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

qemu-kvm内存虚拟化2

2017-04-20

上篇文章对qemu部分的内存虚拟化做了介绍，上篇文章对于要添加的FR，调用了 MEMORY_LISTENER_UPDATE_REGION(frnew, as, Forward, region_add)

```
#define MEMORY_LISTENER_UPDATE_REGION(fr, as, dir, callback)            \
    MEMORY_LISTENER_CALL(callback, dir, (&(MemoryRegionSection) {       \
        .mr = (fr)->mr,                                                 \
        .address_space = (as),                                          \
        .offset_within_region = (fr)->offset_in_region,                 \
        .size = (fr)->addr.size,                                        \
        .offset_within_address_space = int128_get64((fr)->addr.start),  \
        .readonly = (fr)->readonly,                                     \
              }))
```

该宏实际上是另一个宏MEMORY_LISTENER_CALL的封装，在MEMORY_LISTENER_CALL中临时生成一个MemoryRegionSection结构，具体逻辑如下

```
#define MEMORY_LISTENER_CALL(_callback, _direction, _section, _args...) \
    do {                                                                \
        MemoryListener *_listener;                                      \
                                                                        \
        switch (_direction) {                                           \
        case Forward:                                                   \
            QTAILQ_FOREACH(_listener, &memory_listeners, link) {        \
                if (_listener->_callback                                \
                    && memory_listener_match(_listener, _section)) {    \
                    _listener->_callback(_listener, _section, ##_args); \
                }                                                       \
            }                                                           \
            break;                                                      \
        case Reverse:                                                   \
            QTAILQ_FOREACH_REVERSE(_listener, &memory_listeners,        \
                                   memory_listeners, link) {            \
                if (_listener->_callback                                \
                    && memory_listener_match(_listener, _section)) {    \
                    _listener->_callback(_listener, _section, ##_args); \
                }                                                       \
            }                                                           \
            break;                                                      \
        default:                                                        \
            abort();                                                    \
        }                                                               \
    } while (0)
```

这里有个_direction，其实就是遍历方向，因为listener按照优先级从低到高排列，所以这里其实就是确定让谁先处理。Forward就是从前向后，而reverse就是从后向前。还有一个值得注意的是memory_listener_match函数，listener有这对应的AddressSpace，会在其 address_space_filter均指定，注意如果没有指定AddressSpace，那么该listener对所有AddressSpace均适用，否则只适用于其指定的AddressSpace。知道这些，看这些代码就不成问题了。

这样kvm_region_add()函数得到执行，那咱们就从kvm_region_add函数开始，对KVM部分的内存管理做介绍。kvm_region_add()函数是核心listener的添加region的函数，在qemu申请好内存后，针对每个FR，调用了listener的region_add函数。最终需要利用此函数把region信息告知KVM，KVM以此对内存信息做记录。咱们直奔主题，函数核心在static void kvm_set_phys_mem(MemoryRegionSection *section, bool add)函数中，

该函数比较长，咱们还是分段介绍

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)

```
KVMState *s = kvm_state;
    KVMSlot *mem, old;
    int err;
    MemoryRegion *mr = section->mr;
    bool log_dirty = memory_region_is_logging(mr);
    /*是否可写*/
    bool writeable = !mr->readonly && !mr->rom_device;
    bool readonly_flag = mr->readonly || memory_region_is_romd(mr);
    //section中数据的起始偏移
    hwaddr start_addr = section->offset_within_address_space;
    /*section的size*/
    ram_addr_t size = int128_get64(section->size);
    void *ram = NULL;
    unsigned delta;

    /* kvm works in page size chunks, but the function may be called
       with sub-page size and unaligned start address. */
    /*内存对齐后的偏移*/
    delta = TARGET_PAGE_ALIGN(size) - size;
    if (delta > size) {
        return;
    }

    start_addr += delta;
    size -= delta;//这样可以保证size是页对齐的
    size &= TARGET_PAGE_MASK;

    if (!size || (start_addr & ~TARGET_PAGE_MASK)) {
        return;
    }
    /*如果不是rom，则不能进行写操作*/
    if (!memory_region_is_ram(mr)) {
        if (writeable || !kvm_readonly_mem_allowed) {
            return;
        } else if (!mr->romd_mode) {
            /* If the memory device is not in romd_mode, then we actually want
             * to remove the kvm memory slot so all accesses will trap. */
            add = false;
        }
    }
```

