

52 | 计算虚拟化之内存：如何建立独立的办公室？

2019-07-26 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 23:26 大小 21.47M



上一节，我们解析了计算虚拟化之 CPU。可以看到，CPU 的虚拟化是用户态的 qemu 和内核态的 KVM 共同配合完成的。它们二者通过 ioctl 进行通信。对于内存管理来讲，也是需要这两者配合完成的。

咱们在内存管理的时候讲过，操作系统给每个进程分配的内存都是虚拟内存，需要通过页表映射，变成物理内存进行访问。当有了虚拟机之后，情况会变得更加复杂。因为虚拟机对于物理机来讲是一个进程，但是虚拟机里面也有内核，也有虚拟机里面跑的进程。所以有了虚拟机，内存就变成了四类：

虚拟机里面的虚拟内存（Guest OS Virtual Memory，GVA），这是虚拟机里面的进程看到的内存空间；

虚拟机里面的物理内存（ Guest OS Physical Memory , GPA ），这是虚拟机里面的操作系统看到的内存，它认为这是物理内存；

物理机的虚拟内存（ Host Virtual Memory , HVA ），这是物理机上的 qemu 进程看到的内存空间；

物理机的物理内存（ Host Physical Memory , HPA ），这是物理机上的操作系统看到的内存。


咱们内存管理那一章讲的两大内容，一个是内存管理，它变得非常复杂；另一个是内存映射，具体来说就是，从 GVA 到 GPA，到 HVA，再到 HPA，这样几经转手，计算机的性能就会变得很差。当然，虚拟化技术成熟的今天，有了一些优化的手段，具体怎么优化呢？我们这一节就来一一解析。

内存管理

我们先来看内存管理的部分。


由于 CPU 和内存是紧密结合的，因而内存虚拟化的初始化过程，和 CPU 虚拟化的初始化是一起完成的。

上一节说 CPU 虚拟化初始化的时候，我们会调用 `kvm_init` 函数，这里面打开了 `/dev/kvm` 这个字符文件，并且通过 `ioctl` 调用到内核 `kvm` 的 `KVM_CREATE_VM` 操作，除了这些 CPU 相关的调用，接下来还有内存相关的。我们来看看。

 复制代码

```
1 static int kvm_init(MachineState *ms)
2 {
3     MachineClass *mc = MACHINE_GET_CLASS(ms);
4     .....
5     kvm_memory_listener_register(s, &s->memory_listener,
6                                   &address_space_memory, 0);
7     memory_listener_register(&kvm_io_listener,
8                              &address_space_io);
9     .....
10 }
11
12 AddressSpace address_space_io;
13 AddressSpace address_space_memory;
```


这里面有两个地址空间 AddressSpace，一个是系统内存的地址空间 address_space_memory，一个用于 I/O 的地址空间 address_space_io。这里我们重点看 address_space_memory。

 复制代码

```
1 struct AddressSpace {
2     /* All fields are private. */
3     struct rcu_head rcu;
4     char *name;
5     MemoryRegion *root;
6
7     /* Accessed via RCU. */
8     struct FlatView *current_map;
9
10    int ioeventfd_nb;
11    struct MemoryRegionIoeventfd *ioeventfds;
12    QTAILQ_HEAD(, MemoryListener) listeners;
13    QTAILQ_ENTRY(AddressSpace) address_spaces_link;
14 };
```

对于一个地址空间，会有多个内存区域 MemoryRegion 组成树形结构。这里面，root 是这棵树的根。另外，还有一个 MemoryListener 链表，当内存区域发生变化的时候，需要做一些动作，使得用户态和内核态能够协同，就是由这些 MemoryListener 完成的。

在 kvm_init 这个时候，还没有内存区域加入进来，root 还是空的，但是我们可以先注册 MemoryListener，这里注册的是 KVMMemoryListener。


 复制代码

```
1 void kvm_memory_listener_register(KVMState *s, KVMMemoryListener *kml,
2                                   AddressSpace *as, int as_id)
3 {
4     int i;
5
6     kml->slots = g_malloc0(s->nr_slots * sizeof(KVMSlot));
7     kml->as_id = as_id;
8
9     for (i = 0; i < s->nr_slots; i++) {
10         kml->slots[i].slot = i;
11     }
12
13     kml->listener.region_add = kvm_region_add;
14     kml->listener.region_del = kvm_region_del;
```

```
15     kml->listener.priority = 10;
16
17     memory_listener_register(&kml->listener, as);
18 }
```


在这个 KVMMemoryListener 中是这样配置的：当添加一个 MemoryRegion 的时候，region_add 会被调用，这个我们后面会用到。

接下来，在 qemu 启动的 main 函数中，我们会调用 cpu_exec_init_all->memory_map_init.

