

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

qemu-kvm内存虚拟化1

2017-04-18

记得很早之前分析过KVM内部内存虚拟化的原理, 仅仅知道KVM管理一个个slot并以此为基础转换GPA到HVA, 却忽略了qemu端最初内存的申请, 而今有时间借助于qemu源码分析下qemu在最初是如何申请并管理虚拟机内存的, 坦白讲, 还真挺复杂的。


一、概述

qemu-kvm 模型下的虚拟化引擎, 内存虚拟化部分要说简单也挺简单, 在虚拟机启动时, 有qemu在qemu进程地址空间申请内存, 即内存的申请是在用户空间完成的。通过kvm提供的API, 把地址信息注册到KVM中, 这样KVM中维护有虚拟机相关的slot, 这些slot构成了一个完成的虚拟机物理地址空间。slot中记录了其对应的HVA, 页面数、起始GPA等, 利用它可以把一个GPA转化成HVA, 想到这一点自然和硬件虚拟化下的地址转换机制EPT联系起来, 不错, 这正是KVM维护EPT的技术核心。整个内存虚拟化可以分为两部分: qemu部分和kvm部分。qemu完成内存的申请, kvm实现内存的管理。看起来简单, 但是内部实现机制也并非那么简单。本文重点介绍qemu部分。

二、涉及数据结构

qemu中内存管理的数据结构主要涉及: MemoryRegion、AddressSpace、FlatView、FlatRange、MemoryRegionSection、RAMList、RAMBlock、KVMSlot、kvm\_userspace\_memory\_region等

这几个数据结构的确不太容易滤清, 一下是个人的一些见解。 怎么可以把qemu层的内存管理再分为三个层次, MemoryRegion就位于顶级抽象层或者说比较偏向于host端, qemu中两个全局的MemoryRegion, 分别是system\_memory和system\_io, 不过两个均是以指针的形式存在, 在地址空间的时候才会对其分配具体的内存并初始化。MemoryRegion负责管理host的内存, 理论上是树状结构, 但是实际上根据代码来看仅仅有两层,



```
struct MemoryRegion {
    /* All fields are private - violators will be prosecuted */
    const MemoryRegionOps *ops;
    const MemoryRegionIOMMUOps *iommu_ops;
    void *opaque;
    struct Object *owner;
    MemoryRegion *parent; //父区域指针
    Int128 size; //区域的大小
    hwaddr addr;
    void (*destructor)(MemoryRegion *mr);
    ram_addr_t ram_addr; //区域关联的ram地址, GPA
    bool subpage;
    bool terminates;
    bool romd_mode;
    bool ram; //是否是ram
    bool readonly; /* For RAM regions */
    bool enabled;
    bool rom_device;
    bool warning_printed; /* For reservations */
    bool flush_coalesced_mmio;
    MemoryRegion *alias;
    hwaddr alias_offset;
    int priority;
    bool may_overlap;
    QTAILQ_HEAD(subregions, MemoryRegion) subregions; //子区域链表头
    QTAILQ_ENTRY(MemoryRegion) subregions_link; //子区域链表节点
    QTAILQ_HEAD(coalesced_ranges, CoalescedMemoryRange) coalesced;
    const char *name;
    uint8_t dirty_log_mask;
    unsigned ioeventfd_nb;
    MemoryRegionIoeventfd *ioeventfds;
    NotifierList iommu_notify;
};
```



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。  
  
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)

MemoryRegion结构如上，相关注释已经列举，其中parent指向父MR，默认是NULL，size表示区域的大小；默认是64位下的最大地址；ram\_addr比较重要，是区域关联的客户机物理地址空间的偏移，也就是客户机物理地址。alias表明该区域是某一类型的区域（先这么说吧），这么说不知是否合适，实际上虚拟机的ram申请时时一次性申请的一个完成的ram，记录在一个MR中，之后又对此ram按照size进行了划分，形成subregion,而subregion 的alias便指向原始的MR，而alias\_offset 便是在原始ram中的偏移。对于系统地址空间的ram，会把刚才得到的subregion注册到系统中，父MR是刚才提到的全局MR system\_memory,subregions\_link是链表节点。前面提到，实际关联host内存的是subregion->alias指向的MR，其ram\_addr是该MR在虚拟机的物理内存中的偏移，具体是由RAMBlock->offset获得的，RAMBlock最直接的接触host的内存，看下其结构

```
typedef struct RAMBlock {
    struct MemoryRegion *mr;
    uint8_t *host; /* block关联的内存, HVA */
    ram_addr_t offset; /* 在vm物理内存中的偏移 GPA */
    ram_addr_t length; /* block的长度 */
    uint32_t flags;
    char idstr[256];
    /* Reads can take either the iothread or the ramlist lock.
     * Writes must take both locks.
     */
    QTAILQ_ENTRY(RAMBlock) next;
    int fd;
} RAMBlock;
```