第一部分主要是一些基础工作，获取section对应的MR的一些属性，如writeable、readonly_flag。获取section的start_addr和size，其中start_addr就是section中的offset_within_address_space也就是FR中的offset_in_region,接下来对size进行了对齐操作 。如果对应的MR关联的内存并不是作为ram存在，就要进行额外的验证。这种情况如果writeable允许写操作或者kvm不支持只读内存，那么直接返回。

接下来是函数的重点处理部分，即把当前的section转化成一个slot进行添加，但是在此之前需要处理已存在的slot和新的slot的重叠问题，当然如果没有重叠就好办了，直接添加即可。进入while循环

```
 ram = memory_region_get_ram_ptr(mr) + section->offset_within_region + delta;
    /*对重叠部分的处理*/
    while (1) {
        /*查找重叠的部分*/
        mem = kvm_lookup_overlapping_slot(s, start_addr, start_addr + size);
        /*如果没找到重叠，就break*/
        if (!mem) {
            break;
        }
        /*如果要添加区间已经被注册*/
        if (add && start_addr >= mem->start_addr &&
            (start_addr + size <= mem->start_addr + mem->memory_size) &&
            (ram - start_addr == mem->ram - mem->start_addr)) {
            /* The new slot fits into the existing one and comes with
             * identical parameters - update flags and done. */
            kvm_slot_dirty_pages_log_change(mem, log_dirty);
            return;
        }

        old = *mem;

        if (mem->flags & KVM_MEM_LOG_DIRTY_PAGES) {
```

```
            kvm_physical_sync_dirty_bitmap(section);
        }
        /*移除重叠的部分*/
        /* unregister the overlapping slot */
        mem->memory_size = 0;
        err = kvm_set_user_memory_region(s, mem);
        if (err) {
            fprintf(stderr, "%s: error unregistering overlapping slot: %s\n",
                    __func__, strerror(-err));
            abort();
        }

        /* Workaround for older KVM versions: we can't join slots, even not by
         * unregistering the previous ones and then registering the larger
         * slot. We have to maintain the existing fragmentation. Sigh.
         *
         * This workaround assumes that the new slot starts at the same
         * address as the first existing one. If not or if some overlapping
         * slot comes around later, we will fail (not seen in practice so far)
         * - and actually require a recent KVM version. */
         /*如果已有的size小于申请的size，则需要在原来的基础上，添加新的，不能删除原来的再次添加*/
        if (s->broken_set_mem_region &&
            old.start_addr == start_addr && old.memory_size < size && add) {
            mem = kvm_alloc_slot(s);
            mem->memory_size = old.memory_size;
            mem->start_addr = old.start_addr;
            mem->ram = old.ram;
            mem->flags = kvm_mem_flags(s, log_dirty, readonly_flag);

            err = kvm_set_user_memory_region(s, mem);
            if (err) {
                fprintf(stderr, "%s: error updating slot: %s\n", __func__,
                        strerror(-err));
                abort();
            }

            start_addr += old.memory_size;
            ram += old.memory_size;
            size -= old.memory_size;
            continue;
        }

        /* register prefix slot */
        /*new 的start_addr大于old.start_addr，需要补足前面多余的部分*/
        if (old.start_addr < start_addr) {
            mem = kvm_alloc_slot(s);
            mem->memory_size = start_addr - old.start_addr;
            mem->start_addr = old.start_addr;
            mem->ram = old.ram;
            mem->flags =  kvm_mem_flags(s, log_dirty, readonly_flag);

            err = kvm_set_user_memory_region(s, mem);
            if (err) {
                fprintf(stderr, "%s: error registering prefix slot: %s\n",
                        __func__, strerror(-err));
#ifdef TARGET_PPC
                fprintf(stderr, "%s: This is probably because your kernel's " \
                                "PAGE_SIZE is too big. Please try to use 4k " \
                                "PAGE_SIZE!\n", __func__);
#endif
                abort();
            }
        }

        /* register suffix slot */
        /**/
        if (old.start_addr + old.memory_size > start_addr + size) {
            ram_addr_t size_delta;

            mem = kvm_alloc_slot(s);
            mem->start_addr = start_addr + size;
            size_delta = mem->start_addr - old.start_addr;
            mem->memory_size = old.memory_size - size_delta;
            mem->ram = old.ram + size_delta;
            mem->flags = kvm_mem_flags(s, log_dirty, readonly_flag);

            err = kvm_set_user_memory_region(s, mem);
            if (err) {
                fprintf(stderr, "%s: error registering suffix slot: %s\n",
```

