

太初有道，道与神同在，道就是神.....

CnBlogs Home New Post Contact Admin Rss  Posts - 92 Articles - 4 Comments - 45

邮箱: zhunxun@gmail.com

< 2020年5月 >						
日	一	二	三	四	五	六
26	27	28	29	30	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
31	1	2	3	4	5	6

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

- 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 sele
c...
--Linux-inside
- Re:virtIO之VHOST工作原理简析

KVM中断虚拟化浅析

2017-08-24

今天咱们聊聊KVM中断虚拟化, 虚拟机的中断源大致有两种方式, 来自于用户空间qemu和来自于KVM内部。

中断虚拟化起始关键在于对中断控制器的虚拟化, 中断控制器目前主要有APIC, 这种架构下设备控制器通过某种触发方式通知IO APIC, IO APIC根据自身维护的重定向表pci irq routing table格式化出一条中断消息, 把中断消息发送给local APIC, local APIC局部与CPU, 即每个CPU一个, local APIC 具备传统中断控制器的相关功能以及各个寄存器, 中断请求寄存器IRR, 中断屏蔽寄存器IMR, 中断服务寄存器ISF等, 针对这些关键部件的虚拟化是中断虚拟化的重点。在KVM架构下, 每个KVM虚拟机维护一个Io APIC, 但是每个VCPU有一个local APIC。

核心数据结构介绍:

kvm_irq_routing_table

```
struct kvm_irq_routing_table {
    /*ue->gsi*/
    int chip[KVM_NR_IRQCHIPS][KVM_IRQCHIP_NUM_PINS];
    struct kvm_kernel_irq_routing_entry *rt_entries;
    u32 nr_rt_entries;
    /*
     * Array indexed by gsi. Each entry contains list of irq chips
     * the gsi is connected to.
     */
    struct hlist_head map[0];
};
```

这个是一个中断路由表, 每个KVM都有一个, chip是一个二维数组, 表示三个芯片的各个管脚, 每个芯片有24个管脚, 每个数组项纪录对应管脚的GSI号; rt_entries是一个指针, 指向一个kvm_kernel_irq_routing_entry数组, 数组中共有nr_rt_entries项, 每项对应一个IRQ; map其实可以理解为一个链表头数组, 可以根据GSI号作为索引, 找到同一IRQ关联的所有kvm_kernel_irq_routing_entry。具体中断路由表的初始化部分见本文最后一节

```
struct kvm_kernel_irq_routing_entry {
    u32 gsi;
    u32 type;
    int (*set)(struct kvm_kernel_irq_routing_entry *e,
               struct kvm *kvm, int irq_source_id, int level,
               bool line_status);
    union {
        struct {
            unsigned irqchip;
            unsigned pin;
        } irqchip;
        struct msi_msg msi;
    };
    struct hlist_node link;
};
```

gsi是该entry对应的gsi号, 一般和IRQ是一样, set方法是该IRQ关联的触发方法, 通过该方法把IRQ传递给IO-APIC, ; link就是连接点, 连接在上面同一IRQ对应的map上;

中断注入在KVM内部流程起始于一个函数kvm_set_irq

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(21154)
2. PCI 设备详解一(15808)
3. 进程的挂起、阻塞和睡眠(13714)
4. Linux下桥接模式详解一(13467)
5. virtio后端驱动详解(10539)

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)

```
int kvm_set_irq(struct kvm *kvm, int irq_source_id, u32 irq, int level,
                bool line_status)
{
    struct kvm_kernel_irq_routing_entry *e, irq_set[KVM_NR_IRQCHIPS];
    int ret = -1, i = 0;
    struct kvm_irq_routing_table *irq_rt;

    trace_kvm_set_irq(irq, level, irq_source_id);

    /* Not possible to detect if the guest uses the PIC or the
     * IOAPIC. So set the bit in both. The guest will ignore
     * writes to the unused one.
     */
    rcu_read_lock();
    irq_rt = rcu_dereference(kvm->irq_routing);
    if (irq < irq_rt->nr_rt_entries)
        hlist_for_each_entry(e, &irq_rt->map[irq], link)
            irq_set[i++] = *e;
    rcu_read_unlock();
    /*依次调用同一个irq上的所有芯片的set方法*/
    while(i--) {
        int r;
        /*kvm_set_pic_irq kvm_set_ioapic_irq*/
        r = irq_set[i].set(&irq_set[i], kvm, irq_source_id, level,
                          line_status);

        if (r < 0)
            continue;

        ret = r + ((ret < 0) ? 0 : ret);
    }

    return ret;
}
```

