【系列分享】QEMU内存虚拟化源码分析

发布时间：2017-07-12 10:10:48

作者：Terenceli @ 360 Gear Team

投稿方式：发送邮件至linwei#360.cn，或登陆网页版在线投稿

传送门

【系列分享】探索QEMU-KVM中PIO处理的奥秘

内存虚拟化就是为虚拟机提供内存，使得虚拟机能够像在物理机上正常工作，这需要虚拟化软件为虚拟机展示一种物理内存的假象，内存虚拟化是虚拟化技术中关键技术之一。qemu+kvm的虚拟化方案中，内存虚拟化是由qemu和kvm共同完成的。qemu的虚拟地址作为guest的物理地址，一句看似轻描淡写的话幕后的工作确实非常多，加上qemu本身可以独立于kvm，成为一个完整的虚拟化方案，所以其内存虚拟化更加复杂。本文试图全方位的对qemu的内存虚拟化方案进行源码层面的介绍。本文主要介绍qemu在内存虚拟化方面的工作,之后的文章会介绍内存kvm方面的内存虚拟化。

零. 概述

内存虚拟化就是要让虚拟机能够无缝的访问内存，这个内存哪里来的，qemu的进程地址空间分出来的。有了ept之后，CPU在vmx non-root状态的时候进行内存访问会再做一个ept转换。在这个过程中，qemu扮演的角色。1. 首先需要去申请内存用于虚拟机； 2. 需要将虚拟1中申请的地址的虚拟地址与虚拟机的对应的物理地址告诉给kvm，就是指定GPA->HVA的映射关系；3. 需要组织一系列的数据结构去管理控制内存虚拟化，比如，设备注册需要分配物理地址，虚拟机退出之后需要根据地址做模拟等等非常多的工作，由于qemu本身能够支持tcg模式的虚拟化，会显得更加复杂。

首先明确内存虚拟化中QEMU和KVM工作的分界。KVM的ioctl中，设置虚拟机内存的为KVM_SET_USER_MEMORY_REGION，我们看到这个ioctl需要传递的参数是:

```
/* for KVM_SET_USER_MEMORY_REGION */
struct kvm_userspace_memory_region {
    __u32 slot;
    __u32 flags;
    __u64 guest_phys_addr;
    __u64 memory_size; /* bytes */
    __u64 userspace_addr; /* start of the userspace allocated memory */
};
```

这个ioctl主要就是设置GPA到HVA的映射。看似简单的工作在qemu里面却很复杂，下面逐一剖析之。

一. 相关数据结构

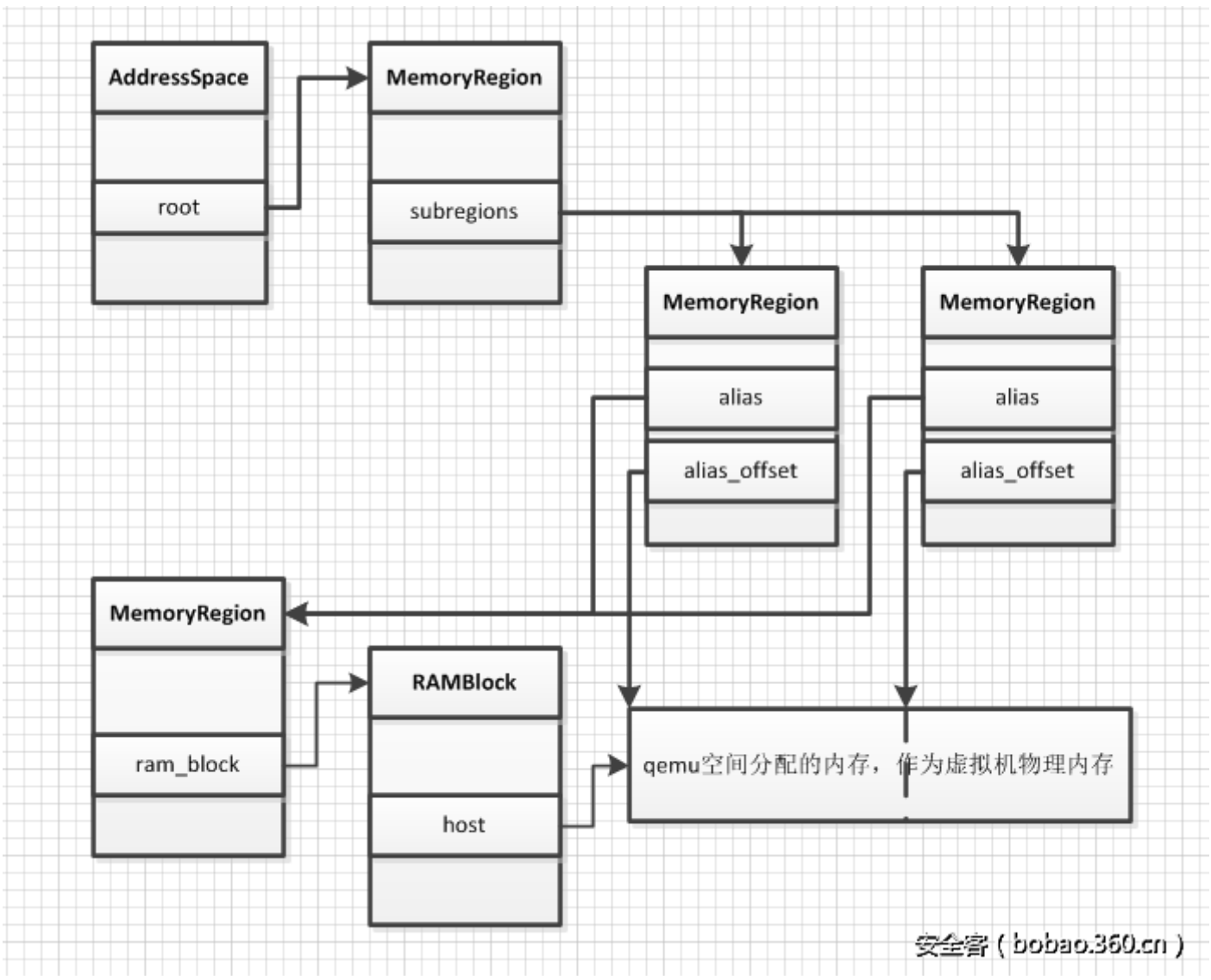
首先，qemu中用AddressSpace用来表示CPU/设备看到的内存，一个AddressSpace下面包含多个MemoryRegion，这些MemoryRegion结构通过树连接起来，树的根是AddressSpace的root域。

```
struct AddressSpace {
    /* All fields are private. */
    struct rcu_head rcu;
    char *name;
    MemoryRegion *root;
    int ref_count;
    bool malloced;
    /* Accessed via RCU.  */
    struct FlatView *current_map;
    int ioeventfd_nb;
    struct MemoryRegionIoeventfd *ioeventfds;
    struct AddressSpaceDispatch *dispatch;
    struct AddressSpaceDispatch *next_dispatch;
    MemoryListener dispatch_listener;
    QTAILQ_HEAD(memory_listeners_as, MemoryListener) listeners;
    QTAILQ_ENTRY(AddressSpace) address_spaces_link;
};
struct MemoryRegion {
    Object parent_obj;
    /* All fields are private - violators will be prosecuted */
    /* The following fields should fit in a cache line */
    bool romd_mode;
    bool ram;
    bool subpage;
    bool readonly; /* For RAM regions */
    bool rom_device;
    bool flush_coalesced_mmio;
    bool global_locking;
    uint8_t dirty_log_mask;
    RAMBlock *ram_block;
    ...
    const MemoryRegionOps *ops;
    void *opaque;
    MemoryRegion *container;
    Int128 size;
    hwaddr addr;
    ...
    MemoryRegion *alias;
    hwaddr alias_offset;
    int32_t priority;
    QTAILQ_HEAD(subregions, MemoryRegion) subregions;
    QTAILQ_ENTRY(MemoryRegion) subregions_link;
    QTAILQ_HEAD(coalesced_ranges, CoalescedMemoryRange) coalesced;
    ...
};
```