仅有的几个字段意义比较明确，理论上一个RAMBlock就代表host的一段虚拟 内存，host指向申请的ram的虚拟地址,是HVA。所有的RAMBlock通过next字段连接起来，表头保存在一个全局的RAMList结构中，但是根据代码来看，原始MR分配内存时分配的是一整块block，之所以这样做也许是为了扩展用吧！！RAMList中有个字段mru\_block指针，指向最近使用的block，这样需要遍历所有的block时，先判断指定的block是否是mru\_block，如果不是再进行遍历从而提高效率。

qemu的内存管理在交付给KVM管理时，中间又加了一个抽象层，叫做address\_space.如果说MR管理的host的内存，那么address\_space管理的更偏向于虚拟机。正如其名字所描述的，它是管理地址空间的，qemu中有几个全局的AddressSpace，address\_space\_memory和address\_space\_io，很明显一个是管理系统地址空间，一个是IO地址空间。它是如何进行管理的呢？展开下AddressSpace的结构；

```
struct AddressSpace {
    /* All fields are private. */
    char *name;
    MemoryRegion *root;
    struct FlatView *current_map; /* 对应的flatview */
    int ioeventfd_nb;
    struct MemoryRegionIoeventfd *ioeventfds;
    struct AddressSpaceDispatch *dispatch;
    struct AddressSpaceDispatch *next_dispatch;
    MemoryListener dispatch_listener;

    QTAILQ_ENTRY(AddressSpace) address_spaces_link;
};
```

对于该结构，源码的注释或许更能解释：AddressSpace: describes a mapping of addresses to #MemoryRegion objects，很明显是把MR映射到虚拟机的物理地址空间。root指向根MR，对于address\_space\_memory来讲，root指向系统全局的MR system\_memory,current\_map指向一个FlatView结构，其他的字段咱们先暂时忽略，所有的AddressSpace通过结构中的address\_spaces\_link连接成链表，表头保存在全局的AddressSpace结构中。FlatView管理MR展开后得到的所有FlatRange，看下FlatView

```
struct FlatView {
    unsigned ref; /* 引用计数，为0时就销毁 */
    FlatRange *ranges; /* 对应的flatrange数组 */
    unsigned nr; /* flatrange 的数目 */
    unsigned nr_allocated; /* 当前数组的项数 */
};
```

各个字段的意义就不说了，ranges是一个数组，记录FlatView下所有的FlatRange，每个FlatRange对应一段虚拟机物理地址区间，各个FlatRange不会重叠，按照地址的顺序保存在数组中。FlatRange结构如下

```
struct FlatRange {
    MemoryRegion *mr; /*指向所属的MR*/
    hwaddr offset_in_region; /*在MR中的offset*/
    AddrRange addr; /*本FR代表的区间*/
    uint8_t dirty_log_mask;
    bool romd_mode;
    bool readonly; /*是否是只读*/
};
```

具体的范围由一个AddrRange结构描述，其描述了地址和大小，offset\_in\_region表示该区间在全局的MR中的offset，根据此可以进行GPA到HVA的转换，mr指向所属的MR。

到此为止，负责管理的结构基本就介绍完毕，剩余几个主要起中介的作用，MemoryRegionSection对应于FlatRange，一个FlatRange代表一个物理地址空间的片段，但是其偏向于address-space，而MemoryRegionSection则在MR端显示的表明了分片，其结构如下

```
struct MemoryRegionSection {
    MemoryRegion *mr; /*所属的MemoryRegion*/
    AddressSpace *address_space; /*region关联的AddressSpace*/
    hwaddr offset_within_region; /*在region内部的偏移*/
    Int128 size; /*section的大小*/
    hwaddr offset_within_address_space; /*首个字节的地址在section中的偏移*/
    bool readonly; /*是否是只读*/
};
```

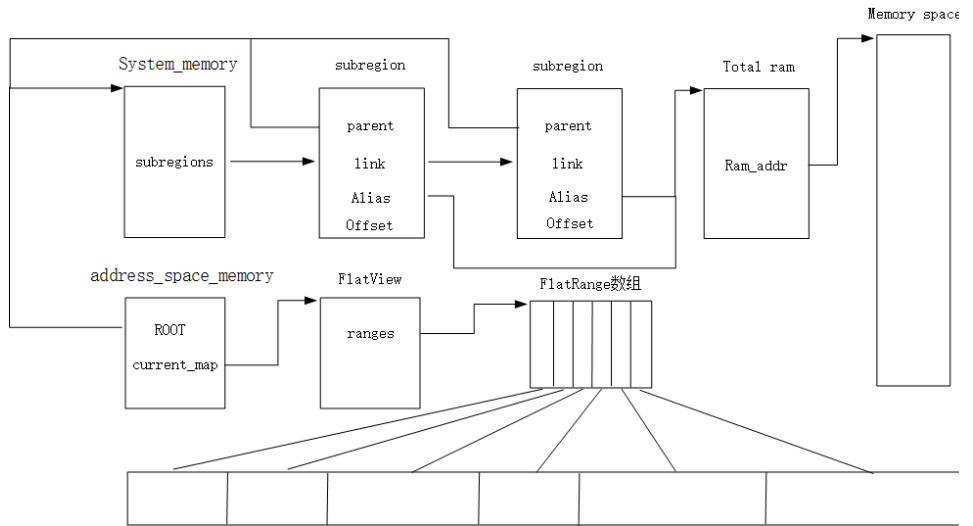
其中注意两个偏移，offset\_within\_region和offset\_within\_address\_space。前者描述的是该section在整个MR中的偏移，一个address\_space可能由多个MR构成，因此该offset是局部的。而offset\_within\_address\_space是在整个地址空间中的偏移，是全局的offset。