kvm指定特定的虚拟机，irq_source_id是中断源ID，一般有KVM_USERSPACE_IRQ_SOURCE_ID和KVM_IRQFD_RESAMPLE_IRQ_SOURCE_ID；irq是全局的中断号，level指定高低电平，需要注意的是，针对边沿触发，需要两个电平触发来模拟，先高电平再低电平。回到函数中，首先要收集的是同一irq上注册的所有的设备信息，这主要在于irq共享的情况，非共享的情况下最多就一个。设备信息抽象成一个kvm_kernel_irq_routing_entry，这里临时放到irq_set数组中。然后对于数组中的每个元素，调用其set方法，目前大都是APIC架构，因此set方法基本都是kvm_set_ioapic_irq，在传统pic情况下，是kvm_set_pic_irq。我们以kvm_set_ioapic_irq为例进行分析，该函数没有实质性的操作，就调用了kvm_ioapic_set_irq函数

```
int kvm_ioapic_set_irq(struct kvm_ioapic *ioapic, int irq, int irq_source_id,
                      int level, bool line_status)
{
    u32 old_irr;
    u32 mask = 1 << irq; //irq对应的位
    union kvm_ioapic_redirect_entry entry;
    int ret, irq_level;

    BUG_ON(irq < 0 || irq >= IOAPIC_NUM_PINS);

    spin_lock(&ioapic->lock);
    old_irr = ioapic->irr;
    /*判断请求高电平还是低电平*/
    irq_level = __kvm_irq_line_state(&ioapic->irq_states[irq],
                                     irq_source_id, level);

    entry = ioapic->redirtbl[irq];
    irq_level ^= entry.fields.polarity;
    /*模拟低电平*/
    if (!irq_level) {
        ioapic->irr &= ~mask;
        ret = 1;
    } else {
        /*判断触发方式*/
        int edge = (entry.fields.trig_mode == IOAPIC_EDGE_TRIG);

        if (irq == RTC_GSI && line_status &&
            rtc_irq_check_coalesced(ioapic)) {
            ret = 0; /* coalesced */
            goto out;
        }
    }
}
```

```

    }
    /*设置中断信号到中断请求寄存器*/
    ioapic->irr |= mask;
    /*如果是电平触发且旧的irr和请求的irr不相等, 调用ioapic_service*/
    if ((edge && old_irr != ioapic->irr) ||
        (!edge && !entry.fields.remote_irr))
        ret = ioapic_service(ioapic, irq, line_status);
    else
        ret = 0; /* report coalesced interrupt */
}
out:
trace_kvm_ioapic_set_irq(entry.bits, irq, ret == 0);
spin_unlock(&ioapic->lock);

return ret;
}

```

到这里, 中断已经到达模拟的IO-APIC了, IO-APIC最重要的就是它的重定向表, 针对重定向表的操作主要在ioapic_service中, 之前都是做一些准备工作, 在进入ioapic_service函数之前, 主要有两个任务:

- 1、判断触发方式, 主要是区分电平触发和边沿触发。
- 2、设置ioapic的irr寄存器。之前我们说过, 电平触发需要两个边沿触发来模拟, 前后电平相反。这里就要先做判断是对应哪一次。只有首次触发才会进行后续的操作, 而二次触发相当于reset操作, 就是把ioapic的irr寄存器清除。在电平触发模式下且请求的irq和ioapic中保存的irq不一致, 就会对其进行更新, 进入ioapic_service函数。

```

static int ioapic_service(struct kvm_ioapic *ioapic, unsigned int idx,
                          bool line_status)
{
    union kvm_ioapic_redirect_entry *pent;
    int injected = -1;
    /*获取重定向表项*/
    pent = &ioapic->redirtbl[idx];

    if (!pent->fields.mask) {
        /*send irq to local apic*/
        injected = ioapic_deliver(ioapic, idx, line_status);
        if (injected && pent->fields.trig_mode == IOAPIC_LEVEL_TRIG)
            pent->fields.remote_irr = 1;
    }
    return injected;
}

