intel EPT 机制详解

2016-11-08

在虚拟化环境下，intel CPU在处理器级别加入了对内存虚拟化的支持。即扩展页表EPT，而AMD也有类似的成为NPT。在此之前，内存虚拟化使用的一个重要技术为影子页表。

背景：

在虚拟化环境下，虚拟机使用的是客户机虚拟地址GVA，而其本身页表机制只能把客户机的虚拟地址转换成客户机的物理地址也就是完成GVA->GPA的转换，但是GPA并不是被用来真正的访存，所以需要想办法把客户机的物理地址GPA转换成宿主机的物理地址HPA。影子页表采用的是一步到位式，即完成客户机虚拟地址GVA到宿主机物理地址HPA的转换，由VMM为每个客户机进程维护。本节对于影子页表不做过多描述，重点在于EPT。内容我想分为两部分，第一部分根据intel手册分析EPT地址转换机制；第二部分借助于KVM源代码分析EPT构建过程。

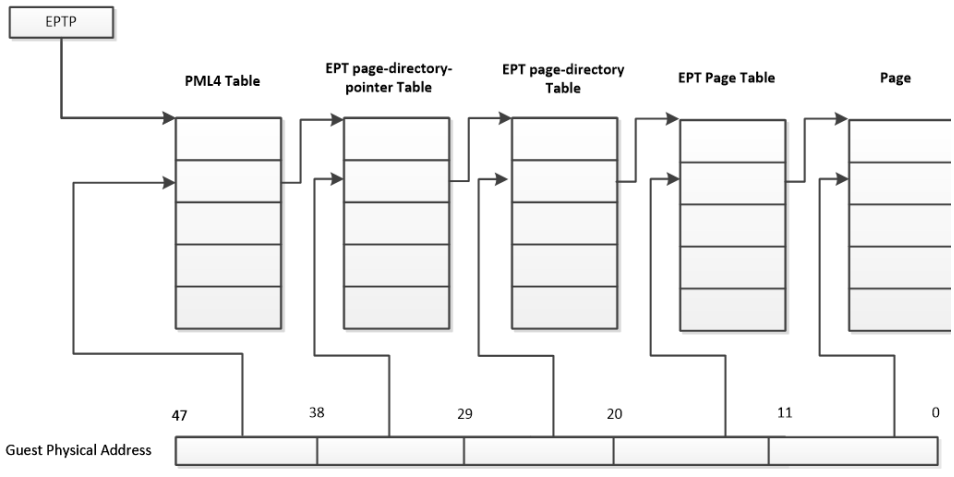
一、EPT地址转换机制

本节内容完全脱离代码，参考intel手册对EPT做出解释。

当一个逻辑CPU处于非根模式下运行客户机代码时，使用的地址是客户机虚拟地址，而访问这个虚拟地址时，同样会发生地址的转换，这里的转换还没有设计到VMM层，和正常的系统一样，这里依然是采用CR3作为基址，利用客户机页表进行地址转换，只是到这里虽然已经转换成物理地址，但是由于是客户机物理地址，不等同于宿主机的物理地址，所以并不能直接访问，需要借助于第二次的转换，也就是EPT的转换。注意EPT的维护有VMM维护，其转换过程由硬件完成，所以其比影子页表有更高的效率。

我们假设已经获取到了客户机的物理地址，下面分析下如何利用一个客户机的物理地址，通过EPT进行寻址。

先看下图：



注意不管是32位客户机还是64位客户机，这里统一按照64位物理地址来寻址。EPT页表是4级页表，页表的大小仍然是一个页即4KB，但是一个表项是8个字节，所以一张表只能容纳512个表项，需要9位来定位具体的表项。客户机的物理地址使用低48位来完成这一工作。从上图可以看到，一个48位的客户机物理地址被分为5部分，前4部分按9位划分，最后12位作为页内偏移。当处于非根模式下的CPU使用客户机操作一个客户机虚拟地址时，首先使用客户机页表进行地址转换，得到客户机物理地址，然后CPU根据此物理地址查询EPT，在VMCS结构中有一个EPTP的指针，其中的12-51位指向EPT页表的一级目录即PML4 Table.这样根据客户机物理地址的首个9位就可以定位一个PML4 entry，一个PML4 entry理论上可以控