MemoryRegion有多种类型，可以表示一段ram，rom，MMIO，alias，alias表示一个MemoryRegion的一部分区域，MemoryRegion也可以表示一个container，这就表示它只是其他若干个MemoryRegion的容器。在MemoryRegion中，'ram_block'表示的是分配的实际内存。

```
struct RAMBlock {
    struct rcu_head rcu;
    struct MemoryRegion *mr;
    uint8_t *host;
    ram_addr_t offset;
    ram_addr_t used_length;
    ram_addr_t max_length;
    void (*resized)(const char*, uint64_t length, void *host);
    uint32_t flags;
    /* Protected by iothread lock.  */
    char idstr[256];
    /* RCU-enabled, writes protected by the ramlist lock */
    QLIST_ENTRY(RAMBlock) next;
    int fd;
    size_t page_size;
};
```

在这里，'host'指向了动态分配的内存，用于表示实际的虚拟机物理内存，而offset表示了这块内存在虚拟机物理内存中的偏移。每一个ram_block还会被连接到全局的'ram_list'链表上。Address, MemoryRegion, RAMBlock关系如下图所示。



AddressSpace下面root及其子树形成了一个虚拟机的物理地址，但是在往kvm进行设置的时候，需要将其转换为一个平坦的地址模型，也就是从0开始的。这个就用FlatView表示，一个AddressSpace对应一个FlatView。

```
struct FlatView {
    struct rcu_head rcu;
    unsigned ref;
    FlatRange *ranges;
    unsigned nr;
    unsigned nr_allocated;
};
```

在FlatView中，FlatRange表示按照需要被切分为了几个范围。

在内存虚拟化中，还有一个重要的结构是MemoryRegionSection，这个结构通过函数section_from_flat_range可由FlatRange转换过来。

```
struct MemoryRegionSection {
    MemoryRegion *mr;
    AddressSpace *address_space;
    hwaddr offset_within_region;
    Int128 size;
    hwaddr offset_within_address_space;
    bool readonly;
};
```

