int        is_added:1;
};
```

每个PCI总线都有一个pci_bus结构与之对应。

系统中所有的根总线的pci_bus结构通过node连接起来；

parent指向该总线的父总线即上一级总线；

children描述这条PCI总线的子总线链表的表头；

devices描述了这条PCI总线的逻辑设备链表的表头；

self指向引出这条PCI总线的桥设备的pci_dev结构；

ops指向一个结构描述访问配置空间的方法；

number描述总线号；

primary描述桥设备所在总线；

struct dev

再问一个问题，从设置ioeventfd那个流程来看的话是guest发起一个IO，首先会陷入到kvm中，然后由kvm向qemu发送一个IO到来的event，最后IO才被处理，是这样的吗?

--Linux-inside

3. Re:virtIO之VHOST工作原理简析

你好。设置ioeventfd这个部分和guest里面的virtio前端驱动有关系吗?
设置ioeventfd和virtio前端驱动是如何发生联系起来的？谢谢。

--Linux-inside

4. Re:QEMU IO事件处理框架

良心博主，怎么停跟了，太可惜了。

--黄铁牛

5. Re:linux 逆向映射机制浅析

小哥哥520脱单了么

--黄铁牛

Top Posts

1. 详解操作系统中断(21043)
2. PCI 设备详解一(15716)
3. 进程的挂起、阻塞和睡眠(13593)
4. Linux下桥接模式详解一(13384)
5. virtio后端驱动详解(10500)

推荐排行榜

1. 进程的挂起、阻塞和睡眠(6)
2. qemu-kvm内存虚拟化1(2)
3. 为何要写博客(2)
4. virtIO前后端notify机制详解(2)
5. 详解操作系统中断(2)

```
struct pci_dev {
    struct list_head bus_list;    /* node in per-bus list对应总线下所有的设备链表 */
    struct pci_bus    *bus;        /* bus this device is on   设备所在的bus*/
    struct pci_bus    *subordinate;    /* bus this device bridges to 当作为PCI桥设备时，指向下级bus */

    void        *sysdata;    /* hook for sys-specific extension */
    struct proc_dir_entry *procent;    /* device entry in /proc/bus/pci */
    struct pci_slot    *slot;        /* Physical slot this device is in */

    unsigned int    devfn;        /* encoded device & function index 设备号和功能号*/
    unsigned short    vendor;
    unsigned short    device;
    unsigned short    subsystem_vendor;
    unsigned short    subsystem_device;
    unsigned int    class;        /* 3 bytes: (base,sub,prog-if) */
    u8        revision;    /* PCI revision, low byte of class word */
    u8        hdr_type;    /* PCI header type (`multi' flag masked out) */
    u8        pcie_cap;    /* PCI-E capability offset */
    u8        msi_cap;    /* MSI capability offset */
    u8        msix_cap;    /* MSI-X capability offset */
    u8        pcie_mpss:3;    /* PCI-E Max Payload Size Supported */
    u8        rom_base_reg;    /* which config register controls the ROM */
    u8        pin;           /* which interrupt pin this device uses */
    u16        pcie_flags_reg;    /* cached PCI-E Capabilities Register */

    struct pci_driver *driver;    /* which driver has allocated this device */
    u64        dma_mask;    /* Mask of the bits of bus address this
                    device implements.  Normally this is
                    0xffffffff.  You only need to change
                    this if your device has broken DMA
                    or supports 64-bit transfers.  */

    struct device_dma_parameters dma_parms;

    pci_power_t     current_state;  /* Current operating state. In ACPI-speak,
                       this is D0-D3, D0 being fully functional,
                       and D3 being off. */
    u8        pm_cap;        /* PM capability offset */
    unsigned int    pme_support:5;    /* Bitmask of states from which PME#
                       can be generated */
    unsigned int    pme_interrupt:1;
    unsigned int    pme_poll:1;    /* Poll device's PME status bit */
    unsigned int    d1_support:1;    /* Low power state D1 is supported */
    unsigned int    d2_support:1;    /* Low power state D2 is supported */
    unsigned int    no_d1d2:1;    /* D1 and D2 are forbidden */
    unsigned int    no_d3cold:1;    /* D3cold is forbidden */
    unsigned int    d3cold_allowed:1;    /* D3cold is allowed by user */
    unsigned int    mmio_always_on:1;    /* disallow turning off io/mem
                           decoding during bar sizing */
    unsigned int    wakeup_prepared:1;
    unsigned int    runtime_d3cold:1;    /* whether go through runtime
                           D3cold, not set for devices
                           powered on/off by the
                           corresponding bridge */
    unsigned int    d3_delay;    /* D3->D0 transition time in ms */
    unsigned int    d3cold_delay;    /* D3cold->D0 transition time in ms */