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

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

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(15806)
3. 进程的挂起、阻塞和睡眠(13713)
4. Linux下桥接模式详解一(13465)
5. virtio后端驱动详解(10538)

推荐排行榜

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

制512GB的区域，这里不是重点，我们不在多说。PML4 entry的格式如下：

Bit Position(s)	Contents
0	Read access; indicates whether reads are allowed from the 512-GByte region controlled by this entry
1	Write access; indicates whether writes are allowed to the 512-GByte region controlled by this entry
2	Execute access; indicates whether instruction fetches are allowed from the 512-GByte region controlled by this entry
7:3	Reserved (must be 0)
8	If bit 6 of EPTP is 1, accessed flag for EPT; indicates whether software has accessed the 512-GByte region controlled by this entry (see Section 28.2.4). Ignored if bit 6 of EPTP is 0
11:9	Ignored
(N–1):12	Physical address of 4-KByte aligned EPT page-directory-pointer table referenced by this entry[1]
51:N	Reserved (must be 0)
63:52	Ignored

1、其实这里我们只需要知道PML4 entry的12-51位记录下一级页表的地址，而这40位肯定是用不完的，根据CPU的架构，采取不同的位数，具体如下：

在Intel中使用MAXPHYADDR来表示最大的物理地址，我们可以通过CPUID的指令来获得处理支持的最大物理地址，然而这已经不在此次的讨论范围之内，我们需要知道的只是：
当MAXPHYADDR 为36位，在Intel平台的桌面处理器上普遍实现了36位的最高物理地址值，也就是我们普通的个人计算机，可寻址64G空间；
当MAXPHYADDR 为40位，在Inter的服务器产品和AMD 的平台上普遍实现40位的最高物理地址，可寻址达1TB；
当MAXPHYADDR为52位，这是x64体系结构描述最高实现值，目前尚未有处理器实现。

而对下级表的物理地址的存储4K页面寻址遵循如下规则：
① 当MAXPHYADDR为52位时，上一级table entry的12~51位提供下一级table物理基地址的高40位，低12位补零，达到基地址在4K边界对齐；
② 当MAXPHYADDR为40位时，上一级table entry的12~39位提供下一级table物理基地址的高28位，此时40~51是保留位，必须置0，低12位补零，达到基地址在4K边界对齐；
③ 当MAXPHYADDR为36位时，上一级table entry的12~35位提供下一级table物理基地址的高24位，此时36~51是保留位，必须置0，低12位补零，达到基地址在4K边界对齐。

而MAXPHYADDR为36位正是普通32位机的PAE模式。

2、就这么定位为下一级的页表EPT Page-Directory-Pointer-Table ，根据客户物理地址的30-38位定位此页表中的一个表项EPT Page-Directory-Pointer-Table entry。注意这里如果该表项的第7位为1，该表项指向一个1G字节的page.为0，则指向下一级页表。下面我们只考虑的是指向页表的情况。

3、然后根据表项中的12-51位，继续往下定位到第三级页表EPT Page-Directory-Pointer-Table，在根据客户物理地址的21-29位来定位到一个EPT Page-Directory-Pointer-Table Entry。如果此entry的第7位为1，则表示该entry指向一个2M的page，为0就指向下一级页表。