 复制代码

```
1 static void memory_map_init(void)
2 {
3     system_memory = g_malloc(sizeof(*system_memory));
4
5     memory_region_init(system_memory, NULL, "system", UINT64_MAX);
6     address_space_init(&address_space_memory, system_memory, "memory");
7
8     system_io = g_malloc(sizeof(*system_io));
9     memory_region_init_io(system_io, NULL, &unassigned_io_ops, NULL, "io",
10                           65536);
11     address_space_init(&address_space_io, system_io, "I/O");
12 }
```

在这里，对于系统内存区域 system_memory 和用于 I/O 的内存区域 system_io，我们都进行了初始化，并且关联到了相应的地址空间 AddressSpace。


 复制代码

```
1 void address_space_init(AddressSpace *as, MemoryRegion *root, const char *name)
2 {
3     memory_region_ref(root);
4     as->root = root;
5     as->current_map = NULL;
6     as->ioeventfd_nb = 0;
7     as->ioeventfds = NULL;
8     QTAILQ_INIT(&as->listeners);
9     QTAILQ_INSERT_TAIL(&address_spaces, as, address_spaces_link);
10    as->name = g_strdup(name ? name : "anonymous");
11    address_space_update_topology(as);
12    address_space_update_ioeventfds(as);
```

```
13 }
```

对于系统内存地址空间 `address_space_memory`，我们需要把它里面内存区域的根 `root` 设置为 `system_memory`。

另外，在这里，我们还调用了 `address_space_update_topology`。

 复制代码

```
1 static void address_space_update_topology(AddressSpace *as)
2 {
3     MemoryRegion *physmr = memory_region_get_flatview_root(as->root);
4
5     flatviews_init();
6     if (!g_hash_table_lookup(flat_views, physmr)) {
7         generate_memory_topology(physmr);
8     }
9     address_space_set_flatview(as);
10 }
11
12 static void address_space_set_flatview(AddressSpace *as)
13 {
14     FlatView *old_view = address_space_to_flatview(as);
15     MemoryRegion *physmr = memory_region_get_flatview_root(as->root);
16     FlatView *new_view = g_hash_table_lookup(flat_views, physmr);
17
18     if (old_view == new_view) {
19         return;
20     }
21     .....
22     if (!QTAILQ_EMPTY(&as->listeners)) {
23         FlatView tmpview = { .nr = 0 }, *old_view2 = old_view;
24
25         if (!old_view2) {
26             old_view2 = &tmpview;
27         }
28         address_space_update_topology_pass(as, old_view2, new_view, false);
29         address_space_update_topology_pass(as, old_view2, new_view, true);
30     }
31
32     /* Writes are protected by the BQL. */
33     atomic_rcu_set(&as->current_map, new_view);
34     .....
35 }
```

这里面会生成 AddressSpace 的 flatview。flatview 是什么意思呢？


我们可以看到，在 AddressSpace 里面，除了树形结构的 MemoryRegion 之外，还有一个 flatview 结构，其实这个结构就是把这样一个树形的内存结构变成平的内存结构。因为树形内存结构比较容易管理，但是平的内存结构，比较方便和内核里面通信，来请求物理内存。虽然操作系统内核里面也是用树形结构来表示内存区域的，但是用户态向内核申请内存的时候，会按照平的、连续的模式进行申请。这里，qemu 在用户态，所以要做这样一个转换。

在 address_space_set_flatview 中，我们将老的 flatview 和新的 flatview 进行比较。如果不同，说明内存结构发生了变化，会调用 address_space_update_topology_pass->MEMORY_LISTENER_UPDATE_REGION->MEMORY_LISTENER_CALL。

这里面调用所有的 listener。但是，这个逻辑这里不会执行的。这是因为这里内存处于初始化的阶段，全局的 flat_views 里面肯定找不到。因而 generate_memory_topology 第一次生成了 FlatView，然后才调用了 address_space_set_flatview。这里面，老的 flatview 和新的 flatview 一定是一样的。

但是，请你记住这个逻辑，到这里我们还没解析 qemu 有关内存的参数，所以这里添加的 MemoryRegion 虽然是一个根，但是是空的，是为了管理使用的，后面真的添加内存的时候，这个逻辑还会调用到。

我们再回到 qemu 启动的 main 函数中。接下来的初始化过程会调用 pc_init1。在这里面，对于 CPU 虚拟化，我们会调用 pc_cpus_init。这个我们在上一节已经讲过了。另外，pc_init1 还会调用 pc_memory_init，进行内存的虚拟化，我们这里解析这一部分。