#ifdef CONFIG_PCIEASPM
    struct pcie_link_state    *link_state;    /* ASPM link state. */
#endif

    pci_channel_state_t error_state;    /* current connectivity state */
    struct    device    dev;        /* Generic device interface */

    int        cfg_size;    /* Size of configuration space */

    /*
     * Instead of touching interrupt line and base address registers
     * directly, use the values stored here. They might be different!
     */
    unsigned int    irq;
    struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O and memory regions + expansion ROMs */

    bool match_driver;        /* Skip attaching driver */
    /* These fields are used by common fixups */
```

```
    unsigned int    transparent:1;    /* Transparent PCI bridge */
    unsigned int    multifunction:1;/* Part of multi-function device */
    /* keep track of device state */
    unsigned int    is_added:1;
    unsigned int    is_busmaster:1; /* device is busmaster */
    unsigned int    no_msi:1;    /* device may not use msi */
    unsigned int    block_cfg_access:1;    /* config space access is blocked */
    unsigned int    broken_parity_status:1;    /* Device generates false positive parity
*/
    unsigned int    irq_reroute_variant:2;    /* device needs IRQ rerouting variant */
    unsigned int     msi_enabled:1;
    unsigned int    msix_enabled:1;
    unsigned int    ari_enabled:1;    /* ARI forwarding */
    unsigned int    is_managed:1;
    unsigned int    is_pcie:1;    /* Obsolete. Will be removed.
                       Use pci_is_pcie() instead */
    unsigned int    needs_freset:1; /* Dev requires fundamental reset */
    unsigned int    state_saved:1;
    unsigned int    is_physfn:1;
    unsigned int    is_virtfn:1;
    unsigned int    reset_fn:1;
    unsigned int    is_hotplug_bridge:1;
    unsigned int    __aer_firmware_first_valid:1;
    unsigned int    __aer_firmware_first:1;
    unsigned int    broken_intx_masking:1;
    unsigned int    io_window_1k:1;    /* Intel P2P bridge 1K I/O windows */
    pci_dev_flags_t dev_flags;
    atomic_t    enable_cnt;    /* pci_enable_device has been called */

    u32        saved_config_space[16]; /* config space saved at suspend time */
    struct hlist_head saved_cap_space;
    struct bin_attribute *rom_attr; /* attribute descriptor for sysfs ROM entry */
    int rom_attr_enabled;        /* has display of the rom attribute been enabled? */
    struct bin_attribute *res_attr[DEVICE_COUNT_RESOURCE]; /* sysfs file for resources */
    struct bin_attribute *res_attr_wc[DEVICE_COUNT_RESOURCE]; /* sysfs file for WC
mapping of resources */
#ifdef CONFIG_PCI_MSI
    struct list_head msi_list;
    struct kset *msi_kset;
#endif
    struct pci_vpd *vpd;
#ifdef CONFIG_PCI_ATS
    union {
        struct pci_sriov *sriov;    /* SR-IOV capability related */
        struct pci_dev *physfn;    /* the PF this VF is associated with */
    };
    struct pci_ats    *ats;    /* Address Translation Service */
#endif
    phys_addr_t rom; /* Physical address of ROM if it's not from the BAR */
    size_t romlen; /* Length of ROM if it's not from the BAR */
};
```

PCI 总线上的每一个逻辑设备都会有一个这样的结构。

struct resource