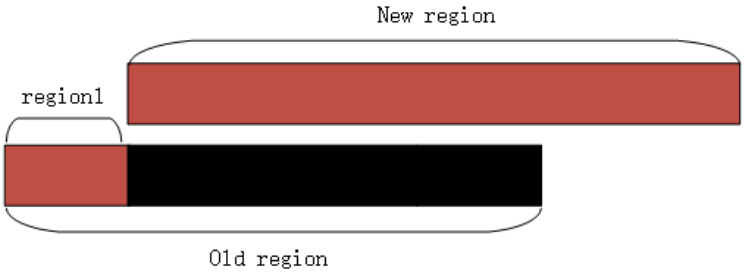
```
                    __func__, strerror(-err));
            abort();
        }
    }
}
```

首先调用了kvm_lookup_overlapping_slot函数找到一个冲突的slot，注意返回结果是按照slot为单位，只要两个地址范围有交叉，就意味着存在冲突，就返回冲突的slot。如果没有，那么直接break，添加新的。否则就根据以下几种情况进行分别处理。其实主要由两种大情况：

1、新的slot完全包含于引起冲突的slot中，并且参数都是一致的。

2、新的slot和引起冲突的slot仅仅是部分交叉。

针对第一种情况，如果flag有变动，则只更新slot的flags,否则，不需要变动。第二种情况，首先要把原来的region　delete掉，具体方式是设置mem->memory_size=0,然后调用kvm_set_user_memory_region()函数。由于 新的region和delete的region不是完全对应的，仅仅是部分交叉，所以就会连带 删除多余的映射，那么接下来的工作就 是分批次弥补映射。如图所示



如图所示，old region是找到的和new region重叠的slot，首次删除把整个slot都删除了，造成了region1部门很无辜的受伤，所以在映射时，要把region1在弥补上。而黑色部分就是实际删除的，这样接下来就可以直接映射new region了，如果多余的部分在后方，也是同样的道理。在把无辜被删除region映射之后，接下来就调用kvm_set_user_memory_region把new slot映射进去。基本思路就是这样。下面看下核心函数kvm_set_user_memory_region

```
static int kvm_set_user_memory_region(KVMState *s, KVMSlot *slot)
{
    struct kvm_userspace_memory_region mem;

    mem.slot = slot->slot;
    mem.guest_phys_addr = slot->start_addr;
    mem.userspace_addr = (unsigned long)slot->ram;
    mem.flags = slot->flags;
    if (s->migration_log) {
        mem.flags |= KVM_MEM_LOG_DIRTY_PAGES;
    }

    if (slot->memory_size && mem.flags & KVM_MEM_READONLY) {
        /* Set the slot size to 0 before setting the slot to the desired
         * value. This is needed based on KVM commit 75d61fbc. */
        mem.memory_size = 0;
        kvm_vm_ioctl(s, KVM_SET_USER_MEMORY_REGION, &mem);
    }
    mem.memory_size = slot->memory_size;
    return kvm_vm_ioctl(s, KVM_SET_USER_MEMORY_REGION, &mem);
}
```

可以看到该函数实现也并不复杂，使用了一个kvm_userspace_memory_region对应，该结构本质上作为参数传递给KVM，只是由于不能共享堆栈，在KVM中需要把该结构复制到内核空间，代码本身没什么难度，只是这里如果是只读的mem,需要调用两次kvm_vm_ioctl，第一次设置mem的size为0.