MemoryRegionSection表示的是MemoryRegion的一部分。这个其实跟FlatRange差不多。这几个数据结构关系如下：

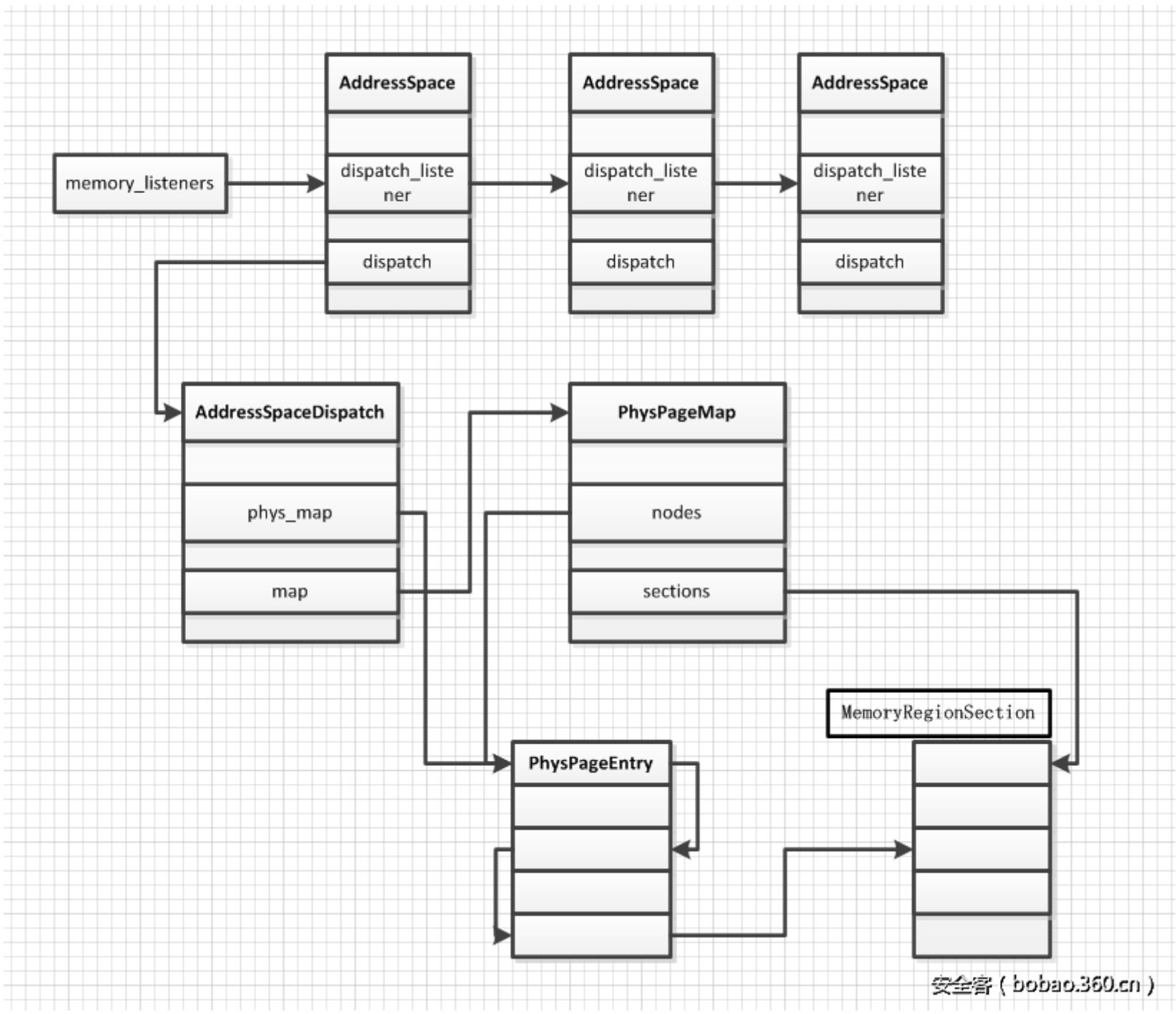
为了监控虚拟机的物理地址访问，对于每一个AddressSpace，会有一个MemoryListener与之对应。每当物理映射（GPA->HVA)发生改变时，会回调这些函数。所有的MemoryListener都会挂在全局变量memory_listeners链表上。同时，AddressSpace也会有一个链表连接器自己注册的MemoryListener。

```
struct MemoryListener {
    void (*begin)(MemoryListener *listener);
    void (*commit)(MemoryListener *listener);
    void (*region_add)(MemoryListener *listener, MemoryRegionSection *section);
    void (*region_del)(MemoryListener *listener, MemoryRegionSection *section);
    void (*region_nop)(MemoryListener *listener, MemoryRegionSection *section);
    void (*log_start)(MemoryListener *listener, MemoryRegionSection *section,
                      int old, int new);
    void (*log_stop)(MemoryListener *listener, MemoryRegionSection *section,
                     int old, int new);
    void (*log_sync)(MemoryListener *listener, MemoryRegionSection *section);
    void (*log_global_start)(MemoryListener *listener);
    void (*log_global_stop)(MemoryListener *listener);
    void (*eventfd_add)(MemoryListener *listener, MemoryRegionSection *section,
                        bool match_data, uint64_t data, EventNotifier *e);
    void (*eventfd_del)(MemoryListener *listener, MemoryRegionSection *section,
                        bool match_data, uint64_t data, EventNotifier *e);
    void (*coalesced_mmio_add)(MemoryListener *listener, MemoryRegionSection *section,
                               hwaddr addr, hwaddr len);
    void (*coalesced_mmio_del)(MemoryListener *listener, MemoryRegionSection *section,
                               hwaddr addr, hwaddr len);
    /* Lower = earlier (during add), later (during del) */
    unsigned priority;
    AddressSpace *address_space;
    QTAILQ_ENTRY(MemoryListener) link;
    QTAILQ_ENTRY(MemoryListener) link_as;
};
```