```
struct resource {
    resource_size_t start;
    resource_size_t end;
    const char *name;
    unsigned long flags;
    struct resource *parent, *sibling, *child;
};
```

系统用resource结构管理地址空间资源，因为这里主要的目的是映射地址空间（PIO或者MMIO）到设备，所以地址空间可以说是一种资源，每段地址空间用一个resource结构来描述。该结构中有地址空间的起始和结束地址，name，和一些位；系统维护了两个全局的resource链表，一个用于管理IO端口空间，一个用于管理IO内存空间。从resource结构可以看出，这里是通过树来组织这些结构。这里有些类似于windows中虚拟内存管理方式，通过这种方式可以比较高效的分配一段未使用的区间或者查找一段区间是否有重叠!

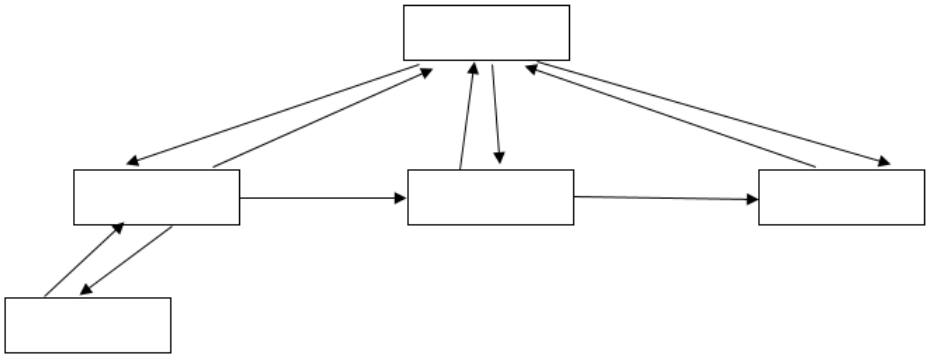
start指向区间的起点

end指向区间的重点

name是区间类型名

flag记录区间的一些标志位

parent指向父res,child指向第一个子res,同一父res的子res通过sibling连接起来，整体结构如下图：



额~~~仔细一数算，涉及到的结构还真不多，那么就不在这耽搁时间了，看LInux内核中PCI设备探测和初始化的过程

PCI设备的探测有多种方式，大体上分为BIOS探测和直接探测。直接探测又分为两种类型。一般而言只要是采用PCI总线的PC机，其BIOS都必须提供对PCI总线的支持，因而成为PCI BIOS。这种BIOS在机器加电之后的自检阶段会从系统中的第一个PCI桥即host-PCI 桥开始进行探测和扫描，逐个的枚举连接在第一条PCI总线上的所有PCI设备并记录。如果其中有个设备是PCI-PCI桥，则更进一步，在探测这个桥所连接总线上的设备，依次递归，知道穷尽所有的PCI设备。但是并不是所有的系统都有BIOS，后来LInux内核也就提供了一种直接探测PCI设备的方式，绕过了BIOS的探测。

PCI设备的初始化在内核中我们分为两个入口：

1、arch_initcall(pci_arch_init);

2、subsys_initcall(pci_subsys_init);

arch_initcall的优先级要高于subsys_initcall,所以**pci_arch_init函数**会在函数**pci_subsys_init**之前执行。这里我们的分析也从两个部分

1、pci_arch_init

```
static __init int pci_arch_init(void)
{
#ifdef CONFIG_PCI_DIRECT
    int type = 0;

    type = pci_direct_probe();
#endif

    if (!(pci_probe & PCI_PROBE_NOEARLY))
        pci_mmcfg_early_init();

    if (x86_init.pci.arch_init && !x86_init.pci.arch_init())
        return 0;

#ifdef CONFIG_PCI_BIOS
    pci_pcbios_init();
#endif
    /*
     * don't check for raw_pci_ops here because we want pcbios as last
     * fallback, yet it's needed to run first to set pcibios_last_bus
     * in case legacy PCI probing is used. otherwise detecting peer busses
     * fails.
     */
#ifdef CONFIG_PCI_DIRECT
    pci_direct_init(type);
#endif
    if (!raw_pci_ops && !raw_pci_ext_ops)
        printk(KERN_ERR
        "PCI: Fatal: No config space access function found\n");

    dmi_check_pciprobe();

    dmi_check_skip_isa_align();
```

```
    return 0;
}
```

该函数完成的主要是与体系结构相关的。函数中根据不同的探索方式有不同的选项，这里我们只关注**CONFIG_PCI_DIRECT**，可以看到这里仅仅是调用了一个**pci_direct_probe**函数