至于为何这么做，咱们到分析KVM端的时候在做讨论。

KVM接收端在kvm_vm_ioctl()函数中

```
……
case KVM_SET_USER_MEMORY_REGION: {
```

```
        struct kvm_userspace_memory_region kvm_userspace_mem;

        r = -EFAULT;
        if (copy_from_user(&kvm_userspace_mem, argp,
                           sizeof (kvm_userspace_mem)))
            goto out;
        kvm_userspace_mem->flags |= 0x1;
        r = kvm_vm_ioctl_set_memory_region(kvm, &kvm_userspace_mem);
        break;
    }
……
```

可以看到首要任务就是把参数复制到内核，然后调用了kvm_vm_ioctl_set_memory_region()函数。

```
int kvm_vm_ioctl_set_memory_region(struct kvm *kvm,
                   struct kvm_userspace_memory_region *mem)
{
    if (mem->slot >= KVM_USER_MEM_SLOTS)
        return -EINVAL;
    return kvm_set_memory_region(kvm, mem);
}
```

函数检查下slot编号如果超额，那没办法，无法添加，否则调用kvm_set_memory_region()函数。而该函数没做别的，直接调用了__kvm_set_memory_region。该函数比较长，咱们还是分段介绍。开始就是一些常规检查。

```
    if (mem->memory_size & (PAGE_SIZE - 1))
        goto out;
    if (mem->guest_phys_addr & (PAGE_SIZE - 1))
        goto out;
    /* We can read the guest memory with __xxx_user() later on. */
    if ((mem->slot < KVM_USER_MEM_SLOTS) &&
        ((mem->userspace_addr & (PAGE_SIZE - 1)) ||
         !access_ok(VERIFY_WRITE,
            (void  *)(unsigned long)mem->userspace_addr,
           mem->memory_size)))
        goto out;
    if (mem->slot >= KVM_MEM_SLOTS_NUM)
        goto out;
    if (mem->guest_phys_addr + mem->memory_size < mem->guest_phys_addr)
        goto out;
```

如果memory_size 不是页对齐的，则失败；如果mem的客户机物理地址不是页对齐的，也失败；如果slot的id在合法范围内但是用户空间地址不是页对齐的或者地址范围内的不能正常访问，则失败；如果slot id大于等于KVM_MEM_SLOTS_NUM，则失败；最后if我是没看明白啊，怎么可能越加越小呢，size也不可能是负值，该问题再议吧。

```
/*定位到指定slot*/
    slot = id_to_memslot(kvm->memslots, mem->slot);
    base_gfn = mem->guest_phys_addr >> PAGE_SHIFT;
    npages = mem->memory_size >> PAGE_SHIFT;

    r = -EINVAL;
    if (npages > KVM_MEM_MAX_NR_PAGES)
        goto out;
    /*如果npages为0，则设置*/
    if (!npages)
        mem->flags &= ~KVM_MEM_LOG_DIRTY_PAGES;
    /*new  为用户空间传递过来的mem,old为和用户空间mem id一致的mem*/
    new = old = *slot;

    new.id = mem->slot;
    new.base_gfn = base_gfn;
```

```
    new.npages = npages;
    new.flags = mem->flags;

    r = -EINVAL;
    /*如果new 的 npage不为0*/
    if (npages) {
        /*如果old 的npage为0，则创建新的mem*/
        if (!old.npages)
            change = KVM_MR_CREATE;
        /*否则修改已有的mem*/
        else { /* Modify an existing slot. */
            if ((mem->userspace_addr != old.userspace_addr) ||
                (npages != old.npages) ||
                ((new.flags ^ old.flags) & KVM_MEM_READONLY))
                goto out;
            /*如果两个mem映射的基址不同*/
            if (base_gfn != old.base_gfn)
                change = KVM_MR_MOVE;
            /*如果标志位不同则更新标志位*/
            else if (new.flags != old.flags)
                change = KVM_MR_FLAGS_ONLY;
            else { /* Nothing to change. */
                /*都一样的话就什么都不做*/
                r = 0;
                goto out;
            }
        }
    }
    /*如果new的npage为0而old的npage不为0，则需要delete已有的*/
    else if (old.npages) {
        change = KVM_MR_DELETE;
    } else /* Modify a non-existent slot: disallowed. */
        goto out;
```