4、根据entry的12-51位定位第四级页表EPT Page-Directory ，然后根据客户物理地址的12-20位定位一个PDE。

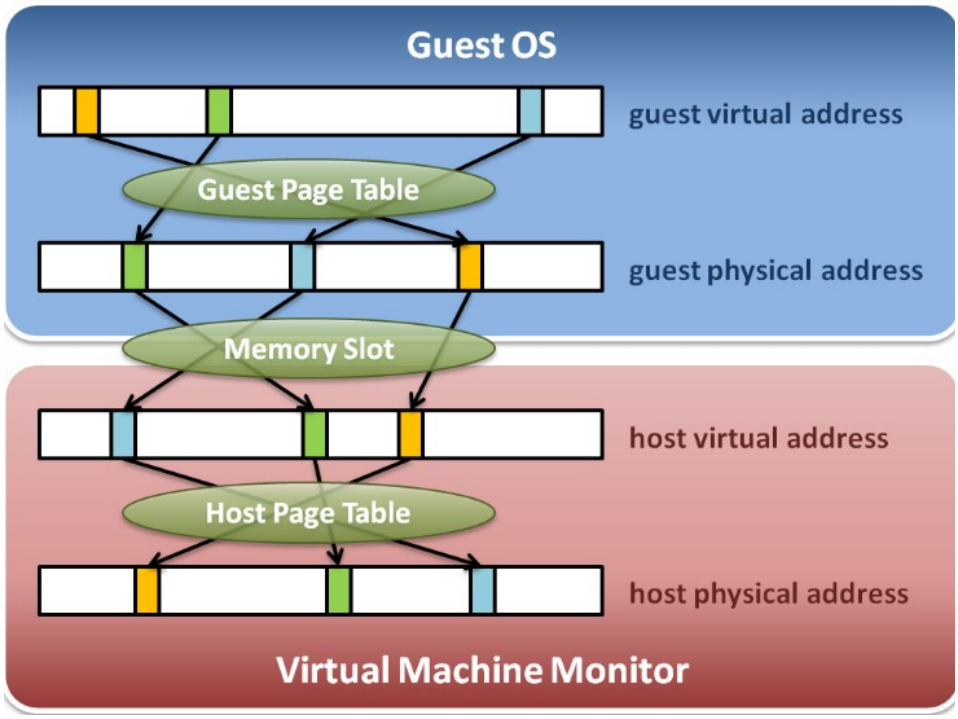
PDE的12-51位指向一个4K物理页面，最后根据客户物理地址的最低12位作为偏移，定位到具体的物理地址。

二、EPT寻址过程

在此之前我们先了解下KVM虚拟机的物理内存组织方式，众所周知，KVM虚拟机运行在qemu的进程地址空间中，所以其实虚拟机使用的物理地址是从对应qemu进程的地址空间中分配的。具体由一个kvm_memory_slot结构管理，结构内容如下：

```
struct kvm_memory_slot {
    gfn_t base_gfn;
    unsigned long npages;
    /*一个slot有许多客户机虚拟页面组成，通过dirty_bitmap标记每一个页是否可用*/
    unsigned long *dirty_bitmap;
    struct kvm_arch_memory_slot arch;
    unsigned long userspace_addr;
    u32 flags;
    short id;
};
```

每个虚拟机的物理内存由多个slot组成，每个slot对应一个kvm_memory_slot结构，从上面的字段可以看出，该结构记录slot映射的是哪些客户物理page，由于映射多个页面，所以有一个ditty_bitmap来标识各个页的状态，注意这个页时客户机的虚拟page。映射架构如下：



下面借助于KVM源代码分析下EPT的构建过程，其构建模式和普通页表一样，属于中断触发式。即初始页表是空的，只有在访问未命中的时候引发缺页中断，然后缺页处理程序构建页表。

初始状态EPT页表为空，当客户机运行时，其使用的GVA转化成GPA后，还需要CPU根据GPA查找EPT，从而定位具体的HPA，但是由于此时EPT为空，所以会引发缺页中断，发生VM-exit,此时CPU进入到根模式，运行VMM（这里指KVM），在KVM中定义了一个异常处理数组来处理对应的VM-exit，

```
1 static int (*const kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu) = {
2     ......3     [EXIT_REASON_EPT_VIOLATION]              = handle_ept_violation,
4     ......5 }
```