```
int __init pci_direct_probe(void)
{
    if ((pci_probe & PCI_PROBE_CONF1) == 0)
        goto type2;
    /*在IO地址空间从0xCF8开始申请八个字节用于用于配置PCI*/
    if (!request_region(0xCF8, 8, "PCI conf1"))
        goto type2;

    if (pci_check_type1()) {
        raw_pci_ops = &pci_direct_conf1;
        port_cf9_safe = true;
        return 1;
    }
    release_region(0xCF8, 8);

 type2:
    ...
    }

    release_region(0xC000, 0x1000);
 fail2:
    release_region(0xCF8, 4);
    return 0;
}
```

该函数完成的功能比较简单，就是从IO地址空间申请8个字节用作访问PCI配置空间的IO端口0xCF8~0xCFF。前四个字节做地址端口，后四个字节做数据端口。

然后调用**pci_check_type1**函数进一步检查，这主要是对1型配置方式做检查：

```
 1 static int __init pci_check_type1(void)
 2 {
 3     unsigned long flags;
 4     unsigned int tmp;
 5     int works = 0;
 6
 7     local_irq_save(flags);
 8
 9     outb(0x01, 0xCFB);
10     tmp = inl(0xCF8);
11     outl(0x80000000, 0xCF8);
12     if (inl(0xCF8) == 0x80000000 && pci_sanity_check(&pci_direct_conf1)) {
13         works = 1;
14     }
15     outl(tmp, 0xCF8);
16     local_irq_restore(flags);
17
18     return works;
19 }
```

这里代码只有短短几行，但是着实不容易理解，首先向0xCFB写入了0x01,然后保存0xCF8 四个字节的内容，接着向地址端口0xCF8写入0x800000000,然后读数据端口。回想下前面提到的PCI总线地址的格式，最高位始终为1，那么这里就意味着要读0总线0设备0功能0寄存器的值。但是下面貌似并没有读取数据端口，而是又读了下地址端口。这我就不太明白了，**难道这里只是检测下端口是否正常？？**

接着就继续调用了pci_sanity_check函数确保这种访问模式可以正确执行。这里是测试HOST-bridge是否存在

```
static int __init pci_sanity_check(const struct pci_raw_ops *o)
{
    u32 x = 0;
    int year, devfn;

    if (pci_probe & PCI_NO_CHECKS)
        return 1;
    /* Assume Type 1 works for newer systems.
       This handles machines that don't have anything on PCI Bus 0. */
    dmi_get_date(DMI_BIOS_DATE, &year, NULL, NULL);
    if (year >= 2001)
        return 1;

    for (devfn = 0; devfn < 0x100; devfn++) {
        /*读取设备的类型*/
        if (o->read(0, 0, devfn, PCI_CLASS_DEVICE, 2, &x))
            continue;
        if (x == PCI_CLASS_BRIDGE_HOST || x == PCI_CLASS_DISPLAY_VGA)
            return 1;
        /*读取设备厂商*/
        if (o->read(0, 0, devfn, PCI_VENDOR_ID, 2, &x))
            continue;
        if (x == PCI_VENDOR_ID_INTEL || x == PCI_VENDOR_ID_COMPAQ)
            return 1;
    }

    DBG(KERN_WARNING "PCI: Sanity check failed\n");
    return 0;
}
```

这样prob阶段的工作就完成了。回到pci_arch_init函数中，接下来调用了**pci_direct_init**函数，

```
void __init pci_direct_init(int type)
{
    if (type == 0)
        return;
    printk(KERN_INFO "PCI: Using configuration type %d for base access\n",
          type);
    if (type == 1) {
        raw_pci_ops = &pci_direct_conf1;
        if (raw_pci_ext_ops)
            return;
        if (!(pci_probe & PCI_HAS_IO_ECS))
            return;
        printk(KERN_INFO "PCI: Using configuration type 1 "
               "for extended access\n");
        raw_pci_ext_ops = &pci_direct_conf1;
        return;
    }
    raw_pci_ops = &pci_direct_conf2;
}
```

该函数就比较短小了，在前面pci_direct_probe函数的时候已经返回1，那么这里实际上就是设置raw_pci_ops=&pci_direct_config，从而设置正确的的读取PCI设备配置空间的方法。到此为止，一阶段已经结束了，看下面第二部分

2、pci_subsys_init

