

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

qemu进程页表和EPT的同步问题

背景分析:

在之前分析EPT violation的时候, 没有太注意qemu进程页表和EPT的关系, 从虚拟机运行过程分析, 虚拟机访存使用自身页表和EPT完成地址转换, 没有用到qemu进程页表, 所以也就想当然的认为虚拟机使用的物理页面在qemu进程的页表中没有体现. 但是最近才发现, 自己的想法是错误的. Linux kernel作为核心管理层, 具体物理页面的管理有其管理, 再怎么讲, 虚拟机在host上表现为一个qemu进程, 而内存管理器只能根据qemu进程页表管理其所拥有的物理页面, 否则, linux kernel怎么知道哪些物理页面是属于qemu进程的? 这是问题1; 还有一个问题就是用一个实例来讲, virtio 的实现包含前后端驱动两个部分, 前后端其实通过共享内存的方式实现数据的0拷贝. 具体来讲, 虚拟机把数据填充好以后, 通知qemu, qemu得到通过把对应的GPA转化成HVA, 如果两个页表不同步, 怎么保证访问的是同一块内存?

带着上面的问题, 我又重新看了下EPT的维护流程, 终于发现了问题, 事实上, KVM并不负责物理页面的分配, 而是请求qemu分配后把对应的地址传递过来, 然后自己的维护EPT. 也就是说, 在qemu进程建立页表后, EPT才会建立. 下面详细描述下, 整体流程大致如图所示:

handle_ept_violation是处理EPT未命中时候的处理函数, 最终落到tdp_page_fault函数中. 有个细节就是该函数在维护EPT之前, 已经申请好了pfn即对应的物理页框号, 具体见try_async_pf函数, 其实之前也注意过这个函数, 就是没多想! ! 唉.....

```
static bool try_async_pf(struct kvm_vcpu *vcpu, bool prefault, gfn_t gfn,
                        gva_t gva, pfn_t *pfn, bool write, bool *writable)
{
    bool async;

    *pfn = gfn_to_pfn_async(vcpu->kvm, gfn, &async, write, writable);

    if (!async)
        return false; /* *pfn has correct page already */

    if (!prefault && can_do_async_pf(vcpu)) {
        trace_kvm_try_async_get_page(gva, gfn);
        if (kvm_find_async_pf_gfn(vcpu, gfn)) {
            trace_kvm_async_pf_doublefault(gva, gfn);
            kvm_make_request(KVM_REQ_APF_HALT, vcpu);
            return true;
        } else if (kvm_arch_setup_async_pf(vcpu, gva, gfn))
            return true;
    }

    *pfn = gfn_to_pfn_prot(vcpu->kvm, gfn, write, writable);

    return false;
}
```

这里其主要作用的有两个函数和gfn_to_pfn_prot, 二者均调用了static pfn_t __gfn_to_pfn(struct kvm *kvm, gfn_t gfn, bool atomic, bool *async, bool write_fault, bool *writable)函数, 区别在于第四个参数bool *async, 前者不为NULL, 而后者为NULL. 先跟着gfn_to_pfn_async函数往下走, 该函数直接调用了__gfn_to_pfn(kvm, gfn, false, async, write_fault, writable);可以看到这里atomic参数被设置成false.

```
static pfn_t __gfn_to_pfn(struct kvm *kvm, gfn_t gfn, bool atomic, bool *async,
                        bool write_fault, bool *writable)
{
    struct kvm_memory_slot *slot;

    if (async)
        *async = false;
```

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)

```
slot = gfn_to_memslot(kvm, gfn);

return __gfn_to_pfn_memslot(slot, gfn, atomic, async, write_fault,
                             writable);
}
```