KVMSlot也是一个中介，只不过更加接近kvm了，

```
typedef struct KVMSlot
{
    hwaddr start_addr; /*客户机物理地址 GPA*/
    ram_addr_t memory_size; /*内存大小*/
    void *ram; /*HVA qemu用户空间地址*/
    int slot; /*slot编号*/
    int flags;
} KVMSlot;
```

kvm\_userspace\_memory\_region是和kvm共享的一个结构，说共享不太恰当，但是其实最终作为参数给kvm使用的，kvm获取控制权后，从栈中复制该结构到内核，其中字段意思很简单，不在赘述。

整体布局大致如图所示



### 三、具体实现机制

qemu部分的内存申请流程上可以分为三小部分，分成三小部分主要是在看代码的时候觉得这三部分耦合性不是很大，相对而言比较独立。众所周知，qemu起始于vl.c中的main函数，那么这三部分也按照在main函数中的调用顺序分别介绍。

#### 3.1 回调函数的注册

涉及函数：configure\_accelerator() --> kvm\_init() --> memory\_listener\_register ()

这里所说的accelerator在这里就是kvm，初始化函数自然调用了kvm\_init，该函数主要完成对kvm的初始化，包括一些常规检查如CPU个数、kvm版本等，还会通过ioctl和内核交互创建kvm结构，这些并非本文重点，不在赘述。在kvm\_init函数的结尾调用了memory\_listener\_register

```
memory_listener_register(&kvm_memory_listener, &address_space_memory);
memory_listener_register(&kvm_io_listener, &address_space_io);
```

通过memory\_listener\_register函数，针对地址空间注册了listener，listener本身是一组函数表，当地址空间发生变化的时候，会调用到listener中的相应函数，从而保持内核和用户空间的内存信息的一致性。虚拟机包含有两个地址空间address\_space\_memory和address\_space\_io，很容易理解，正常系统也包含系统地址空间和IO地址空间。

memory\_listener\_register函数不复杂，咱们看下

```
void memory_listener_register(MemoryListener *listener, AddressSpace *filter)
{
    MemoryListener *other = NULL;
    AddressSpace *as;

    listener->address_space_filter = filter;
    /*如果listener为空或者当前listener的优先级大于最后一个listener的优先级，则可以直接插入*/
    if (QTAILQ_EMPTY(&memory_listeners)
        || listener->priority >= QTAILQ_LAST(&memory_listeners,
                                              memory_listeners)->priority) {
        QTAILQ_INSERT_TAIL(&memory_listeners, listener, link);
    } else {
        /*listener按照优先级升序排列*/
        QTAILQ_FOREACH(other, &memory_listeners, link) {
            if (listener->priority < other->priority) {
                break;
            }
        }
        /*插入listener*/
        QTAILQ_INSERT_BEFORE(other, listener, link);
    }
    /*全局address_spaces-->as*/
    /*对于每个address_spaces，设置listener*/
    QTAILQ_FOREACH(as, &address_spaces, address_spaces_link) {
        listener_add_address_space(listener, as);
    }
}
```

系统中可以存在多个listener，listener之间有着明确的优先级关系，通过链表进行组织，链表头是全局的memory\_listeners。函数中，如果memory\_listeners为空或者当前listener的优先级大于最后一个listener的优先级，即直接把当前listener插入。否则需要挨个遍历链表，找到合适的位置。具体按照优先级升序查找。在函数最后还针对每个address\_space，调用listener\_add\_address\_space函数，该函数对其对应的address\_space管理的flatrange向KVM注册。当然，实际上此时address\_space尚未经过初始化，所以这里的循环其实是空循环。

### 3.2 Address\_Space的初始化

涉及函数：cpu\_exec\_init\_all() memory\_map\_init()

在第一节中已经注册了listener，但是addressspace尚未初始化，本节就介绍下其初始化流程。从上节的configure\_accelerator()函数往下走，会执行cpu\_exec\_init\_all()函数，该函数主要初始化了IO地址空间和系统地址空间。memory\_map\_init()函数初始化系统地址空间，有一个全局的MemoryRegion指针system\_memory指向该区域的MemoryRegion结构。

```
static void memory_map_init(void)
{
    /*为system_memory分配内存*/
    system_memory = g_malloc(sizeof(*system_memory));

    assert(ADDR_SPACE_BITS <= 64);

    memory_region_init(system_memory, NULL, "system",
                       ADDR_SPACE_BITS == 64 ?
                       UINT64_MAX : (0x1ULL << ADDR_SPACE_BITS));
    /*初始化全局的地址_space_memory*/
    address_space_init(&address_space_memory, system_memory, "memory");

    system_io = g_malloc(sizeof(*system_io));
    memory_region_init_io(system_io, NULL, &unassigned_io_ops, NULL, "io",
                          65536);
    address_space_init(&address_space_io, system_io, "I/O");

    memory_listener_register(&core_memory_listener, &address_space_memory);
    if (tcg_enabled()) {
        memory_listener_register(&tcg_memory_listener, &address_space_memory);
    }
}
```