 复制代码

```
1 void pc_memory_init(PCMachineState *pcms,
2                     MemoryRegion *system_memory,
3                     MemoryRegion *rom_memory,
4                     MemoryRegion **ram_memory)
5 {
6     int linux_boot, i;
7     MemoryRegion *ram, *option_rom_mr;
8     MemoryRegion *ram_below_4g, *ram_above_4g;
9     FWCfgState *fw_cfg;
10    MachineState *machine = MACHINE(pcms);
11    PCMachineClass *pcmc = PC_MACHINE_GET_CLASS(pcms);
12    .....
```


```

13  /* Allocate RAM. We allocate it as a single memory region and use
14  * aliases to address portions of it, mostly for backwards compatibility with older
15  */
16  ram = g_malloc(sizeof(*ram));
17  memory_region_allocate_system_memory(ram, NULL, "pc.ram",
18                                     machine->ram_size);
19  *ram_memory = ram;
20  ram_below_4g = g_malloc(sizeof(*ram_below_4g));
21  memory_region_init_alias(ram_below_4g, NULL, "ram-below-4g", ram,
22                          0, pcms->below_4g_mem_size);
23  memory_region_add_subregion(system_memory, 0, ram_below_4g);
24  e820_add_entry(0, pcms->below_4g_mem_size, E820_RAM);
25  if (pcms->above_4g_mem_size > 0) {
26      ram_above_4g = g_malloc(sizeof(*ram_above_4g));
27      memory_region_init_alias(ram_above_4g, NULL, "ram-above-4g", ram, pcms->below_4g{
28      memory_region_add_subregion(system_memory, 0x100000000ULL,
29                                ram_above_4g);
30      e820_add_entry(0x100000000ULL, pcms->above_4g_mem_size, E820_RAM);
31  }
32  .....
33  }

```

在 `pc_memory_init` 中，我们已经知道了虚拟机要申请的内存 `ram_size`，于是通过 `memory_region_allocate_system_memory` 来申请内存。

接下来的调用链为：`memory_region_allocate_system_memory->allocate_system_memory_nonnuma->memory_region_init_ram_nomigrate->memory_region_init_ram_shared_nomigrate`。

 复制代码

```

1 void memory_region_init_ram_shared_nomigrate(MemoryRegion *mr,
2                                             Object *owner,
3                                             const char *name,
4                                             uint64_t size,
5                                             bool share,
6                                             Error **errp)
7 {
8     Error *err = NULL;
9     memory_region_init(mr, owner, name, size);
10    mr->ram = true;
11    mr->terminates = true;
12    mr->destructor = memory_region_destructor_ram;
13    mr->ram_block = qemu_ram_alloc(size, share, mr, &err);
14    .....
15 }

```




```

16
17 static
18 RAMBlock *qemu_ram_alloc_internal(ram_addr_t size, ram_addr_t max_size, void (*resized)
19 {
20     RAMBlock *new_block;
21     size = HOST_PAGE_ALIGN(size);
22     max_size = HOST_PAGE_ALIGN(max_size);
23     new_block = g_malloc0(sizeof(*new_block));
24     new_block->mr = mr;
25     new_block->resized = resized;
26     new_block->used_length = size;
27     new_block->max_length = max_size;
28     new_block->fd = -1;
29     new_block->page_size = getpagesize();
30     new_block->host = host;
31     .....
32     ram_block_add(new_block, &local_err, share);
33     return new_block;
34 }
35
36 static void ram_block_add(RAMBlock *new_block, Error **errp, bool shared)
37 {
38     RAMBlock *block;
39     RAMBlock *last_block = NULL;
40     ram_addr_t old_ram_size, new_ram_size;
41     Error *err = NULL;
42     old_ram_size = last_ram_page();
43     new_block->offset = find_ram_offset(new_block->max_length);
44     if (!new_block->host) {
45         new_block->host = phys_mem_alloc(new_block->max_length, &new_block->mr->align, :
46     .....
47     }
48     }
49     .....
50 }

```

这里面，我们会调用 `qemu_ram_alloc`，创建一个 `RAMBlock` 用来表示内存块。这里面调用 `ram_block_add->phys_mem_alloc`。`phys_mem_alloc` 是一个函数指针，指向函数 `qemu_anon_ram_alloc`，这里面调用 `qemu_ram_mmap`，在 `qemu_ram_mmap` 中调用 `mmap` 分配内存。

 复制代码

```

1 static void *(*phys_mem_alloc)(size_t size, uint64_t *align, bool shared) = qemu_anon_r
2
3 void *qemu_anon_ram_alloc(size_t size, uint64_t *alignment, bool shared)
4 {