为了在虚拟机退出时，能够顺利根据物理地址找到对应的HVA地址，qemu会有一个AddressSpaceDispatch结构，用来在AddressSpace中进行位置的找寻，继而完成对IO/MMIO地址的访问。

```
struct AddressSpaceDispatch {
    struct rcu_head rcu;
    MemoryRegionSection *mru_section;
    /* This is a multi-level map on the physical address space.
     * The bottom level has pointers to MemoryRegionSections.
     */
    PhysPageEntry phys_map;
    PhysPageMap map;
    AddressSpace *as;
};
```

这里面有一个PhysPageMap，这其实也是保存了一个GPA->HVA的一个映射，通过多层页表实现，当kvm exit退到qemu之后，通过这个AddressSpaceDispatch里面的map查找对应的MemoryRegionSection，继而找到对应的主机HVA。这几个结构体的关系如下：



下面对流程做一些分析。

二. 初始化

首先在main->cpu_exec_init_all->memory_map_init中对全局的memory和io进行初始化，system_memory作为address_space_memory的根MemoryRegion，大小涵盖了整个64位空间的大小，当然，这是一个pure contaner,并不会分配空间的，system_io作为address_space_io的根MemoryRegion，大小为65536，也就是平时的io port空间。

```
static void memory_map_init(void)
{
    system_memory = g_malloc(sizeof(*system_memory));
    memory_region_init(system_memory, NULL, "system", UINT64_MAX);
    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");
}
```

在随后的cpu初始化之中，还会初始化多个AddressSpace，这些很多都是disabled的，对虚拟机意义不大。重点在随后的main->pc_init_v2_8->pc_init1->pc_memory_init中，这里面是分配系统ram，也是第一次真正为虚拟机分配物理内存。整个过程中，分配内存也不会像MemoryRegion那么频繁，mr很多时候是创建一个alias，指向已经存在的mr的一部分，这也是alias的作用，就是把一个mr分割成多个不连续的mr。真正分配空间的大概有这么几个，pc.ram, pc.bios, pc.rom, 以及设备的一些ram, rom等，vga.vram, vga.rom, e1000.rom等。

分配pc.ram的流程如下：

```
memory_region_allocate_system_memory
allocate_system_memory_nonnuma
memory_region_init_ram
qemu_ram_alloc
ram_block_add
phys_mem_alloc
qemu_anon_ram_alloc
qemu_ram_mmap
mmap
```

可以看到，qemu通过使用mmap创建一个内存映射来作为ram。

继续pc_memory_init，函数在创建好了ram并且分配好了空间之后，创建了两个mr alias，ram_below_4g以及ram_above_4g，这两个mr分别指向ram的低4g以及高4g空间，这两个alias是挂在根system_memory mr下面的。以后的情形类似，创建根mr，创建AddressSpace，然后在根mr下面加subregion。

三. 内存的提交

当我们每一次更改上层的内存布局之后，都需要通知到kvm。这个过程是通过一系列的MemoryListener来实现的。首先系统有一个全局的memory_listeners，上面挂上了所有的MemoryListener，在address_space_init->address_space_init_dispatch->memory_listener_register这个过程中完成MemoryListener的注册。

```
void address_space_init_dispatch(AddressSpace *as)
{
    as->dispatch = NULL;
    as->dispatch_listener = (MemoryListener) {
        .begin = mem_begin,
        .commit = mem_commit,
        .region_add = mem_add,
        .region_nop = mem_add,
        .priority = 0,
    };
    memory_listener_register(&as->dispatch_listener, as);
}
```

这里有初始化了listener的几个回调，他们的的调用时间之后讨论。 值得注意的是，并不是只有AddressSpace初始化的时候会注册回调，kvm_init同样会注册回调。

```
static int kvm_init(MachineState *ms)
{
    ...
    kvm_memory_listener_register(s, &s->memory_listener,
                                 &address_space_memory, 0);
    memory_listener_register(&kvm_io_listener,
                             &address_space_io);
    ...
}
void kvm_memory_listener_register(KVMState *s, KVMMemoryListener *kml,
                                  AddressSpace *as, int as_id)
{
    int i;
    kml->slots = g_malloc0(s->nr_slots * sizeof(KVMSlot));
    kml->as_id = as_id;
    for (i = 0; i < s->nr_slots; i++) {
        kml->slots[i].slot = i;
    }
    kml->listener.region_add = kvm_region_add;
    kml->listener.region_del = kvm_region_del;
    kml->listener.log_start = kvm_log_start;
    kml->listener.log_stop = kvm_log_stop;
    kml->listener.log_sync = kvm_log_sync;
    kml->listener.priority = 10;
    memory_listener_register(&kml->listener, as);
}
```

在这里我们看到kvm也注册了自己的MemoryListener。

在上面看到MemoryListener之后，我们看看什么时候需要更新内存。 进行内存更新有很多个点，比如我们新创建了一个AddressSpace address_space_init，再比如我们将一个mr添加到另一个mr的subregions中memory_region_add_subregion,再比如我们更改了一端内存的属性memory_region_set_readonly，将一个mr设置使能或者非使能memory_region_set_enabled, 总之一句话，我们修改了虚拟机的内存布局/属性时，就需要通知到各个Listener，这包括各个AddressSpace对应的，以及kvm注册的，这个过程叫做commit，通过函数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) {
    if (memory_region_update_pending) {
      MEMORY_LISTENER_CALL_GLOBAL(begin, Forward);
      QTAILQ_FOREACH(as, &address_spaces, address_spaces_link) {
        address_space_update_topology(as);
      }
      MEMORY_LISTENER_CALL_GLOBAL(commit, Forward);
    } else if (ioeventfd_update_pending) {
      QTAILQ_FOREACH(as, &address_spaces, address_spaces_link) {
        address_space_update_ioeventfds(as);
      }
    }
    memory_region_clear_pending();
  }
}
#define MEMORY_LISTENER_CALL_GLOBAL(_callback, _direction, _args...)
  do {
    MemoryListener *_listener;