所以在发生EPT violation的时候，KVM中会执行handle_ept_violation：

```
1 static int handle_ept_violation(struct kvm_vcpu *vcpu)
2 {
3     unsigned long exit_qualification;
4     gpa_t gpa;
5     u32 error_code;
6     int gla_validity;
7
8     exit_qualification = vmcs_readl(EXIT_QUALIFICATION);
9
10    gla_validity = (exit_qualification >> 7) & 0x3;
11    if (gla_validity != 0x3 && gla_validity != 0x1 && gla_validity != 0) {
12        printk(KERN_ERR "EPT: Handling EPT violation failed!\n");
13        printk(KERN_ERR "EPT: GPA: 0x%lx, GVA: 0x%lx\n",
14            (long unsigned int)vmcs_read64(GUEST_PHYSICAL_ADDRESS),
15            vmcs_readl(GUEST_LINEAR_ADDRESS));
16        printk(KERN_ERR "EPT: Exit qualification is 0x%lx\n",
17            (long unsigned int)exit_qualification);
18        vcpu->run->exit_reason = KVM_EXIT_UNKNOWN;
19        vcpu->run->hw.hardware_exit_reason = EXIT_REASON_EPT_VIOLATION;
20        return 0;
21    }
22
23    gpa = vmcs_read64(GUEST_PHYSICAL_ADDRESS);
24    trace_kvm_page_fault(gpa, exit_qualification);
25
26    /* It is a write fault? */
27    error_code = exit_qualification & (1U << 1);
```

```
    /* ept page table is present? */
    error_code |= (exit_qualification >> 3) & 0x1;

    return kvm_mmu_page_fault(vcpu, gpa, error_code, NULL, 0);
}
```

而该函数并没有做具体的工作，只是获取一下发生VM-exit的时候的一些状态信息如发生此VM-exit的时候正在执行的客户物理地址GPA、退出原因等，然后作为参数继续往下传递，调用kvm_mmu_page_fault

```
int kvm_mmu_page_fault(struct kvm_vcpu *vcpu, gva_t cr2, u32 error_code,
                void *insn, int insn_len)
{
    int r, emulation_type = EMULTYPE_RETRY;
    enum emulation_result er;

    r = vcpu->arch.mmu.page_fault(vcpu, cr2, error_code, false);
    if (r < 0)
        goto out;

    if (!r) {
        r = 1;
        goto out;
    }
    //查看是否是MMIO 引起的退出
    if (is_mmio_page_fault(vcpu, cr2))
        emulation_type = 0;

    er = x86_emulate_instruction(vcpu, cr2, emulation_type, insn, insn_len);

    switch (er) {
    case EMULATE_DONE:
        return 1;
    case EMULATE_USER_EXIT:
        ++vcpu->stat.mmio_exits;
        /* fall through */
    case EMULATE_FAIL:
        return 0;
    default:
        BUG();
    }
out:
    return r;
}
```

该函数会调用MMU单元注册的pagefault函数，具体初始化过程我们简单看下：

```
static int init_kvm_mmu(struct kvm_vcpu *vcpu)
{
    if (mmu_is_nested(vcpu))
        return init_kvm_nested_mmu(vcpu);
    else if (tdp_enabled)//是否支持EPT
        return init_kvm_tdp_mmu(vcpu);//EPT初始化方式
    else
        return init_kvm_softmmu(vcpu);//影子页表初始化方式
}
```

第一种情况是嵌套虚拟化的，我们暂且不考虑，可以看到在支持EPT的情况下，会调用init_kvm_tdp_mmu函数初始化MMU。在该函数中

```
static int init_kvm_tdp_mmu(struct kvm_vcpu *vcpu)
{
```

```
    struct kvm_mmu *context = vcpu->arch.walk_mmu;

    ......
    context->page_fault = tdp_page_fault;
    ......

    return 0;
}
```