```

该函数比较简单, 就是获取根据irq号, 获取重定向表中的一项, 然后向本地APIC传递, 即调用ioapic_deliver函数, 当然前提是kvm_ioapic_redirect_entry没有设置mask, ioapic_deliver主要任务就是根据kvm_ioapic_redirect_entry, 构建kvm_lapic_irq, 这就类似于在总线上的传递过程。构建之后调用kvm_irq_delivery_to_apic, 该函数会把消息传递给相应的VCPU, 具体需要调用kvm_apic_set_irq函数, 继而调用__apic_accept_irq, 该函数中会根据不同的传递模式处理消息, 大部分情况都是APIC_DM_FIXED, 在该模式下, 中断被传递到特定的CPU, 其中会调用kvm_x86_ops->deliver_posted_interrupt, 实际上对应于vmx.c中的vmx_deliver_posted_interrupt

```

static void vmx_deliver_posted_interrupt(struct kvm_vcpu *vcpu, int vector)
{
    struct vmx_vmx *vmx = to_vmx(vcpu);
    int r;
    /*设置位图*/
    if (pi_test_and_set_pir(vector, &vmx->pi_desc))
        return;
    /*标记位图更新标志*/
    r = pi_test_and_set_on(&vmx->pi_desc);
    kvm_make_request(KVM_REQ_EVENT, vcpu);
#ifdef CONFIG_SMP
    if (!r && (vcpu->mode == IN_GUEST_MODE));
    else
#endif
        kvm_vcpu_kick(vcpu);
}

```

这里主要是设置vmx->pi_desc中的位图即struct pi_desc 中的pir字段，其是一个32位的数组，共8项。因此最大标记256个中断，每个中断向量对应一位。设置好后，请求KVM_REQ_EVENT事件，在下次vm-entry的时候会进行中断注入。

具体注入过程：

在vcpu_enter_guest (x86.c)函数中,有这么一段代码

```
if (kvm_check_request(KVM_REQ_EVENT, vcpu) || req_int_win) {
    kvm_apic_accept_events(vcpu);
    if (vcpu->arch.mp_state == KVM_MP_STATE_INIT_RECEIVED) {
        r = 1;
        goto out;
    }
    /*注入中断在vcpu加载到真实cpu上后，相当于某些位已经被设置*/
    inject_pending_event(vcpu); //中断注入
}
.....
```

即在进入非跟模式之前会检查KVM_REQ_EVENT事件，如果存在pending的事件，则调用kvm_apic_accept_events接收，这里主要是处理APIC初始化期间和IPI中断的，暂且不关注。之后会调用inject_pending_event，在这里会检查当前是否有可注入的中断，而具体检查过程时首先会通过kvm_cpu_has_injectable_intr函数，其中调用kvm_apic_has_interrupt->apic_find_highest_irr->vmx_sync_pir_to_irr,vmx_sync_pir_to_irr函数对中断进行收集，就是检查vmx->pi_desc中的位图，如果有，则会调用kvm_apic_update_irr把信息更新到apic寄存器里。然后调用apic_search_irr获取IRR寄存器中的中断，没找到的话会返回-1.找到后调用kvm_queue_interrupt，把中断记录到vcpu中。

```
static inline void kvm_queue_interrupt(struct kvm_vcpu *vcpu, u8 vector,
    bool soft)
{
    vcpu->arch.interrupt.pending = true;
    vcpu->arch.interrupt.soft = soft;
    vcpu->arch.interrupt.nr = vector;
}
```