所以在函数起始，就对system\_memory分配了内存，然后调用了memory\_region\_init函数对其进行初始化，其中size设置为整个地址空间：如果是64位就是 $2^{64}$ 。接着调用了address\_space\_init函数对address\_space\_memory进行了初始化。

```
void address_space_init(AddressSpace *as, MemoryRegion *root, const char *name)
{
    if (QTAILQ_EMPTY(&address_spaces)) {
        memory_init();
    }

    memory_region_transaction_begin();
    /*指定address_space_memory的root为system_memory*/
    as->root = root;
    /*创建并关联了一个FlatView*/
    as->current_map = g_new(FlatView, 1);
    /*初始化FlatView*/
    flatview_init(as->current_map);
    as->ioeventfd_nb = 0;
    as->ioeventfds = NULL;
    /*把address_space_memory插入全局链表*/
    QTAILQ_INSERT_TAIL(&address_spaces, as, address_spaces_link);
    as->name = g_strdup(name ? name : "anonymous");
    address_space_init_dispatch(as);
    memory_region_update_pending |= root->enabled;
    memory_region_transaction_commit();
}
```

函数主要做了以下几个工作，设置addressSpace和MR的关联，并初始化对应的FlatView,设置其名称。最后把address\_space\_memory加入到全局的地址\_spaces链表中，最后调用memory\_region\_transaction\_commit()提交本次修改，关于memory\_region\_transaction\_commit后imianzai做论述。回到memory\_map\_init()函数中，接下来按照同样的模式对IO区域system\_io和IO地址空间address\_space\_io做了初始化。

### 3.3 实际内存的分配

前面注册listener也好，或是初始化addressspace也好，实际上均没有对应的物理内存。顺着main函数往下走，会调用到machine—>init，实际上对应于pc\_init1函数，在该函数中有pc\_memory\_init(函数对实际的内存做了分配。我们直接从pc\_memory\_init()函数开始



```
FWCfgState *pc_memory_init(MemoryRegion *system_memory,
                           const char *kernel_filename,
                           const char *kernel_cmdline,
                           const char *initrd_filename,
                           ram_addr_t below_4g_mem_size,
                           ram_addr_t above_4g_mem_size,
                           MemoryRegion *rom_memory,
                           MemoryRegion **ram_memory,
                           PcguestInfo *guest_info)
{
    int linux_boot, i;
    MemoryRegion *ram, *option_rom_mr;
    MemoryRegion *ram_below_4g, *ram_above_4g;
    FWCfgState *fw_cfg;

    linux_boot = (kernel_filename != NULL);

    /* Allocate RAM. We allocate it as a single memory region and use
     * aliases to address portions of it, mostly for backwards compatibility
     * with older qemus that used qemu_ram_alloc().
     */
    ram = g_malloc(sizeof(*ram));
    //分配具体的内存
    memory_region_init_ram(ram, NULL, "pc.ram",
                          below_4g_mem_size + above_4g_mem_size);
    //那mr中的name设置进block
    vmstate_register_ram_global(ram);
    *ram_memory = ram;
    ram_below_4g = g_malloc(sizeof(*ram_below_4g));
    /*对整体ram进行划分*/
    memory_region_init_alias(ram_below_4g, NULL, "ram-below-4g", ram,
                            0, below_4g_mem_size);

    /******/
    memory_region_add_subregion(system_memory, 0, ram_below_4g);
    e820_add_entry(0, below_4g_mem_size, E820_RAM);
    if (above_4g_mem_size > 0) {
        ram_above_4g = g_malloc(sizeof(*ram_above_4g));
        memory_region_init_alias(ram_above_4g, NULL, "ram-above-4g", ram,
                                below_4g_mem_size, above_4g_mem_size);
        memory_region_add_subregion(system_memory, 0x100000000ULL,
                                    ram_above_4g);
        e820_add_entry(0x100000000ULL, above_4g_mem_size, E820_RAM);
    }

    /* Initialize PC system firmware */
    pc_system_firmware_init(rom_memory, guest_info->isapc_ram_fw);

    option_rom_mr = g_malloc(sizeof(*option_rom_mr));
    memory_region_init_ram(option_rom_mr, NULL, "pc.rom", PC_ROM_SIZE);
    vmstate_register_ram_global(option_rom_mr);
    memory_region_add_subregion_overlap(rom_memory,
                                       PC_ROM_MIN_VGA,
                                       option_rom_mr,
                                       1);

    fw_cfg = bochs_bios_init();
    rom_set_fw(fw_cfg);

    if (linux_boot) {
```

```

        load_linux(fw_cfg, kernel_filename, initrd_filename, kernel_cmdline,
below_4g_mem_size);
    }

    for (i = 0; i < nb_option_roms; i++) {
        rom_add_option(option_rom[i].name, option_rom[i].bootindex);
    }
    guest_info->fw_cfg = fw_cfg;
    return fw_cfg;
}

```

从总体上来讲，该函数主要完成了三个工作：分配全局ram（一整个memory region），然后根据below\_4g\_mem\_size、above\_4g\_mem\_size分别对ram进行划分，形成子MR，并注册子MR到root MR system\_memory 的subregions链表中。最后需要调用memory\_region\_transaction\_commit()函数提交修改。具体而言，分配全局ram由memory\_region\_init\_ram()完成，

```

void memory_region_init_ram(MemoryRegion *mr,
                            Object *owner,
                            const char *name,
                            uint64_t size)
{
    memory_region_init(mr, owner, name, size);
    mr->ram = true;
    mr->terminates = true;
    mr->destructor = memory_region_destructor_ram;
    //为MemoryRegion分配ram 实际上是block.offset,指的是客户机物理地址空间的偏移即GPA
    mr->ram_addr = qemu_ram_alloc(size, mr);
}