vcpu->arch.walk_mmu.pagefault被初始化成tdp_page_fault。所以我们的正式分析从tdp_page_fault函数开始。

```
1  static int tdp_page_fault(struct kvm_vcpu *vcpu, gva_t gpa, u32 error_code,
2          bool prefault)
3  {
4      pfn_t pfn;
5      int r;
6      int level;
7      int force_pt_level;
8      gfn_t gfn = gpa >> PAGE_SHIFT;//物理地址右移12位得到物理页框号(相对于虚拟机而言)
9      unsigned long mmu_seq;
10     int write = error_code & PFERR_WRITE_MASK;
11     bool map_writable;
12
13     ASSERT(vcpu);
14     ASSERT(VALID_PAGE(vcpu->arch.mmu.root_hpa));
15     if (unlikely(error_code & PFERR_RSVD_MASK)) {
21         r = handle_mmio_page_fault(vcpu, gpa, error_code, true);//mmio pagefault的处理
22
23         if (likely(r != RET_MMIO_PF_INVALID))
24             return r;
25     }
26     r = mmu_topup_memory_caches(vcpu);//分配缓存池
27     if (r)
28         return r;
29     force_pt_level = mapping_level_dirty_bitmap(vcpu, gfn);
30     if (likely(!force_pt_level)) {
31         level = mapping_level(vcpu, gfn);
32         /*这里是获取大页的页框号*/
33         gfn &= ~(KVM_PAGES_PER_HPAGE(level) - 1);
34     } else
35         level = PT_PAGE_TABLE_LEVEL;
36     /**/
37     if (fast_page_fault(vcpu, gpa, level, error_code))//
38         return 0;
39     mmu_seq = vcpu->kvm->mmu_notifier_seq;
40     smp_rmb();
41     /*得到PFN */
42     if (try_async_pf(vcpu, prefault, gfn, gpa, &pfn, write, &map_writable))
43         return 0;
44     if (handle_abnormal_pfn(vcpu, 0, gfn, pfn, ACC_ALL, &r))//处理反常的物理页框
45         return r;
46     spin_lock(&vcpu->kvm->mmu_lock);
47     if (mmu_notifier_retry(vcpu->kvm, mmu_seq))
48         goto out_unlock;
49     make_mmu_pages_available(vcpu);
50     if (likely(!force_pt_level))
51         transparent_hugepage_adjust(vcpu, &gfn, &pfn, &level);
52     r = __direct_map(vcpu, gpa, write, map_writable,
53             level, gfn, pfn, prefault);//修正EPT
54     spin_unlock(&vcpu->kvm->mmu_lock);
55     return r;
56
57 out_unlock:
58     spin_unlock(&vcpu->kvm->mmu_lock);
59     kvm_release_pfn_clean(pfn);
60     return 0;
61 }
```

该函数首先就判断本次exit是否是MMIO引起的，如果是，则调用handle_mmio_page_fault函数处理MMIO pagefault,具体为何这么判断可参考intel手册。然后调用mmu_topup_memory_caches函数进

行缓存池的分配，官方的解释是为了避免在运行时分配空间失败，这里提前分配浩足额的空间，便于运行时使用。该部分内容最后单独详解。然后调用mapping_level_dirty_bitmap函数判断当前gfn对应的slo是否可用，当然绝大多数情况下是可用的。为什么要进行这样的判断呢？在if内部可以看到是获取level，如果当前GPN对应的slot可用，我们就可以获取分配slot的pagesize，然后得到最低级的level，比如如果是2M的页，那么level就为2，为4K的页，level就为1.

接着调用了fast_page_fault尝试快速处理violation，只有当GFN对应的物理页存在且violation是由读写操作引起的，才可以使用快速处理，因为这样不用加MMU-lock.