    switch (_direction) {
    case Forward:
      QTAILQ_FOREACH(_listener, &memory_listeners, link) {
        if (_listener->_callback) {
          _listener->_callback(_listener, ##_args);
        }
      }
      break;
    case Reverse:
      QTAILQ_FOREACH_REVERSE(_listener, &memory_listeners,
                             memory_listeners, link) {
        if (_listener->_callback) {
          _listener->_callback(_listener, ##_args);
        }
      }
      break;
    default:
      abort();
    }
  } while (0)
```

MEMORY_LISTENER_CALL_GLOBAL对memory_listeners上的各个MemoryListener调用指定函数。commit中最重要的是address_space_update_topology调用。

```
static void address_space_update_topology(AddressSpace *as)
{
  FlatView *old_view = address_space_get_flatview(as);
  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);
    /* Writes are protected by the BQL.  */
    atomic_rcu_set(&as->current_map, new_view);
    call_rcu(old_view, flatview_unref, rcu);
    /* 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);
}
```

前面我们已经说了，as->root会被展开为一个FlatView，所以在这里update topology中，首先得到上一次的FlatView，之后调用generate_memory_topology生成一个新的FlatView，

```
static FlatView *generate_memory_topology(MemoryRegion *mr)
{
    FlatView *view;
    view = g_new(FlatView, 1);
    flatview_init(view);
    if (mr) {
        render_memory_region(view, mr, int128_zero(),
                             addrrange_make(int128_zero(), int128_2_64()), false);
    }
    flatview_simplify(view);
    return view;
}
```

最主要的是render_memory_region生成view，这个render函数很复杂，需要递归render子树，具体以后有机会单独讨论。在生成了view之后会调用flatview_simplify进行简化，主要是合并相邻的FlatRange。在生成了当前as的FlatView之后，我们就可以更新了，这在函数address_space_update_topology_pass中完成，这个函数就是逐一对比新旧FlatView的差别，然后进行更新。

```
static void address_space_update_topology_pass(AddressSpace *as,
                                               const FlatView *old_view,
                                               const FlatView *new_view,
                                               bool adding)
{
    unsigned iold, inew;
    FlatRange *frold, *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) {
            frold = &old_view->ranges[iold];
        } else {
            frold = NULL;
        }
    if (inew < new_view->nr) {
        frnew = &new_view->ranges[inew];
    } else {
        frnew = NULL;
    }
    if (frold
        && (!frnew
            || int128_lt(frold->addr.start, frnew->addr.start)
            || (int128_eq(frold->addr.start, frnew->addr.start)
                && !flatrange_equal(frold, frnew)))) {
        /* In old but not in new, or in both but attributes changed. */
        if (!adding) {
            MEMORY_LISTENER_UPDATE_REGION(frold, as, Reverse, region_del);
        }
        ++iold;
    } else if (frold && frnew && flatrange_equal(frold, frnew)) {
        /* In both and unchanged (except logging may have changed) */
        if (adding) {
            MEMORY_LISTENER_UPDATE_REGION(frnew, as, Forward, region_nop);
            if (frnew->dirty_log_mask & ~frold->dirty_log_mask) {
                MEMORY_LISTENER_UPDATE_REGION(frnew, as, Forward, log_start,
                                              frold->dirty_log_mask,
                                              frnew->dirty_log_mask);
            }
            if (frold->dirty_log_mask & ~frnew->dirty_log_mask) {
                MEMORY_LISTENER_UPDATE_REGION(frnew, as, Reverse, log_stop,
                                              frold->dirty_log_mask,
                                              frnew->dirty_log_mask);
            }
        }
        ++iold;
        ++inew;
    } else {
```

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

最重要的当然是MEMORY_LISTENER_UPDATE_REGION宏，这个宏会将每一个FlatRange转换为一个MemoryRegionSection，之后调用这个as对应的各个MemoryListener的回调函数。这里我们以kvm对象注册Listener为例，从kvm_memory_listener_register，我们看到其region_add回调为kvm_region_add。

```
static void kvm_region_add(MemoryListener *listener,
                           MemoryRegionSection *section)
{
    KVMMemoryListener *kml = container_of(listener, KVMMemoryListener, listener);

    memory_region_ref(section->mr);

    kvm_set_phys_mem(kml, section, true);
}
```

这个函数看似复杂，主要是因为，需要判断变化的各种情况是否与之前的重合，是否是脏页等等情况。我们只看最开始的情况。