这里如果检查都通过了，首先通过传递进来的slot的id在kvm维护的slot数组中找到对应的slot结构，此结构可能为空或者为旧的slot。然后获取物理页框号、页面数目。如果页面数目大于KVM_MEM_MAX_NR_PAGES，则失败；如果npages为0，则去除KVM_MEM_LOG_DIRTY_PAGES标志。使用旧的slot对新的slot内容做初始化，然后对new slot做设置，参数基本是从用户空间接收的kvm_userspace_memory_region的参数。然后进入下面的if判断

1、如果npages不为0，表明本次要添加slot此时如果old slot的npages为0，表明之前没有对应的slot，需要添加新的,设置change为KVM_MR_CREATE；如果不为0，则需要先修改已有的slot，注意这里如果old slot和new slot的page数目和用户空间地址必须相等，还有就是两个slot的readonly属性必须一致。如果满足上述条件，进入下面的流程。如果映射的物理页框号不同，则设置change KVM_MR_MOVE，如果flags不同，设置KVM_MR_FLAGS_ONLY，否则，什么都不做。

2、如果npages为0，而old.pages不为0，表明需要删除old slot,设置change为KVM_MR_DELETE。到这里基本是做一些准备工作，确定用户空间要进行的操作，接下来就执行具体的动作了

```
if ((change == KVM_MR_CREATE) || (change == KVM_MR_MOVE)) {
        /* Check for overlaps */
        r = -EEXIST;
        kvm_for_each_memslot(slot, kvm->memslots) {
            if ((slot->id >= KVM_USER_MEM_SLOTS) ||
                (slot->id == mem->slot))
                continue;
            if (!((base_gfn + npages <= slot->base_gfn) ||
                  (base_gfn >= slot->base_gfn + slot->npages)))
                goto out;
        }
    }

    /* Free page dirty bitmap if unneeded */
    if (!(new.flags & KVM_MEM_LOG_DIRTY_PAGES))
        new.dirty_bitmap = NULL;

    r = -ENOMEM;
    if (change == KVM_MR_CREATE) {
        new.userspace_addr = mem->userspace_addr;

        if (kvm_arch_create_memslot(&new, npages))
            goto out_free;
    }
```

```
    /* Allocate page dirty bitmap if needed */
    if ((new.flags & KVM_MEM_LOG_DIRTY_PAGES) && !new.dirty_bitmap) {
        if (kvm_create_dirty_bitmap(&new) < 0)
            goto out_free;
    }
    /*如果用户层请求释放*/
    if ((change == KVM_MR_DELETE) || (change == KVM_MR_MOVE)) {
        r = -ENOMEM;
        slots = kmemdup(kvm->memslots, sizeof(struct kvm_memslots),
                GFP_KERNEL);
        if (!slots)
            goto out_free;
        /*先根据id定位具体的slot*/
        slot = id_to_memslot(slots, mem->slot);
        /*首先设置非法*/
        slot->flags |= KVM_MEMSLOT_INVALID;

        old_memslots = install_new_memslots(kvm, slots, NULL);

        /* slot was deleted or moved, clear iommu mapping */
        kvm_iommu_unmap_pages(kvm, &old);
        /* From this point no new shadow pages pointing to a deleted,
         * or moved, memslot will be created.
         *
         * validation of sp->gfn happens in:
         *      - gfn_to_hva (kvm_read_guest, gfn_to_pfn)
         *      - kvm_is_visible_gfn (mmu_check_roots)
         */
        kvm_arch_flush_shadow_memslot(kvm, slot);
        slots = old_memslots;
    }

    r = kvm_arch_prepare_memory_region(kvm, &new, mem, change);
    if (r)
        goto out_slots;

    r = -ENOMEM;
    /*
     * We can re-use the old_memslots from above, the only difference
     * from the currently installed memslots is the invalid flag.  This
     * will get overwritten by update_memslots anyway.
     */
    if (!slots) {
        slots = kmemdup(kvm->memslots, sizeof(struct kvm_memslots),
                GFP_KERNEL);
        if (!slots)
            goto out_free;
    }

    /*
     * IOMMU mapping:  New slots need to be mapped.  Old slots need to be
     * un-mapped and re-mapped if their base changes.  Since base change
     * unmapping is handled above with slot deletion, mapping alone is
     * needed here.  Anything else the iommu might care about for existing
     * slots (size changes, userspace addr changes and read-only flag
     * changes) is disallowed above, so any other attribute changes getting
     * here can be skipped.
     */
    if ((change == KVM_MR_CREATE) || (change == KVM_MR_MOVE)) {
        r = kvm_iommu_map_pages(kvm, &new);
        if (r)
            goto out_slots;
    }

    /* actual memory is freed via old in kvm_free_physmem_slot below */
    if (change == KVM_MR_DELETE) {
        new.dirty_bitmap = NULL;
        memset(&new.arch, 0, sizeof(new.arch));
    }
    old_memslots = install_new_memslots(kvm, slots, &new);
    kvm_arch_commit_memory_region(kvm, mem, &old, change);
    kvm_free_physmem_slot(&old, &new);
    kfree(old_memslots);
    return 0;
```