最后会调用kvm_x86_ops->set_irq，进行中断注入的最后一步，即写入到vmcs结构中。该函数指针指向vmx_inject_irq

```
static void vmx_inject_irq(struct kvm_vcpu *vcpu)
{
    struct vcpu_vmx *vmx = to_vmx(vcpu);
    uint32_t intr;
    int irq = vcpu->arch.interrupt.nr; //中断号

    trace_kvm_inj_virq(irq);

    ++vcpu->stat.irq_injections;
    if (vmx->rmode.vm86_active) {
        int inc_eip = 0;
        if (vcpu->arch.interrupt.soft)
            inc_eip = vcpu->arch.event_exit_inst_len;
        if (kvm_inject_realmode_interrupt(vcpu, irq, inc_eip) != EMULATE_DONE)
            kvm_make_request(KVM_REQ_TRIPLE_FAULT, vcpu);
        return;
    }
    intr = irq | INTR_INFO_VALID_MASK; //设置有中断向量的有效性
    if (vcpu->arch.interrupt.soft) { //如果是软件中断
        intr |= INTR_TYPE_SOFT_INTR; //内部中断
        vmcs_write32(VM_ENTRY_INSTRUCTION_LEN,
            vmx->vcpu.arch.event_exit_inst_len); //软件中断需要写入指令长度
    } else
        intr |= INTR_TYPE_EXT_INTR; //标记外部中断
    vmcs_write32(VM_ENTRY_INTR_INFO_FIELD, intr);
}
```

最终会写入到vmcs的VM_ENTRY_INTR_INFO_FIELD中，这需要按照一定的格式。具体格式详见intel手册。0-7位是向量号，8-10位是中断类型（硬件中断或者软件中断），最高位是有效位，12位是NMI标志。


```
#define INTR_INFO_VECTOR_MASK      0xff          /* 7:0 */
#define INTR_INFO_INTR_TYPE_MASK  0x700         /* 10:8 */
#define INTR_INFO_DELIVER_CODE_MASK 0x800       /* 11 */
#define INTR_INFO_UNBLOCK_NMI     0x1000        /* 12 */
#define INTR_INFO_VALID_MASK      0x80000000    /* 31 */
```

中断路由表的初始化

用户空间qemu通过KVM_CREATE_DEVICE API接口进入KVM的kvm_vm_ioctl处理函数，继而进入kvm_arch_vm_ioctl，根据参数中的KVM_CREATE_IRQCHIP标志进入初始化中断控制器的流程，首先肯定是注册pic和io APIC，这里我们就不详细阐述，重点在于后面对中断路由表的初始化过程。中断路由表的初始化通过kvm_setup_default_irq_routing函数实现，

```
int kvm_setup_default_irq_routing(struct kvm *kvm)
{
    return kvm_set_irq_routing(kvm, default_routing,
                              ARRAY_SIZE(default_routing), 0);
}
```

首个参数kvm指定特定的虚拟机，后面default_routing是一个全局的kvm_irq_routing_entry数组，就定义在irq_comm.c中，该数组没别的作用，就是初始化kvm_irq_routing_table，看下kvm_set_irq_routing

```

int kvm_set_irq_routing(struct kvm *kvm,
                       const struct kvm_irq_routing_entry *ue,
                       unsigned nr,
                       unsigned flags)
{
    struct kvm_irq_routing_table *new, *old;
    u32 i, j, nr_rt_entries = 0;
    int r;
    /*正常情况下, nr_rt_entries=nr*/
    for (i = 0; i < nr; ++i) {
        if (ue[i].gsi >= KVM_MAX_IRQ_ROUTES)
            return -EINVAL;
        nr_rt_entries = max(nr_rt_entries, ue[i].gsi);
    }
    nr_rt_entries += 1;
    /*为中断路由表申请空间*/
    new = kzalloc(sizeof(*new) + (nr_rt_entries * sizeof(struct hlist_head))
                  + (nr * sizeof(struct kvm_kernel_irq_routing_entry)),
                  GFP_KERNEL);

    if (!new)
        return -ENOMEM;
    /*设置指针*/
    new->rt_entries = (void *) &new->map[nr_rt_entries];

    new->nr_rt_entries = nr_rt_entries;
    for (i = 0; i < KVM_NR_IRQCHIPS; i++)
        for (j = 0; j < KVM_IRQCHIP_NUM_PINS; j++)
            new->chip[i][j] = -1;
    /*初始化每一项kvm_kernel_irq_routing_entry*/
    for (i = 0; i < nr; ++i) {
        r = -EINVAL;
        if (ue->flags)
            goto out;
        r = setup_routing_entry(new, &new->rt_entries[i], ue);
        if (r)
            goto out;
        ++ue;
    }
    mutex_lock(&kvm->irq_lock);
    old = kvm->irq_routing;
    kvm_irq_routing_update(kvm, new);
    mutex_unlock(&kvm->irq_lock);

    synchronize_rcu();
    /*释放old*/
}
```

```

    new = old;
    r = 0;
out:
    kfree(new);
    return r;
}