```
static void kvm_set_phys_mem(KVMMemoryListener *kml,
                             MemoryRegionSection *section, bool add)
{
    KVMState *s = kvm_state;
    KVMSlot *mem, old;
    int err;
    MemoryRegion *mr = section->mr;
    bool writeable = !mr->readonly && !mr->rom_device;
    hwaddr start_addr = section->offset_within_address_space;
    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. Pad the start
       address to next and truncate size to previous page boundary. */
    delta = qemu_real_host_page_size - (start_addr & ~qemu_real_host_page_mask);
    delta &= ~qemu_real_host_page_mask;
    if (delta > size) {
        return;
    }
    start_addr += delta;
    size -= delta;
    size &= qemu_real_host_page_mask;
    if (!size || (start_addr & ~qemu_real_host_page_mask)) {
        return;
    }
    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;
        }
    }
    ram = memory_region_get_ram_ptr(mr) + section->offset_within_region + delta;
    ...
    if (!size) {
        return;
    }
    if (!add) {
        return;
    }
    mem = kvm_alloc_slot(kml);
    mem->memory_size = size;
    mem->start_addr = start_addr;
    mem->ram = ram;
    mem->flags = kvm_mem_flags(mr);
    err = kvm_set_user_memory_region(kml, mem);
    if (err) {
```

```
        fprintf(stderr, "%s: error registering slot: %sn", __func__,
                strerror(-err));
        abort();
    }
}
```

这个函数主要就是得到MemoryRegionSection在address_space中的位置，这个就是虚拟机的物理地址，函数中是start_addr, 然后通过memory_region_get_ram_ptr得到对应其对应的qemu的HVA地址，函数中是ram，当然还有大小的size以及这块内存的flags，这些参数组成了一个KVMSlot，之后传递给kvm_set_user_memory_region。

```
static int kvm_set_user_memory_region(KVMMemoryListener *kml, KVMSlot *slot)
{
    KVMState *s = kvm_state;
    struct kvm_userspace_memory_region mem;
    mem.slot = slot->slot | (kml->as_id << 16);
    mem.guest_phys_addr = slot->start_addr;
    mem.userspace_addr = (unsigned long)slot->ram;
    mem.flags = slot->flags;
    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);
}
```

通过层层抽象，我们终于完成了GPA->HVA的对应，并且传递到了KVM。

四. kvm exit之后的内存寻址

在address_space_init_dispatch函数中，我们可以看到，每一个通过AddressSpace都会注册一个Listener回调，回调的各个函数都一样，mem_begin， mem_add等。

```
void address_space_init_dispatch(AddressSpace *as)
{
    as->dispatch = NULL;
    as->dispatch_listener = (MemoryListener) {
        .begin = mem_begin,
        .commit = mem_commit,
        .region_add = mem_add,
        .region_nop = mem_add,
        .priority = 0,
    };
    memory_listener_register(&as->dispatch_listener, as);
}
```

我们重点看看mem_add

```
static void mem_add(MemoryListener *listener, MemoryRegionSection *section)
{
    AddressSpace *as = container_of(listener, AddressSpace, dispatch_listener);
    AddressSpaceDispatch *d = as->next_dispatch;
    MemoryRegionSection now = *section, remain = *section;
    Int128 page_size = int128_make64(TARGET_PAGE_SIZE);
    if (now.offset_within_address_space & ~TARGET_PAGE_MASK) {
        uint64_t left = TARGET_PAGE_ALIGN(now.offset_within_address_space)
                        - now.offset_within_address_space;
        now.size = int128_min(int128_make64(left), now.size);
        register_subpage(d, &now);
    } else {
        now.size = int128_zero();
    }
    while (int128_ne(remain.size, now.size)) {
        remain.size = int128_sub(remain.size, now.size);
        remain.offset_within_address_space += int128_get64(now.size);
        remain.offset_within_region += int128_get64(now.size);
        now = remain;
        if (int128_lt(remain.size, page_size)) {
            register_subpage(d, &now);
        } else if (remain.offset_within_address_space & ~TARGET_PAGE_MASK) {
            now.size = page_size;
            register_subpage(d, &now);
        } else {
            now.size = int128_and(now.size, int128_neg(page_size));
            register_multipage(d, &now);
        }
    }
}
```

mem_add在添加了内存区域之后会被调用，调用路径为

```
address_space_update_topology_pass
MEMORY_LISTENER_UPDATE_REGION(frnew, as, Forward, region_add);
#define MEMORY_LISTENER_UPDATE_REGION(fr, as, dir, callback, _args...)
  do {
    MemoryRegionSection mrs = section_from_flat_range(fr, as);
    MEMORY_LISTENER_CALL(as, callback, dir, &mrs, ##_args);
  } while(0)
```

如果新增加了一个FlatRange，则会调用将该fr转换为一个MemroyRegionSection，然后调用Listener的region_add。

回到mem_add，这个函数主要是调用两个函数如果是添加的地址落到一个页内，则调用register_subpage，如果是多个页，则调用register_multipage，先看看register_multipage，因为最开始注册都是一波大的，比如pc.ram。首先now.offset_within_address_space并不会落在一个页内。所以直接进入while循环，之后进入register_multipage，d这个AddressSpaceDispatch是在mem_begin创建的。

```
static void register_multipage(AddressSpaceDispatch *d,
                               MemoryRegionSection *section)
{
    hwaddr start_addr = section->offset_within_address_space;
    uint16_t section_index = phys_section_add(&d->map, section);
    uint64_t num_pages = int128_get64(int128_rshift(section->size,
                                                    TARGET_PAGE_BITS));
    assert(num_pages);
    phys_page_set(d, start_addr >> TARGET_PAGE_BITS, num_pages, section_index);
}
```

首先分一个d->map->sections空间出来，其index为section_index。