假设这里不能快速处理，那么到后面就调用try_async_pf函数根据GFN获取对应的PFN，这个过程具体来说需要首先获取GFN对应的slot，转化成HVA，接着就是正常的HOST地址翻译的过程了，如果HVA对应的地址并不在内存中，还需要HOST自己处理缺页中断。

接着调用transparent_hugepage_adjust对level和gfn、pfn做出调整。紧着着就调用了__direct_map函数，该函数是构建页表的核心函数：

```
1 static int __direct_map(struct kvm_vcpu *vcpu, gpa_t v, int write,
2             int map_writable, int level, gfn_t gfn, pfn_t pfn,
3             bool prefault)
4 {
5     struct kvm_shadow_walk_iterator iterator;
6     struct kvm_mmu_page *sp;
7     int emulate = 0;
8     gfn_t pseudo_gfn;
9 
10    for_each_shadow_entry(vcpu, (u64)gfn << PAGE_SHIFT, iterator) {
11        /*如果需要映射的level正是iterator.level，那么*/
12        if (iterator.level == level) {
13            mmu_set_spte(vcpu, iterator.sptep, ACC_ALL,
14                     write, &emulate, level, gfn, pfn,
15                     prefault, map_writable);
16            direct_pte_prefetch(vcpu, iterator.sptep);
17            ++vcpu->stat.pf_fixed;
18            break;
19        }
20        /*判断当前entry指向的页表是否存在，不存在的话需要建立*/
21        if (!is_shadow_present_pte(*iterator.sptep)) {
22            /*iterator.addr是客户物理地址的物理页帧*/
23            u64 base_addr = iterator.addr;
24            /*确保对应层级的偏移部分为0，如level=1，则baseaddr的低12位就清零*/
25            base_addr &= PT64_LVL_ADDR_MASK(iterator.level);//
26            /*得到物理页框号*/
27            pseudo_gfn = base_addr >> PAGE_SHIFT;
28            sp = kvm_mmu_get_page(vcpu, pseudo_gfn, iterator.addr,
29                          iterator.level - 1,
30                          1, ACC_ALL, iterator.sptep);
31             /*设置页表项的sptep指针指向sp*/
32            link_shadow_page(iterator.sptep, sp);
33        }
34    }
35    return emulate;
36 }
```

首先进入的便是for_each_shadow_entry，用于根据GFN遍历EPT页表的对应项，这点后面会详细解释。循环中首先判断entry的level和请求的level是否相等，相等说明该entry处引起的violation，即该entry对应的下级页或者页表不在内存中，或者直接为NULL。

如果level不相等，就进入后面的if判断，这是判断该entry对应的下一级页是否存在，如果不存在需要重新构建，存在就直接向后遍历，即对比二级页表中的entry。整个处理流程就是这样，根据GPA组逐层查找EPT，最终level相等的时候，就根据最后一层的索引定位一个PTE，该PTE应该指向的就是GFN对应的PFN，那么这时候set spite就可以了。最好的情况就是最后一级页表中的entry指向的物理页被换出外磁盘，这样只需要处理一次EPT violation，而如果在初始全部为空的状态下访问，每一级的页表都需要重新构建，则需要处理四次EPTviolation，发生4次VM-exit。

构建页表的过程即在level相等之前，发现需要的某一级的页表项为NULL，就调用kvm_mmu_get_page获取一个page，然后调用link_shadow_page设置页表项指向page，