在__gfn_to_pfn函数中，如果async不为NULL，则初始化成false，然后根据gfn获取对应的slot结构。接下来调用__gfn_to_pfn_memslot(slot, gfn, atomic, async, write_fault, writable);该函数主要做了两件事情，首先根据gfn和slot得到具体得到host的虚拟地址，然后就是调用了hva_to_pfn函数根据虚拟地址得到对应的pfn。

```
static pfn_t hva_to_pfn(unsigned long addr, bool atomic, bool *async,
                        bool write_fault, bool *writable)
{
    struct vm_area_struct *vma;
    pfn_t pfn = 0;
    int npages;

    /* we can do it either atomically or asynchronously, not both */
    /*这里二者不能同时为真*/
    BUG_ON(atomic && async);
    /*主要实现逻辑*/
    if (hva_to_pfn_fast(addr, atomic, async, write_fault, writable, &pfn))//查tlb缓存
        return pfn;

    if (atomic)
        return KVM_PFN_ERR_FAULT;
    /*如果前面没有成功，则调用hva_to_pfn_slow*/
    npages = hva_to_pfn_slow(addr, async, write_fault, writable, &pfn);//快表未命中，查内存
    if (npages == 1)
        return pfn;

    down_read(&current->mm->mmap_sem);
    if (npages == -EHWPOISON ||
        (!async && check_user_page_hwpoison(addr))) {
        pfn = KVM_PFN_ERR_HWPOISON;
        goto exit;
    }

    vma = find_vma_intersection(current->mm, addr, addr + 1);

    if (vma == NULL)
        pfn = KVM_PFN_ERR_FAULT;
    else if ((vma->vm_flags & VM_PFNMAP)) {
        pfn = ((addr - vma->vm_start) >> PAGE_SHIFT) +
            vma->vm_pgoff;
        /*如果PFN不是MMIO*/
        BUG_ON(!kvm_is_mmio_pfn(pfn));
    } else {
        if (async && vma_is_valid(vma, write_fault))
            *async = true;
        pfn = KVM_PFN_ERR_FAULT;
    }
exit:
    up_read(&current->mm->mmap_sem);
    return pfn;
}
```

在本函数中涉及到两个重要函数hva_to_pfn_fast和hva_to_pfn_slow，首选是前者，在前者失败后，调用后者。hva_to_pfn_fast核心是调用了__get_user_pages_fast函数，而hva_to_pfn_slow函数的主体其实是get_user_pages_fast函数，可以看到这里两个函数就查了一个前缀，前者默认页表项已经存在，直接通过遍历页表得到对应的页框；而后者不做这种假设，如果有页表项没有建立，还需要建立页表项，物理页面没有分配就需要分配物理页面。考虑到这里是KVM，在开始EPT violation时候，虚拟地址肯定没有分配具体的物理地址，所以这里调用后者的可能性比较大。get_user_pages_fast函数的前半部分基本就是__get_user_pages_fast，所以这里我们简要分析下get_user_pages_fast函数。

```

int get_user_pages_fast(unsigned long start, int nr_pages, int write,
                        struct page **pages)
{
    struct mm_struct *mm = current->mm;
    unsigned long addr, len, end;
    unsigned long next;
    pgd_t *pgdp, pgd;
    int nr = 0;

    start &= PAGE_MASK;
    addr = start;
    len = (unsigned long) nr_pages << PAGE_SHIFT;
    end = start + len;
    if ((end < start) || (end > TASK_SIZE))
        goto slow_irqon;

    /*
     * local_irq_disable() doesn't prevent pagetable teardown, but does
     * prevent the pagetables from being freed on s390.
     *
     * So long as we atomically load page table pointers versus teardown,
     * we can follow the address down to the the page and take a ref on it.
     */
    local_irq_disable();
    pgdp = pgd_offset(mm, addr);
    do {
        pgd = *pgdp;
        barrier();
        next = pgd_addr_end(addr, end);
        if (pgd_none(pgd))
            goto slow;
        if (!gup_pud_range(pgdp, pgd, addr, next, write, pages, &nr))
            goto slow;
    } while (pgdp++, addr = next, addr != end);
    local_irq_enable();

    VM_BUG_ON(nr != (end - start) >> PAGE_SHIFT);
    return nr;

    {
        int ret;
slow:
        local_irq_enable();
slow_irqon:
        /* Try to get the remaining pages with get_user_pages */
        start += nr << PAGE_SHIFT;
        pages += nr;

        down_read(&mm->mmap_sem);
        ret = get_user_pages(current, mm, start,
                              (end - start) >> PAGE_SHIFT, write, 0, pages, NULL);
        up_read(&mm->mmap_sem);


        /* Have to be a bit careful with return values */
        if (nr > 0) {
            if (ret < 0)
                ret = nr;
            else
                ret += nr;
        }

        return ret;
    }
}