```
static void phys_page_set(AddressSpaceDispatch *d,
                          hwaddr index, hwaddr nb,
                          uint16_t leaf)
{
    /* Wildly overreserve - it doesn't matter much. */
    phys_map_node_reserve(&d->map, 3 * P_L2_LEVELS);
    phys_page_set_level(&d->map, &d->phys_map, &index, &nb, leaf, P_L2_LEVELS - 1);
}
```

之后start_addr右移12位，计算出总共需要多少个页。这里说一句，qemu在这里总共使用了6级页表，最后一级长度12，然后是5 * 9 + 7。phys_map_node_reserve首先分配页目录项。

```
static void phys_map_node_reserve(PhysPageMap *map, unsigned nodes)
{
    static unsigned alloc_hint = 16;
    if (map->nodes_nb + nodes > map->nodes_nb_alloc) {
        map->nodes_nb_alloc = MAX(map->nodes_nb_alloc, alloc_hint);
        map->nodes_nb_alloc = MAX(map->nodes_nb_alloc, map->nodes_nb + nodes);
        map->nodes = g_renew(Node, map->nodes, map->nodes_nb_alloc);
        alloc_hint = map->nodes_nb_alloc;
    }
}
```

phys_page_set_level填充页表。初始调用时，level为5，因为要从最开始一层填充。

```
static void phys_page_set_level(PhysPageMap *map, PhysPageEntry *lp,
                                hwaddr *index, hwaddr *nb, uint16_t leaf,
                                int level)
{
    PhysPageEntry *p;
    hwaddr step = (hwaddr)1 << (level * P_L2_BITS);
    if (lp->skip && lp->ptr == PHYS_MAP_NODE_NIL) {
        lp->ptr = phys_map_node_alloc(map, level == 0);
    }
    p = map->nodes[lp->ptr];
    lp = &p[(*index >> (level * P_L2_BITS)) & (P_L2_SIZE - 1)];
    while (*nb && lp < &p[P_L2_SIZE]) {
        if ((*index & (step - 1)) == 0 && *nb >= step) {
            lp->skip = 0;
            lp->ptr = leaf;
            *index += step;
            *nb -= step;
        } else {
            phys_page_set_level(map, lp, index, nb, leaf, level - 1);
        }
        ++lp;
    }
}
```

这个函数主要就是建立一个多级页表。如图所示

```
struct PhysPageEntry {
    /* How many bits skip to next level (in units of L2_SIZE). 0 for a leaf. */
    uint32_t skip : 6;
     /* index into phys_sections (!skip) or phys_map_nodes (skip) */
    uint32_t ptr : 26;
};
```

简单说说PhysPageEntry, skip表示需要移动多少步到下一级页表，如果skip为0，说明这是最末级页表了，ptr指向的是map->sections数组的某一项。如果skip不为0，则ptr指向的是哪一个node，也就是页目录。总而言之，这个函数的作用就是建立起一个多级页表，最末尾的页表项表示的是MemoryRegionSection，这跟OS里面的页表是一个道理，而AddressSpaceDispatch中的phys_map域则相当于CR3寄存器，用来最开始的寻址。

好了，我们已经分析好了register_multipage。现在看看register_subpage。

为什么会有在一个页面内注册的需求呢，我的理解是这样的 我们来看一下io port的分布，很明显在一个page里面会有多个MemoryRegion，所以这些内存空间需要分开的MemroyRegionSection,但是呢，这种情况又不是很普遍的，对于内存来说，很多时候1页，2页都是同一个MemoryRegion，总不能对于所有的地址都来一个MemoryRegionSection，所以呢，才会有这么一个subpage，有需要的时候再创建，没有就是整个mutipage。

```
0000000000000000-0000000000000007 (prio 0, RW): dma-chan
0000000000000008-000000000000000f (prio 0, RW): dma-cont
0000000000000020-0000000000000021 (prio 0, RW): kvm-pic
0000000000000040-0000000000000043 (prio 0, RW): kvm-pit
0000000000000060-0000000000000060 (prio 0, RW): i8042-data
0000000000000061-0000000000000061 (prio 0, RW): pcspk
0000000000000064-0000000000000064 (prio 0, RW): i8042-cmd
0000000000000070-0000000000000071 (prio 0, RW): rtc
```

有subpage的情况如下图：

好了，有了上面的知识，我们可以来看对于kvm io exit之后的寻址过程了。

```
int kvm_cpu_exec(CPUState *cpu)
{
    switch (run->exit_reason) {
    case KVM_EXIT_IO:
      DPRINTF("handle_ion");
      /* Called outside BQL */
      kvm_handle_io(run->io.port, attrs,
                    (uint8_t *)run + run->io.data_offset,
                    run->io.direction,
                    run->io.size,
                    run->io.count);
      ret = 0;
      break;
    case KVM_EXIT_MMIO:
      DPRINTF("handle_mmion");
      /* Called outside BQL */
      address_space_rw(&address_space_memory,
                       run->mmio.phys_addr, attrs,
                       run->mmio.data,
                       run->mmio.len,
                       run->mmio.is_write);
      ret = 0;
      break;
}
```

这里我们以KVM_EXIT_IO为例说明

```
static void kvm_handle_io(uint16_t port, MemTxAttrs attrs, void *data, int direction,
                          int size, uint32_t count)
{
    int i;
    uint8_t *ptr = data;
    for (i = 0; i < count; i++) {
        address_space_rw(&address_space_io, port, attrs,
                         ptr, size,
                         direction == KVM_EXIT_IO_OUT);
        ptr += size;
    }
}
```

可以看到是在全局的address_space_io中寻址，这里我们只看寻址过程，找到HVA之后数据拷贝这些就不说了。