```
int __init pci_subsys_init(void)
{
    /*
     * The init function returns an non zero value when
     * pci_legacy_init should be invoked.
     */
     /*这pci.init可能会被初始化成两个函数pci_legacy_init和pci_acpi_init 猜想这里是要确保
pci_legacy_init得到执行*/
    if (x86_init.pci.init())
```

```
        pci_legacy_init();

    pcibios_fixup_peer_bridges();
    x86_init.pci.init_irq();
    pcibios_init();
    return 0;
}
```

在Pci_x86.c文件中，有这么一段初始化的宏：

```
# ifdef CONFIG_ACPI
#  define x86_default_pci_init        pci_acpi_init
# else
#  define x86_default_pci_init        pci_legacy_init
```

这里就是如果配置了ACPI的话，就初始化成**pci_acpi_init**，否则就初始化成**pci_legacy_init**，这里我们先只考虑没有配置ACPI的情况。

```
int __init pci_legacy_init(void)
{
    if (!raw_pci_ops) {
        printk("PCI: System does not support PCI\n");
        return 0;
    }
    printk("PCI: Probing PCI hardware\n");
    pcibios_scan_root(0);
    return 0;
}
```

在第一部分，已经设置了raw_pci_ops，所以这里正常情况就略过，执行**pcibios_scan_root**

关键函数在于**pcibios_scan_root**，分析pci_find_next_bus的代码就可以知道，在发现root总线之前，这里是返回NULL的，需要通过**pci_scan_bus_on_node**来探测root总线。

```
 1 struct pci_bus *pcibios_scan_root(int busnum)
 2 {
 3     struct pci_bus *bus = NULL;
 4     /*找到0号总线后，后续就比较容易了,这里直接遍历总线链表，若已经存在对应总线号的总线，则说明已经探测到了，就直接返回*/
 5     while ((bus = pci_find_next_bus(bus)) != NULL) {
 6         if (bus->number == busnum) {
 7             /* Already scanned */
 8             return bus;
 9         }
10     }
11     /*否则还需要根据总线号寻找总线并返回*/
12     return pci_scan_bus_on_node(busnum, &pci_root_ops,
13                     get_mp_bus_to_node(busnum));
14 }
```

我们看下**pci_scan_bus_on_node**函数做了什么

```
 1 struct pci_bus *pci_scan_bus_on_node(int busno, struct pci_ops *ops, int node)
 2 {
 3     LIST_HEAD(resources);
 4     struct pci_bus *bus = NULL;
 5     struct pci_sysdata *sd;
 6
 7     /*
 8      * Allocate per-root-bus (not per bus) arch-specific data.
 9      * TODO: leak; this memory is never freed.
10      * It's arguable whether it's worth the trouble to care.
11      */
12     sd = kzalloc(sizeof(*sd), GFP_KERNEL);
13     if (!sd) {
14         printk(KERN_ERR "PCI: OOM, skipping PCI bus %02x\n", busno);
15         return NULL;
```

```
    }
    sd->node = node;
    /*给总线分配资源*/
    x86_pci_root_bus_resources(busno, &resources);
    printk(KERN_DEBUG "PCI: Probing PCI hardware (bus %02x)\n", busno);
    bus = pci_scan_root_bus(NULL, busno, ops, sd, &resources);
    if (!bus) {
        pci_free_resource_list(&resources);
        kfree(sd);
    }
    return bus;
}
```

其实这里关于**sysdata**的问题，我也不是很清楚，貌似和CPU架构相关，这里我们不重点考虑，后续晓得了 再补充。我们可以看到这里调用了**x86_pci_root_bus_resource**函数给总线分配资源，这里需要说下就是一条总线上的设备的资源也是从总线的资源里面分配的，就是我们所说的窗口，设备或者桥的窗口一定是对应总线窗口的子窗口，如果可能后面会专门拿出一节分析资源的分配。

接着就调用**pci_scan_root_bus**函数探测根总线。