```

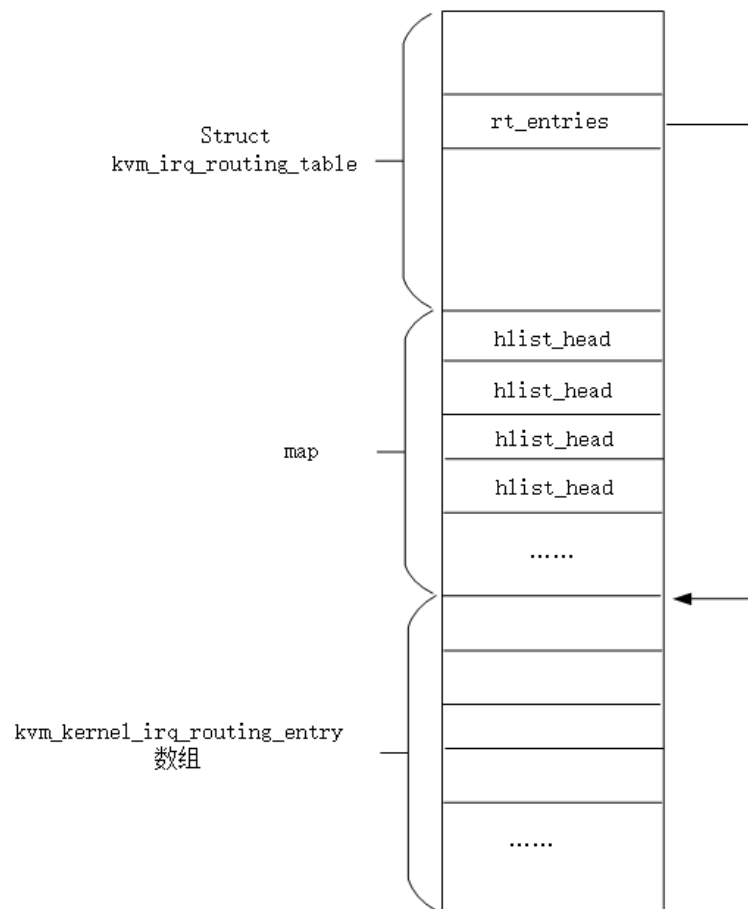
可以参考一个宏：

```

#define IOAPIC_ROUTING_ENTRY(irq) \
{ .gsi = irq, .type = KVM_IRQ_ROUTING_IRQCHIP, .u.irqchip.irqchip = \
  KVM_IRQCHIP_IOAPIC, .u.irqchip.pin = (irq) }

```

这是初始化default_routing的一个关键宏，没一项都是通过该宏传递irq号（0-23）64位下是0-47,可见gsi就是irq号，所以实际上，回到函数中nr_rt_entries就是数组中项数，接着为kvm_irq_routing_table分配空间，注意分配的空间包含三部分：kvm_irq_routing_table结构、nr_rt_entries个hlist_head和nr个kvm_kernel_irq_routing_entry，所以kvm_irq_routing_table的大小是和全局数组的大小一样的。整个结构如下图所示