```
address_space_rw->address_space_write->address_space_translate->address_space_translate_internal
```

直接看最后一个函数

```
address_space_translate_internal(AddressSpaceDispatch *d, hwaddr addr, hwaddr *xlat,
                                 hwaddr *plen, bool resolve_subpage)
{
    MemoryRegionSection *section;
    MemoryRegion *mr;
    Int128 diff;
    section = address_space_lookup_region(d, addr, resolve_subpage);
    /* Compute offset within MemoryRegionSection */
    addr -= section->offset_within_address_space;
    /* Compute offset within MemoryRegion */
    *xlat = addr + section->offset_within_region;
    mr = section->mr;

    if (memory_region_is_ram(mr)) {
        diff = int128_sub(section->size, int128_make64(addr));
        *plen = int128_get64(int128_min(diff, int128_make64(*plen)));
    }
    return section;
}
```

最重要的当然是找到对应的MemroyRegionSection

```
static MemoryRegionSection *address_space_lookup_region(AddressSpaceDispatch *d,
                                                        hwaddr addr,
                                                        bool resolve_subpage)
{
    MemoryRegionSection *section = atomic_read(&d->mru_section);
    subpage_t *subpage;
    bool update;
    if (section && section != &d->map.sections[PHYS_SECTION_UNASSIGNED] &&
        section_covers_addr(section, addr)) {
        update = false;
    } else {
        section = phys_page_find(d->phys_map, addr, d->map.nodes,
                                 d->map.sections);
        update = true;
    }
    if (resolve_subpage && section->mr->subpage) {
        subpage = container_of(section->mr, subpage_t, iomem);
        section = &d->map.sections[subpage->sub_section[SUBPAGE_IDX(addr)]];
    }
    if (update) {
        atomic_set(&d->mru_section, section);
    }
    return section;
}
```

d->mru_section作为一个缓存，由于局部性原理，这样可以提高效率。我们看到phys_page_find，类似于一个典型的页表查询过程，通过addr一步一步查找到最后的MemoryRegionSection。

```
static MemoryRegionSection *phys_page_find(PhysPageEntry lp, hwaddr addr,
                                           Node *nodes, MemoryRegionSection *sections)
{
    PhysPageEntry *p;
    hwaddr index = addr >> TARGET_PAGE_BITS;
    int i;
    for (i = P_L2_LEVELS; lp.skip && (i -= lp.skip) >= 0;) {
        if (lp.ptr == PHYS_MAP_NODE_NIL) {
            return &sections[PHYS_SECTION_UNASSIGNED];
        }
        p = nodes[lp.ptr];
        lp = p[(index >> (i * P_L2_BITS)) & (P_L2_SIZE - 1)];
    }
    if (section_covers_addr(&sections[lp.ptr], addr)) {
        return &sections[lp.ptr];
    } else {
        return &sections[PHYS_SECTION_UNASSIGNED];
    }
}
```

回到address_space_lookup_region，接着解析subpage，如果之前的subpage部分理解了，这里就很容易了。这样就返回了我们需要的MemoryRegionSection。

五. 总结

写这篇文章算是对qemu内存虚拟化的一个总结，参考了网上大神的文章，感谢之，当然，自己也有不少内容。这篇文章也有很多细节没有写完，比如从mr renader出FlatView，比如，根据前后的FlatView进行memory的commit，如果以后有时间补上。

六. 参考

1. 六六哥的博客

2. OENHAN

传送门

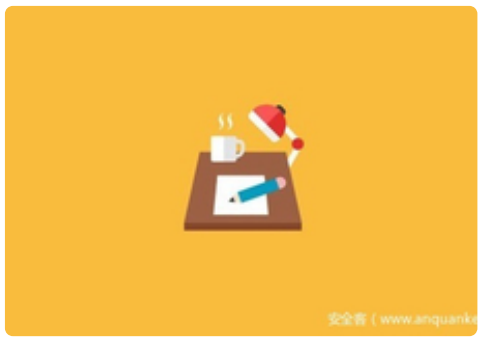
【系列分享】探索QEMU-KVM中PIO处理的奥秘

安全知识

推荐阅读

通过两道题浅看java安全

2020-05-18 16:30:48



2020网鼎杯白虎组的部分wp

2020-05-18 16:00:20

通过EFER寄存器实现基于VT的syscall挂钩

2020-05-18 15:30:52

honggfuzz-v2.X版本变异策略及const_feedback特性分析

2020-05-18 14:30:49

发表评论

发表你的评论吧

昵称 杨教授 换一个 发表评论

评论列表

还没有评论呢，快去抢个沙发吧~

360GearTeam

这个人太懒了，签名都懒得写一个

文章 32　粉丝 6

✚ 关注

TA的文章

Chimay-Red：RouterOS Integer Overflow Analysis

2019-12-25 10:30:54

From Dvr to See Exploit of IoT Device

2019-06-13 15:00:40

Make It Clear with RouterOS

2019-03-07 11:56:44

BLE安全初探之HACKMELOCK

2018-11-28 14:47:34

Linux 内核提权 CVE-2018-13405 分析

2018-07-31 16:30:53

安全客
有思想的安全新媒体

安全客

关于我们

加入我们

联系我们

用户协议

商务合作

合作内容

联系方式

友情链接

内容须知

投稿须知

转载须知

官网QQ群3：830462644

官网QQ群2：814450983(已满)

官网QQ群1：702511263(已满)

合作单位

CNCERT/CC 国家互联网应急中心