```

该函数实现很简单，首先调用memory\_region\_init对MR做初始化。然后进行简单的设置，最重要的还是最后的qemu\_ram\_alloc，咱们重点看下这个函数，该函数不过是qemu\_ram\_alloc\_from\_ptr函数的封装，该函数围绕RAMBlock结构，在函数起始对size进行页对齐，然后申请一个RAMBlock结构，之后对其字段进行一些设置如指定MR 和offset，offset理论上需要调用find\_ram\_offset在已有的RAMBlock中找到一个可用的。但是此时实际上还没有其他block，所以这里应该直接返回0的。本部分最重要的莫过于设置其host了，其对应的宿主虚拟地址空间的虚拟地址。在支持大页的情况下调用file\_ram\_alloc()函数进行分配，否则调用phys\_mem\_alloc()函数进行分配。二者分配其实都是利用mmap分配的，但是在支持大页的情况下需要传递参数mem\_path,所以需要两个函数。在非大页的情况下，分配好内存尝试和相邻的block合并下。最后把block插入到全局的链表ram\_list中，只是block保持从大到小的顺序。

经过上面的初始化，我们得到一个完整的ram，下面的vmstate\_register\_ram\_global仅仅是吧MR的名字设置到对应的block中。此为全局的MR，根据ram\_below\_4g和ram\_above\_4g，全局的MR被划分为两部分，形成两个子MR。memory\_region\_init\_alias()函数便对两个子MR进行初始化，这里就不在进行内存的申请，主要是设置alias和alias\_offset字段，代码如下。

```

void memory_region_init_alias(MemoryRegion *mr,
                              Object *owner,
                              const char *name,
                              MemoryRegion *orig,
                              hwaddr offset,
                              uint64_t size)
{
    memory_region_init(mr, owner, name, size);
    memory_region_ref(orig);
    mr->destructor = memory_region_destructor_alias;
    mr->alias = orig;
    mr->alias_offset = offset;
}

```

在初始化完毕，需要调用memory\_region\_add\_subregion()函数把子MR注册到全局的system\_memory。该函数实现是比较简单的，但是需要重点介绍下，因为这里执行了关键的更新操作。函数核心交给memory\_region\_add\_subregion\_common()函数实现，该函数同样比较简单，设置subregion->parent和system\_memory的关联，设置其addr字段为offset，即在全局MR中的偏移。然后就按照优先级顺序吧subregion插入到system\_memory的subregions链表中，这些都比较简单，目

前我们添加了区域，地址空间已经发生变化，自然要把变化和KVM进行同步，这一工作由memory\_region\_transaction\_commit()实现。

```
void memory_region_transaction_commit(void)
{
    AddressSpace *as;

    assert(memory_region_transaction_depth);
    --memory_region_transaction_depth;
    if (!memory_region_transaction_depth && memory_region_update_pending) {
        memory_region_update_pending = false;
        MEMORY_LISTENER_CALL_GLOBAL(begin, Forward);
        /*更新各个addressspace 拓扑结构*/
        QTAILQ_FOREACH(as, &address_spaces, address_spaces_link) {
            address_space_update_topology(as);
        }

        MEMORY_LISTENER_CALL_GLOBAL(commit, Forward);
    }
}
```

可以看到，这里listener的作用就凸显出来了。对于每个address\_space，调用address\_space\_update\_topology()执行更新。里面涉及两个重要的函数generate\_memory\_topology和address\_space\_update\_topology\_pass。前者对于一个给定的MR，生成其对应的FlatView，而后者则根据oldview和newview对当前视图进行更新。我们还是看下address\_space\_update\_topology函数代码

```
static void address_space_update_topology(AddressSpace *as)
{
    FlatView *old_view = address_space_get_flatview(as);
    /*根据AddressSpace对应的MR生成一个新的FlatView*/
    FlatView *new_view = generate_memory_topology(as->root);

    address_space_update_topology_pass(as, old_view, new_view, false);
    address_space_update_topology_pass(as, old_view, new_view, true);

    qemu_mutex_lock(&flat_view_mutex);
    flatview_unref(as->current_map);
    /*设置新的FlatView*/
    as->current_map = new_view;
    qemu_mutex_unlock(&flat_view_mutex);

    /* Note that all the old MemoryRegions are still alive up to this
     * point. This relieves most MemoryListeners from the need to
     * ref/unref the MemoryRegions they get---unless they use them
     * outside the iothread mutex, in which case precise reference
     * counting is necessary.
     */
    flatview_unref(old_view);

    address_space_update_ioeventfds(as);
}
```

在获取了新旧两个FlatView之后，调用了两次address\_space\_update\_topology\_pass()函数，首次调用重在删除原来的，而后者重在添加。之后设置as->current\_map = new\_view。并对old\_view减少引用，当引用计数为1时会被删除。接下来重点在两个地方：1、如何根据一个MR获取对应的FlatView；2、如何对旧的FlatView进行更新。

前者自然少不了分析generate\_memory\_topology函数

```
static FlatView *generate_memory_topology(MemoryRegion *mr)
{
    FlatView *view;
    /*mr是system_memory*/
}
```



```

view = g_new(FlatView, 1);
flatview_init(view);
/*addrrange_make生成一个0-2^64的地址空间*/
if (mr) {
    /*最初是让MR 按照基址为0映射到地址空间中*/
    render_memory_region(view, mr, intl28_zero(),
                        addrrange_make(intl28_zero(), intl28_2_64()), false);
}
flatview_simplify(view);

return view;
}