看下kvm_mmu_get_page函数、

```
static struct kvm_mmu_page *kvm_mmu_get_page(struct kvm_vcpu *vcpu,
                         gfn_t gfn,
                         gva_t gaddr,
                         unsigned level,
                         int direct,
                         unsigned access,
                         u64 *parent_pte)
{
    union kvm_mmu_page_role role;
    unsigned quadrant;
    struct kvm_mmu_page *sp;
    bool need_sync = false;

    role = vcpu->arch.mmu.base_role;
    role.level = level;
    role.direct = direct;
    if (role.direct)
        role.cr4_pae = 0;
    role.access = access;
    /*quadrant 对应页表项的索引，来自于GPA*/
    if (!vcpu->arch.mmu.direct_map
        && vcpu->arch.mmu.root_level <= PT32_ROOT_LEVEL) {
        quadrant = gaddr >> (PAGE_SHIFT + (PT64_PT_BITS * level));
        quadrant &= (1 << ((PT32_PT_BITS - PT64_PT_BITS) * level)) - 1;
        role.quadrant = quadrant;
    }
    /*根据gfn遍历KVM维护的mmu_page_hash哈希链表*/
    for_each_gfn_sp(vcpu->kvm, sp, gfn) {
        /**/
        if (is_obsolete_sp(vcpu->kvm, sp))
            continue;

        if (!need_sync && sp->unsync)
            need_sync = true;

        if (sp->role.word != role.word)
            continue;

        if (sp->unsync && kvm_sync_page_transient(vcpu, sp))
            break;
        /*设置sp->parent_pte=parent_pte*/
        mmu_page_add_parent_pte(vcpu, sp, parent_pte);
        if (sp->unsync_children) {
            kvm_make_request(KVM_REQ_MMU_SYNC, vcpu);
            kvm_mmu_mark_parents_unsync(sp);
        } else if (sp->unsync)
            kvm_mmu_mark_parents_unsync(sp);

        __clear_sp_write_flooding_count(sp);
        trace_kvm_mmu_get_page(sp, false);
        return sp;
    }
    /*如果根据页框号没有遍历到合适的page，就需要重新创建一个页*/
    ++vcpu->kvm->stat.mmu_cache_miss;
    sp = kvm_mmu_alloc_page(vcpu, parent_pte, direct);
    if (!sp)
        return sp;
    /*设置其对应的客户机物理页框号*/
    sp->gfn = gfn;
    sp->role = role;
    /*把该也作为一个节点加入到哈希表相应的链表汇总*/
    hlist_add_head(&sp->hash_link,
        &vcpu->kvm->arch.mmu_page_hash[kvm_page_table_hashfn(gfn)]);
    if (!direct) {
        if (rmap_write_protect(vcpu->kvm, gfn))
            kvm_flush_remote_tlbs(vcpu->kvm);
        if (level > PT_PAGE_TABLE_LEVEL && need_sync)
            kvm_sync_pages(vcpu, gfn);

        account_shadowed(vcpu->kvm, gfn);
    }
    sp->mmu_valid_gen = vcpu->kvm->arch.mmu_valid_gen;
    /*暂时对所有表项清零*/
    init_shadow_page_table(sp);
    trace_kvm_mmu_get_page(sp, true);
```

```
76      return sp;
77  }
```

具体的细节方面后面单独讲述，比如kvm_mmu_page_role结构，目前我们只需要知道一个kvm_mmu_page对应于一个kvm_mmu_page_role，kvm_mmu_page_role记录对应page的各种属性。下面for_each_gfn_sp是一个遍历链表的宏定义，KVM为了根据GFN查找对应的kvm_mmu_page，用一个HASH数组记录所有的kvm_mmu_page，每一个表项都是一个链表头，即根据GFN获取到的HASH值相同的，位于一个链表中。这也是HASH表处理冲突常见方法。

如果在对应链表中找到一个合适的页（怎么算是合适暂且不清楚），就直接利用该页，否则需要调用kvm_mmu_alloc_page函数重新申请一个页，主要是申请一个kvm_mmu_page结构和一个存放表项的page，这就用到了之前我们说过的三种缓存，不错这里只用到了两个，分别是mmu_page_header_cache和mmu_page_cache。这样分配好后，把对应的kvm_mmu_page作为一个节点加入到全局的HASH链表中，然后对数组项清零，最后返回sp.

for_each_shadow_entry

```
1 #define for_each_shadow_entry(_vcpu, _addr, _walker)    \
2     for (shadow_walk_init(&(_walker), _vcpu, _addr);    \
3          shadow_walk_okay(&(_walker));            \
4          shadow_walk_next(&(_walker)))
```