根据上图就可以理解new->rt_entries = (void *)&new->map[nr_rt_entries];这行代码的含义，接下来是对没项的table的chip数组做初始化，这里初始化为-1.接下来就是一个循环，对每一个kvm_kernel_irq_routing_entry做初始化，该过程是通过setup_routing_entry函数实现的，这里看下该函数

```

static int setup_routing_entry(struct kvm_irq_routing_table *rt,
                             struct kvm_kernel_irq_routing_entry *e,
                             const struct kvm_irq_routing_entry *ue)
{
    int r = -EINVAL;
    struct kvm_kernel_irq_routing_entry *ei;

    /*
     * Do not allow GSI to be mapped to the same irqchip more than once.
     * Allow only one to one mapping between GSI and MSI.
     */
}

```

```

    */
    hlist_for_each_entry(ei, &rt->map[ue->gsi], link)
        if (ei->type == KVM_IRQ_ROUTING_MSI ||
            ue->type == KVM_IRQ_ROUTING_MSI ||
            ue->u.irqchip.irqchip == ei->irqchip.irqchip)
            return r;
    e->gsi = ue->gsi;
    e->type = ue->type;
    r = kvm_set_routing_entry(rt, e, ue);
    if (r)
        goto out;
    hlist_add_head(&e->link, &rt->map[e->gsi]);
    r = 0;
out:
    return r;
}

```

之前的初始化过程我们已经看见了，.type为KVM_IRQ_ROUTING_IRQCHIP，所以这里实际上就是把e->gsi = ue->gsi;e->type = ue->type;然后调用了kvm_set_routing_entry，该函数中主要是设置了kvm_kernel_irq_routing_entry中的set函数，APIC的话设置的是kvm_set_ioapic_irq函数，而pic的话设置kvm_set_pic_irq函数，然后设置irqchip的类型和管脚，对于IOAPIC也是直接复制过来，PIC由于管脚计算是irq%8，所以这里需要加上8的偏移。之后设置table的chip为gsi号。回到setup_routing_entry函数中，就把kvm_kernel_irq_routing_entry以gsi号位索引，加入到了map数组中对应的双链表中。再回到kvm_set_irq_routing函数中，接下来就是更新kvm结构中的irq_routing指针了。

中断虚拟化流程

```

kvm_set_irq
    kvm_ioapic_set_irq
        ioapic_service
            ioapic_deliver
                kvm_irq_delivery_to_apic
                    kvm_apic_set_irq
                        __apic_accept_irq
                            vmx_deliver_posted_interrupt

```

具体注入阶段

```

vcpu_enter_guest
    kvm_apic_accept_events
        inject_pending_event
            kvm_queue_interrupt
                vmx_inject_irq
                    vmcs_write32(VM_ENTRY_INTR_INFO_FIELD, intr);

```

中断路由表初始化

```

x86.c kvm_arch_vm_ioctl
    kvm_setup_default_irq_routing irq_common.c
        kvm_set_irq_routing irq_chip.c
            setup_routing_entry irq_chip.c
                kvm_set_routing_entry irq_chip.c
                    e->set = kvm_set_ioapic_irq; irq_common.c

```

以马内利！

参考资料：

Linux3.10.1源码

分类: [KVM虚拟化技术](#), [linux 内核源码分析](#)

好文要顶

关注我

收藏该文



jack.chen

关注 - 12

粉丝 - 44

[+加关注](#)

« 上一篇: [virtIO之VHOST工作原理简析](#)
» 下一篇: [proc_create函数内幕初探](#)

posted @ 2017-09-04 19:25 jack.chen Views(2507) Comments(0) Edit 收藏

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论, 请 [登录](#) 或 [注册](#), [访问](#) 网站首页。



最新 IT 新闻:

- 腾讯在列! 微软宣布超140家工作室为Xbox Series X开发游戏
 - 黑客声称从微软GitHub私人数据库当中盗取500GB数据
 - IBM开源用于简化AI模型开发的Elyra工具包
 - 中国网民人均安装63个App: 腾讯系一家独大
 - Lyft颁布新规: 强制要求乘客和司机佩戴口罩
- » [更多新闻...](#)

Copyright © 2020 jack.chen
Powered by .NET Core on Kubernetes