```

首先申请一个FlatView结构，并对其进行初始化，然后调用render\_memory\_region函数实现核心功能，最后还调用flatview\_simplify尝试合并相邻的FlatRange.static void render\_memory\_region(FlatView \*view,MemoryRegion \*mr,Intl28 base,AddrRange clip,bool readonly)是一个递归函数，参数中，view表示当前FlatView,mr最初为system\_memory即全局的MR，base起初为0表示从地址空间的其实开始，clip最初为一个完整的地址空间。readonly标识是否只读。

```

static void render_memory_region(FlatView *view,
                                MemoryRegion *mr,
                                Intl28 base,
                                AddrRange clip,
                                bool readonly)
{
    MemoryRegion *subregion;
    unsigned i;
    hwaddr offset_in_region;
    Intl28 remain;
    Intl28 now;
    FlatRange fr;
    AddrRange tmp;

    if (!mr->enabled) {
        return;
    }

    intl28_addto(&base, intl28_make64(mr->addr));
    readonly |= mr->readonly;
    /*获得当前MR的地址区间范围*/
    tmp = addrrange_make(base, mr->size);
    /*判断目标MR和clip是否有交叉，即MR应该落在clip的范围内*/
    if (!addrrange_intersects(tmp, clip)) {
        return;
    }
    /*缩小clip到交叉的部分*/
    clip = addrrange_intersection(tmp, clip);
    /*子区域才有alias字段*/
    if (mr->alias) {
        intl28_subfrom(&base, intl28_make64(mr->alias->addr));
        intl28_subfrom(&base, intl28_make64(mr->alias->offset));
        /*这里base应该为subregion对应的alias中的区间基址*/
        render_memory_region(view, mr->alias, base, clip, readonly);
        return;
    }

    /* Render subregions in priority order. */
    QTAILQ_FOREACH(subregion, &mr->subregions, subregions_link) {
        render_memory_region(view, subregion, base, clip, readonly);
    }

    if (!mr->terminates) {
        return;
    }

    /*offset_in_region is distance between clip.start and base */
    /*clip.start表示地址区间的起始，base为本次映射的基址，差值就为offset*/
    offset_in_region = intl28_get64(intl28_sub(clip.start, base));
    /*开始映射时，clip表示映射的区间范围，base作为一个移动指导每个FR的映射，remain表示clip总还没映射的大小*/
    /*最初base=clip.start */
}

```

```

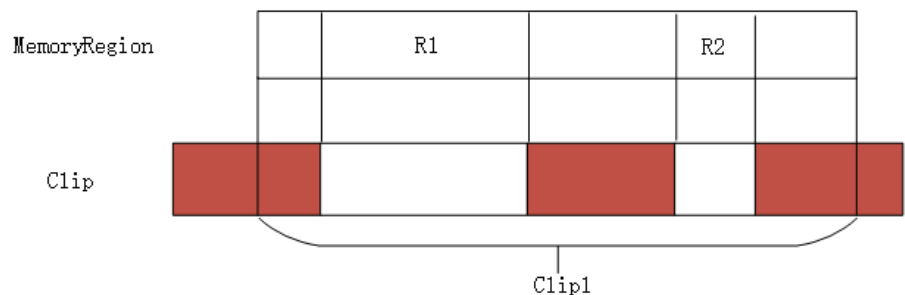
base = clip.start;
remain = clip.size;

fr.mr = mr;
fr.dirty_log_mask = mr->dirty_log_mask;
fr.romd_mode = mr->romd_mode;
fr.readonly = readonly;

/* Render the region itself into any gaps left by the current view. */
for (i = 0; i < view->nr && int128_nz(remain); ++i) {
    /*if base > addrrange_end(view->ranges[i].addr) 即大于一个range的end*/
    if (int128_ge(base, addrrange_end(view->ranges[i].addr))) {
        continue;
    }
    /*如果base <= view->ranges[i].addr.start*/
    /*now 表示已经存在的FR 或者本次填充的FR的长度*/
    if (int128_lt(base, view->ranges[i].addr.start)) {
        now = int128_min(remain,
                        int128_sub(view->ranges[i].addr.start, base));
        /*fr.offset_in_region表示在整个地址空间中的偏移*/
        fr.offset_in_region = offset_in_region;
        fr.addr = addrrange_make(base, now);
        flatview_insert(view, i, &fr);
        ++i;
        int128_addto(&base, now);
        offset_in_region += int128_get64(now);
        int128_subfrom(&remain, now);
    }
    /*if base > rang[i].start, means overlap exists, need escape*/
    now = int128_sub(int128_min(int128_add(base, remain),
                                addrrange_end(view->ranges[i].addr)),
                    base);
    int128_addto(&base, now);
    offset_in_region += int128_get64(now);
    int128_subfrom(&remain, now);
}
/*如果还有剩下的clip没有映射, 则下面不会在发生冲突, 直接一次性的映射完成*/
if (int128_nz(remain)) {
    fr.offset_in_region = offset_in_region;
    fr.addr = addrrange_make(base, remain);
    flatview_insert(view, i, &fr);
}
}
}

```

在此必须清楚MR和clip的意义，即该函数实现的是把MR的区域填充clip空间。填充的方式就是生成一个FlatRange，并交由FlatView管理。基本管理逻辑理论上如图所示