说白了其实就是一个for循环，只不过循环的三个部分由三个函数组成，有点小复杂。

```
 1 static void shadow_walk_init(struct kvm_shadow_walk_iterator *iterator,
 2                 struct kvm_vcpu *vcpu, u64 addr)
 3 {
 4     iterator->addr = addr;
 5     iterator->shadow_addr = vcpu->arch.mmu.root_hpa;
 6     iterator->level = vcpu->arch.mmu.shadow_root_level;
 7
 8     if (iterator->level == PT64_ROOT_LEVEL &&
 9         vcpu->arch.mmu.root_level < PT64_ROOT_LEVEL &&
10         !vcpu->arch.mmu.direct_map)
11         --iterator->level;
12
13     if (iterator->level == PT32E_ROOT_LEVEL) {
14         iterator->shadow_addr
15             = vcpu->arch.mmu.pae_root[(addr >> 30) & 3];
16         iterator->shadow_addr &= PT64_BASE_ADDR_MASK;
17         --iterator->level;
18         if (!iterator->shadow_addr)
19             iterator->level = 0;
20     }
21 }
```

初始化函数，即CPU拿到一个GPA，构建页表的第一步，构建最初始的kvm_shadow_walk_iterator，而关于此结构：

```
1 struct kvm_shadow_walk_iterator {
2     u64 addr;//寻找的GuestOS的物理页帧，即(u64)gfn << PAGE_SHIFT
3     hpa_t shadow_addr;//当前EPT页表基地址
4     u64 *sptep;//指向下一级EPT页表的指针/
5     int level;//当前所处的页表级别
6     unsigned index;//对应于addr的表项在当前页表的索引
7 };
```

每一级的遍历通过一个kvm_shadow_walk_iterator进行，其中各个字段的意义已经注明，我们只需要明白初始状态iterator.shadow_addr指向EPT页表基地址，addr是GPA的客户物理页帧号，level是当前iterator所处的级别，其值会随着一层一层的遍历递减。

然后看循环条件

```
1 static bool shadow_walk_okay(struct kvm_shadow_walk_iterator *iterator)
2 {
3     if (iterator->level < PT_PAGE_TABLE_LEVEL)
4         return false;
5
6     iterator->index = SHADOW_PT_INDEX(iterator->addr, iterator->level);
7     iterator->sptep    = ((u64 *)__va(iterator->shadow_addr)) + iterator->index;
8     return true;
9 }
```

循环的条件比较 简单，就是判断是否循环到了最后一级的页表，即iterator.level<1,如果iterator.level递减到0，则本次页表构建过程也完毕了。如果iterator还没有到最后一级，则需要设置iterator的index,这是要寻找的addr在本级页表中对应的页表项索引，然后设置iterator->sptep，指向下一级的页表页。

处理函数：

```
1 static void shadow_walk_next(struct kvm_shadow_walk_iterator *iterator)
2 {
3     return __shadow_walk_next(iterator, *iterator->sptep);
4 }
```

```
 1 static void __shadow_walk_next(struct kvm_shadow_walk_iterator *iterator,
 2                                 u64 spte)
 3 {
 4     if (is_last_spte(spte, iterator->level)) {
 5         iterator->level = 0;
 6         return;
 7     }
 8
 9     iterator->shadow_addr = spte & PT64_BASE_ADDR_MASK;
10     --iterator->level;
11 }
```

正如前面所说，此过程是往后移动iterator的过程，当level为PT_PAGE_TABLE_LEVEL即1的时候，此时应该设置对应的页表指向对应的页了，而不需要再次往后遍历，所以，就直接return，否则，需要往后移动iterator，设置iterator的shadow_addr和level。

总结：

本节比较详细的介绍了EPT的结构和其工作的原理，虽然笔者已经尽最大可能的讲述清楚，但是还是感觉差强人意。写博客时间不长，慢慢来吧，水平有限，文中难免出现差错，有看到的老师还请指正，我们一起进步！

分类: KVM虚拟化技术, linux 内核源码分析

jack.chen
关注 - 12
粉丝 - 44
+加关注

0 0

« 上一篇： 为何要写博客
» 下一篇： PAE 分页模式详解

相关博文：
· KVM的ept机制
· 内存虚拟化
· PAE 分页模式详解
· x64 结构体系下的内存寻址
· 虚拟内存机制
» 更多推荐...

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...

以马内利