```
void x86_pci_root_bus_resources(int bus, struct list_head *resources)
{
    struct pci_root_info *info = x86_find_pci_root_info(bus);
    struct pci_root_res *root_res;
    struct pci_host_bridge_window *window;
    bool found = false;

    if (!info)
        goto default_resources;

    printk(KERN_DEBUG "PCI: root bus %02x: hardware-probed resources\n",
           bus);

    /* already added by acpi ? */
    /*resource 是记录windows的链表*/
    /*这里判断是否已经分配了总线号资源*/
    list_for_each_entry(window, resources, list)
        if (window->res->flags & IORESOURCE_BUS) {
            found = true;
            break;
        }
    /*如果没有分配需要重新分配*/
    if (!found)
        /*相当于向系统注册总线号资源*/
        pci_add_resource(resources, &info->busn);

    list_for_each_entry(root_res, &info->resources, list) {
        struct resource *res;
        struct resource *root;
        res = &root_res->res;
        /*向系统注册地址空间资源*/
        pci_add_resource(resources, res);
        if (res->flags & IORESOURCE_IO)
            root = &ioport_resource;
        else
            root = &iomem_resource;
        /*把res插入资源树*/
        insert_resource(root, res);
    }
    return;

default_resources:
    /*
     * We don't have any host bridge aperture information from the
     * "native host bridge drivers," e.g., amd_bus or broadcom_bus,
     * so fall back to the defaults historically used by pci_create_bus().
     */
    printk(KERN_DEBUG "PCI: root bus %02x: using default resources\n", bus);
    pci_add_resource(resources, &ioport_resource);
    pci_add_resource(resources, &iomem_resource);
}
```

这里首先判断总线的总线号资源分配情况，因为总线号也是一种资源，且由于总线遍历是按照深度优先遍历，一条总线上的所有子总线可以看成一棵树，并且这些总线号是连续的，所以可以用区间来表示，这就和IO资源、MEM资源类似了，所以这里总线号资源也是通过resource结构来表示的。一个host-bridge有一个全局的资源链表resources,连接了所有pci_host_bridge_window，总线号资源、IO端口资源、IO内存资源在里面都有体现.为什么这么说呢？？因为不论是那种资源都都会有一个pci_host_bridge_window与之对应，注意是一个区间就有一个pci_host_bridge_window结构，我们还是看下这个结构：

```
1 struct pci_host_bridge_window {
2     struct list_head list;
3     struct resource *res;         /* host bridge aperture (CPU address) */
4     resource_size_t offset;         /* bus address + offset = CPU address 记录总线地址到存储器地址的偏移*/
5 };
```

list使结构作为节点连接在resources链表中，res指向其所对应的resource，而offset表示映射的偏移，该字段主要用在IO端口和IO内存和物理内存的映射，总线号资源这里默认是0.

回到**x86_pci_root_bus_resource**函数中。之前咱们看到初始化的时候x86.init有可能被初始化成acpi_init,如果是这个那么资源很可能就分配好了。所以这里判断下。如果没有的话就调用**pci_add_resource**函数添加资源。所完成的功能就是申请一个**pci_host_bridge_window**结构，指向对应的**res**结构，挂入全局**resource**链表。函数内容我们就不看了，要不就讲不完了。

接着就注册地址空间资源（IO端口和IO内存），这里基本分为两步：

1、创建对应的windows，并挂入链表

2、把res插入到全局的资源树中。

第一步就不在赘述，第二步其实就是从全局的资源中申请空间，类似于windows中VAD树和LInux中的红黑树，但是又不同。这里树的结构前面文章有提到，总之，插入之后表明该段空间已经被占用，在次分配就不能分配和本段冲突。

下面就该**pci_scan_root_bus函数，**这才是探测总线的重点！！

分类：硬件


jack.chen
关注 - 12
粉丝 - 44
+加关注

1 0

« 上一篇： PCI 设备详解一
» 下一篇： LINux网络的NAPI机制详解一

posted @ 2016-10-12 16:33 jack.chen Views(9174) Comments(0) Edit 收藏

相关博文：
· Linux PCI 设备驱动基本框架（二）
· PCI 设备详解一
· PCI设备的地址空间
· Pci设备驱动0:设备枚举
· Pci设备驱动1：pci设备驱动实例(realtek8168)
» 更多推荐...

以马内利