棕色部分意味着已经存在的映射，这种情况下只需要把R1和R2映射进来即可。最终FlatView中的FlatRange按照在物理地址空间的布局，依次排列。但是按照实际情况来讲，实际上传递进来的MR是整个地址空间system\_space,所以像图中那样比较复杂的格局应该基本不会出现。不过咱们还是根据代码来看。首先获取了当前MR的区间范围，以base为起点。我们目的是要把MR的区间映射进clip中，所以如果两者没有交叉，那么无法完成映射。接着设置clip为二者地址区间重叠的部分，以图中所示，clip就成了clip1所标的范围。如果当前MR是某个subregion,则需要对其原始的MR进行展开，因为具体的信息都保存在原始的MR中。但是全局MR system\_memory作为参数传递进来，那么这里mr->alias为NULL，所以到了下面对system\_memory的每个subregion均进行展开。就这样再次进入render\_memory\_region函数的时候，MR为某个subregion，clip也为subregion对应的区域和原始clip的交集，由于其mr->alia指向原始MR，进入if判断，对原始MR的对应区间进行展开，再次调用render\_memory\_region。这一次就要进行真正的展开操作了，即生成对应的FlatRange。

顺着函数往下走，涉及几个变量这里先介绍下，offset\_in\_region为对应MR在全局地址空间中的偏移，base为一个移动指针，指向当前映射的小区间，指导每个FR的映射。now当前已经映射的FR的长度，有

两种可能，第一可能是当前映射的FR，第二可能是已经映射的FR。remain表示当前clip中剩下的未映射的部分（不考虑已经存在的FR），有了这些再看下面的代码就不吃力了。

核心的工作起始于一个for循环，循环的条件是 `view->nr && int128_nz(remain)`，表示当前还有未遍历的FR并且remain还有剩余。循环中如果MR base的值大于或者等于当前FR的end,则继续往后遍历FR，否则进入下面，如果base小于当前FR的start，则表明base到start之间的区间还没有映射，则为其进行映射，now表示要映射的长度，取remain和`int128_sub(view->ranges[i].addr.start, base)`之间的最小值，后者表示下一个FR的start和base之间的差值，当然按照clip为准。接下来就没难度了，设置FR的`offset_in_region`和`addr`,然后调用`flatview_insert`插入到FlatView的FlatRange数组中。不过由于FR按照地址顺序排列，如果插入位置靠前，则需要移动较多的项，不知道为何不用链表实现。下面就很自然了移动base，增加`offset_in_region`，减少remain等操作。出了if,此时base已经和FR的start对齐，所以还需要略过当前FR。就这么一直映射下去。

出了for循环，如果remain不为0，则表明还有没有映射的，但是现在已经没有已经存在的FR了，所以不会发生冲突，直接把remain直接映射成一个FR即可。

按照这个思路，把所有的subregion都映射下去，最终把FlatView返回。这样`generate_memory_topology`就算是介绍完了，下面的`flatview_simplify`是对数组表项的尝试合并，这里就不再介绍。到此为止，已经针对当前MR生成了一个新的FlatView，接下来需要用`address_space_update_topology_pass`函数对`old_view`和`new_view`做对比。连续调用了两次这个函数，不过最后的adding参数先后为false和true。进入函数分析下

```
static void address_space_update_topology_pass(AddressSpace *as,
                                              const FlatView *old_view,
                                              const FlatView *new_view,
                                              bool adding)
{
    /*FR计数*/
    unsigned iold, inew;
    FlatRange *froid, *frnew;

    /* Generate a symmetric difference of the old and new memory maps.
     * Kill ranges in the old map, and instantiate ranges in the new map.
     */
    iold = inew = 0;
    while (iold < old_view->nr || inew < new_view->nr) {
        if (iold < old_view->nr) {
            froid = &old_view->ranges[iold];
        } else {
            froid = NULL;
        }
        if (inew < new_view->nr) {
            frnew = &new_view->ranges[inew];
        } else {
            frnew = NULL;
        }
        /*int128_lt 小于等于*/
        /*int128_eq 等于*/
        /*froid not null and( frnew is null || froid not null and old.start <= new.start
        || froid not null and old.start == new.start) and old !=new*/
        if (froid
            && (!frnew
                || int128_lt(froid->addr.start, frnew->addr.start)
                || (int128_eq(froid->addr.start, frnew->addr.start)
                    && !flatrange_equal(froid, frnew)))) {
            /* In old but not in new, or in both but attributes changed. */
            /*if not add*/
            if (!adding) {
                MEMORY_LISTENER_UPDATE_REGION(froid, as, Reverse, region_del);
            }

            ++iold;
            /*if old and new and old==new*/
        } else if (froid && frnew && flatrange_equal(froid, frnew)) {
            /* In both and unchanged (except logging may have changed) */
            /*if add*/
            if (adding) {
                MEMORY_LISTENER_UPDATE_REGION(frnew, as, Forward, region_nop);
                if (froid->dirty_log_mask && !frnew->dirty_log_mask) {
                    MEMORY_LISTENER_UPDATE_REGION(frnew, as, Reverse, log_stop);
                } else if (frnew->dirty_log_mask && !froid->dirty_log_mask) {
                    MEMORY_LISTENER_UPDATE_REGION(frnew, as, Forward, log_start);
                }
            }
            ++iold;
        }
    }
}
```

```

        ++inew;
    } else {
        /* In new */
        /*if add*/
        if (adding) {
            MEMORY_LISTENER_UPDATE_REGION(frnew, as, Forward, region_add);
        }
        ++inew;
    }
}
}
}