这里就根据change来做具体的设置了，如果 KVM_MR_CREATE，则设置new.用户空间地址为新的地址。如果new slot要求KVM_MEM_LOG_DIRTY_PAGES，但是new并没有分配dirty_bitmap，则为其

分配。如果change为KVM_MR_DELETE或者KVM_MR_MOVE，这里主要由两个操作，一是设置对应slot标识为KVM_MEMSLOT_INVALID，更新页表。二是增加slots->generation，撤销iommu mapping。接下来对于私有映射的话(memslot->id >= KVM_USER_MEM_SLOTS)，如果是要创建，则需要手动建立映射。

接下来确保slots不为空，如果是KVM_MR_CREATE或者KVM_MR_MOVE，就需要重新建立映射，使用kvm_iommu_map_pages函数 ，而如果是KVM_MR_DELETE，就没必要为new设置dirty_bitmap，并对其arch字段的结构清零。最终都要执行操作install_new_memslots，不过当为delete操作时，new的memory size为0，那么看下该函数做了什么。

```
static struct kvm_memslots *install_new_memslots(struct kvm *kvm,
        struct kvm_memslots *slots, struct kvm_memory_slot *new)
{
    struct kvm_memslots *old_memslots = kvm->memslots;

    update_memslots(slots, new, kvm->memslots->generation);
    rcu_assign_pointer(kvm->memslots, slots);
    kvm_synchronize_srcu_expedited(&kvm->srcu);
    kvm_arch_memslots_updated(kvm);
    return old_memslots;
}
```

这里核心操作在update_memslots里，如果是添加新的（create or move），那么new 的memory size肯定不为0，则根据new的id，在kvm维护的slot 数组中找到对应的slot，然后一次性吧new的内容赋值给old slot.如果页面数目不一样，则需要进行排序。如果是删除操作，new的memorysize 为0，这里就相当于把清空了一个slot。update之后就更改kvm->memslots，该指针是受RCU机制保护的，所以才能直接修改，需要先分配好，调用API修改。最后再刷新MMIO页表项。

```
void update_memslots(struct kvm_memslots *slots, struct kvm_memory_slot *new,
             u64 last_generation)
{
    if (new) {
        int id = new->id;
        struct kvm_memory_slot *old = id_to_memslot(slots, id);
        unsigned long npages = old->npages;

        *old = *new;
            /*如果是删除操作，那么new.npages就是0*/
        if (new->npages != npages)
            sort_memslots(slots);
    }

    slots->generation = last_generation + 1;
}
```

参考：

linux 3.10.1源码

分类: KVM虚拟化技术, linux 内核源码分析, qemu

好文要顶 关注我 收藏该文

jack.chen
关注 - 12
粉丝 - 44
+加关注

0 0

« 上一篇： qemu进程页表和EPT的同步问题
» 下一篇： linux页缓存

posted @ 2017-04-23 20:23 jack.chen Views(1438) 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】精品问答：精品问答：Python 技术 1000 问
【推荐】精品问答：Java 技术 1000 问

相关博文：
· [原] KVM 虚拟化原理探究（2） — QEMU启动过程
· Qemu创建KVM虚拟机内存初始化流程
· qemu-kvm内存虚拟化1
· QEMU,KVM及QEMU-KVM介绍
· qemu kvm 虚拟化
» 更多推荐...

2019 Flink Forward 大会最全视频来了！5大专题不容错过

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

以马内利