```

函数开始获取虚拟页框号和结束地址，在咱们分析的情况下，一般这里就是一个页面的大小。然后调用 `local_irq_disable` 禁止本地中断，开始遍历当前进程的页表。pgdp是在页目录表中的偏移+一级页表基址。进入while循环，获取二级表的基址，next在这里基本就是end了，因为前面申请的仅仅是一个页面的长度。可以看到这里如果表项内容为空，则goto到了slow，即要为其建立表项。这里暂且略过。先假设其存在，继续调用gup_pud_range函数。在x86架构下，使用的二级页表而在64位下使用四级页表。64位暂且不考虑，所以中间两层处理其实就是走个过场。这里直接把pgdp指针转成了pudp即pud_t类型的指

针，接下来还是进行同样的工作，只不过接下来调用的是gup_pmd_range函数，该函数取出表项的内容，往下一级延伸，重点看其调用的gup_pte_range函数。




```
static inline int gup_pte_range(pmd_t *pmdp, pmd_t pmd, unsigned long addr,
                               unsigned long end, int write, struct page **pages, int *nr)
{
    unsigned long mask;
    pte_t *ptep, pte;
    struct page *page;

    mask = (write ? _PAGE_RO : 0) | _PAGE_INVALID | _PAGE_SPECIAL;

    ptep = ((pte_t *) pmd_deref(pmd)) + pte_index(addr);
    do {
        pte = *ptep;
        barrier();
        if ((pte_val(pte) & mask) != 0)
            return 0;
        VM_BUG_ON(!pfn_valid(pte_pfn(pte)));
        page = pte_page(pte);
        if (!page_cache_get_speculative(page))
            return 0;
        if (unlikely(pte_val(pte) != pte_val(*ptep))) {
            put_page(page);
            return 0;
        }
        pages[*nr] = page;
        (*nr)++;

    } while (ptep++, addr += PAGE_SIZE, addr != end);

    return 1;
}
```



这里就根据pmd和虚拟地址的二级偏移，定位到二级页表项的地址ptep，在循环中，就取出ptep的内容，不出意外就是物理页面的地址及pfn，后面调用了page = pte_page(pte);实质是把pfn转成了page结构，然后设置参数中的page数组。没有错误就返回1.上面就是整个页表遍历的过程。如果失败了，就为其维护页表并分配物理页面，注意这里如果当初申请的是多个页面，就一并处理了，而不是一个页面一个页面的处理。实现的主体是get_user_pages函数，该函数是__get_user_pages函数的封装，__get_user_pages比较长，我们这里就不在介绍，感兴趣的朋友可以参考具体的代码或者其他资料。

参考资料：

linux内核3.10.1代码

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](#)

00

« 上一篇: [Linux进程虚拟地址空间管理2](#)
» 下一篇: [qemu-kvm内存虚拟化2](#)

posted @ 2017-04-23 20:16 jack.chen Views(1221) 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。

- 【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】96秒100亿！哪些“黑科技”支撑全球最大流量洪峰？
- 【推荐】阿里专家五年方法论总结！技术人如何实现职业突破？

相关博文：

- KVM中EPT逆向映射机制分析
 - intel EPT 机制详解
 - KVM地址翻译流程及EPT页表的建立过程
 - kvm_read_guest*函数分析
 - KVM的ept机制
- » 更多推荐...

2019必看8大技术大会 & 300+ 公开课全集（500+PDF下载）

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...