```

该函数倒是不长，主体是一个while循环，循环条件是old\_view->nr和new\_view->nr,表示新旧view的可用FlatRange数目。这里依次对FR数组的对应FR 做对比，主要由下面几种情况：froid和frnew均存在、froid存在但frnew不存在，froid不存在但frnew存在。下面的if划分和上面的略有不同：

- 1、如果froid不为空&&（frnew为空||froid.start<frnew.start||froid.start=frnew.start）&&froid!=frnew 这种情况是新旧view的地址范围不一样，则需要调用linter的region\_del对froid进行删除。
- 2、如果froid和frnew均不为空且froid.start=frnew.start 这种情况需要判断日志掩码，如果froid->dirty\_log\_mask && !frnew->dirty\_log\_mask，调用log\_stop回调函数；如果frnew->dirty\_log\_mask && !froid->dirty\_log\_mask，调用log\_start回调函数。
- 3、froid为空但是frnew不为空 这种情况直接调用region\_add回调函数添加region。

函数主体逻辑基本如上所述，那我们注意到，当adding为false时，执行的只有第一个情况下的处理，就是删除froid的操作，其余的处理只有在adding 为true的时候才得以执行。这意图就比较明确，首次执行先删除多余的，下次直接添加或者对日志做更新操作了。

总结：qemu内存虚拟化部分先介绍到这里，本来还想把kvm\_region\_add函数介绍下，但是考虑到篇幅，同时也正好利用该函数过渡到KVM中，不会显得突兀。由于笔者能力有限，文中不免有不对的地方，还望老师们多多指教，大家一起学习，共同进步！！

参考：qemu源码 kvm源码

分类: [KVM虚拟化技术](#), [qemu](#)




[jack.chen](#)  
 关注 - 12  
 粉丝 - 44  
[+加关注](#)

2 0

« 上一篇: [Linux进程虚拟地址空间的管理](#)  
 » 下一篇: [Linux进程虚拟地址空间管理2](#)

posted @ 2017-04-20 11:11 jack.chen Views(3178) Comments(6) Edit 收藏

## Post Comment

#1楼 2017-06-26 21:52 | fangying

赞，接触QEMU一年多了，但对这块理解一直理解不够深刻~

支持(0) 反对(1)

#2楼 2017-08-05 18:33 | aduev

博主，您好，根据qmp的info mtree的实现，想和您探讨一下我的这些理解对不对：

- 1.MemoryRegion的addr成员是VM中某个物理内存块的首地址(GPA)，size是此内存块的大小，ram\_addr成员对内嵌在RAMBlock中的mr好像才有意义，非内嵌于RAMBlock的MR的ram\_addr缺省都是64位的最大值（貌似以后不会去修改）
- 2.qemu对分配的所有RAMBlock会统一进行连续编址作为物理内存，RAMBlock的offset表示的是此RAMBlock在这连续物理地址空间的偏移，并且会把这个值也赋给内嵌于它的MR的ram\_addr成员

3.alias是给一个大内存块内的一小块内存块起的别名，但实际上好像只会用在内嵌于RAMBlock的MR上，即只有对非实际ram的MR上才会划分小的MR别起别名

支持(0) 反对(0)

#3楼 [楼主] 2017-08-15 14:30 | jack.chen

@ aduev  
抱歉，有些忘了，凑空看看再回复您！

支持(0) 反对(0)

#4楼 [楼主] 2017-09-01 10:11 | jack.chen

@ aduev  
你好，很抱歉现在才回复！一下是个人针对你问题的观点  
1、感觉这里你理解颠倒了，系统中生效的除去IO，就是一个systemmemory的MemoryRegion，该MemoryRegion包含两个子MemoryRegion，按照4G做划分，针对父MemoryRegion，其addr没有太大意义，addr主要表示子a在父MemoryRegion中的offset，比如below\_4g，offset就是0。对于ram\_addr,若分层次，其应该位于MR下面，你这说法略有不妥，而ram\_addr实际上才是GPA。  
2、同意。  
3、alias主要是针对子MR而言的。  
如果有任何问题，欢迎交流

支持(0) 反对(0)

#5楼 2017-09-01 17:20 | aduev

@ jack.chen  
谢谢博主的认真回复，addr确实是子mr在父mr上的偏移。由于mr树只有两层，并且below\_4g的起始offset是0，且创建的vm内存没有超过4G，这些巧合导致叶子节点mr的addr刚好是vm中的GPA，所以我误以为addr就是GPA；实际上是从mr树的根开始到叶子mr的路径上的addr加起来才是叶子mr对应内存的GPA。  
ram\_addr对非内嵌于RAMBlock中的mr有实际意义吗？好像初始化为RAM\_ADDR\_INVALID后没有再修改过？

支持(0) 反对(0)

#6楼 [楼主] 2017-09-01 19:47 | jack.chen

@ aduev  
子MR的该字段的确没有设置

支持(0) 反对(0)

[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】2019热门技术盛会400则演讲资料全收录

【推荐】35个面试详解，170道挑战题，1460个精彩问答 | Java面试宝典

相关博文：

- qemu对虚拟机的内存管理（二）
  - qemu中的内存管理
  - Qemu创建KVM虚拟机内存初始化流程
  - qemu-kvm内存虚拟化2
  - qemu对虚拟机的内存管理（一）
- » 更多推荐...

斩获阿里offer的必看12篇面试合集

**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