```



```


5     size_t align = QEMU_VMALLOC_ALIGN;
6     void *ptr = qemu_ram_mmap(-1, size, align, shared);
7     .....
8     if (alignment) {
9         *alignment = align;
10    }
11    return ptr;
12 }
13
14 void *qemu_ram_mmap(int fd, size_t size, size_t align, bool shared)
15 {
16     int flags;
17     int guardfd;
18     size_t offset;
19     size_t pagesize;
20     size_t total;
21     void *guardptr;
22     void *ptr;
23     .....
24     total = size + align;
25     guardfd = -1;
26     pagesize = getpagesize();
27     flags = MAP_PRIVATE | MAP_ANONYMOUS;
28     guardptr = mmap(0, total, PROT_NONE, flags, guardfd, 0);
29     .....
30     flags = MAP_FIXED;
31     flags |= fd == -1 ? MAP_ANONYMOUS : 0;
32     flags |= shared ? MAP_SHARED : MAP_PRIVATE;
33     offset = QEMU_ALIGN_UP((uintptr_t)guardptr, align) - (uintptr_t)guardptr;
34     ptr = mmap(guardptr + offset, size, PROT_READ | PROT_WRITE, flags, fd, 0);
35     .....
36     return ptr;
37 }

```

我们回到 `pc_memory_init`，通过 `memory_region_allocate_system_memory` 申请到内存以后，为了兼容过去的版本，我们分成两个 `MemoryRegion` 进行管理，一个是 `ram_below_4g`，一个是 `ram_above_4g`。对于这两个 `MemoryRegion`，我们都会初始化一个 `alias`，也即别名，意思是说，两个 `MemoryRegion` 其实都指向 `memory_region_allocate_system_memory` 分配的内存，只不过分成两个部分，起两个别名指向不同的区域。

这两部分 `MemoryRegion` 都会调用 `memory_region_add_subregion`，将这两部分作为子的内存区域添加到 `system_memory` 这棵树上。


接下来的调用链为：memory_region_add_subregion->memory_region_add_subregion_common->memory_region_update_container_subregions。

 复制代码

```
1 static void memory_region_update_container_subregions(MemoryRegion *subregion)
2 {
3     MemoryRegion *mr = subregion->container;
4     MemoryRegion *other;
5
6     memory_region_transaction_begin();
7
8     memory_region_ref(subregion);
9     QTAILQ_FOREACH(other, &mr->subregions, subregions_link) {
10         if (subregion->priority >= other->priority) {
11             QTAILQ_INSERT_BEFORE(other, subregion, subregions_link);
12             goto done;
13         }
14     }
15     QTAILQ_INSERT_TAIL(&mr->subregions, subregion, subregions_link);
16 done:
17     memory_region_update_pending |= mr->enabled && subregion->enabled;
18     memory_region_transaction_commit();
19 }
```

在 memory_region_update_container_subregions 中，我们会将子区域放到链表中，然后调用 memory_region_transaction_commit。在这里面，我们会调用 address_space_set_flatview。因为内存区域变了，flatview 也会变，就像上面分析过的一样，listener 会被调用。

因为添加了一个 MemoryRegion，region_add 也即 kvm_region_add。

 复制代码

```
1 static void kvm_region_add(MemoryListener *listener,
2                             MemoryRegionSection *section)
3 {
4     KVMMemoryListener *kml = container_of(listener, KVMMemoryListener, listener);
5     kvm_set_phys_mem(kml, section, true);
6 }
7
8 static void kvm_set_phys_mem(KVMMemoryListener *kml,
9                             MemoryRegionSection *section, bool add)
```


```

10 {
11     KVMSlot *mem;
12     int err;
13     MemoryRegion *mr = section->mr;
14     bool writeable = !mr->readonly && !mr->rom_device;
15     hwaddr start_addr, size;
16     void *ram;
17     .....
18     size = kvm_align_section(section, &start_addr);
19     .....
20     /* use aligned delta to align the ram address */
21     ram = memory_region_get_ram_ptr(mr) + section->offset_within_region + (start_addr -
22     .....
23     /* register the new slot */
24     mem = kvm_alloc_slot(kml);
25     mem->memory_size = size;
26     mem->start_addr = start_addr;
27     mem->ram = ram;
28     mem->flags = kvm_mem_flags(mr);
29
30     err = kvm_set_user_memory_region(kml, mem, true);
31     .....
32 }

```

kvm_region_add 调用的是 kvm_set_phys_mem，这里面分配一个用于放这块内存的 KVMSlot 结构，就像一个内存条一样，当然这是在用户态模拟出来的内存条，放在 KVMState 结构里面。这个结构是我们上一节创建虚拟机的时候创建的。

接下来，kvm_set_user_memory_region 就会将用户态模拟出来的内存条，和内核中的 KVM 模块关联起来。

 复制代码

```

1 static int kvm_set_user_memory_region(KVMState *kml, KVMSlot *slot, bool new)
2 {
3     KVMState *s = kvm_state;
4     struct kvm_userspace_memory_region mem;
5     int ret;
6
7     mem.slot = slot->slot | (kml->as_id << 16);
8     mem.guest_phys_addr = slot->start_addr;
9     mem.userspace_addr = (unsigned long)slot->ram;
10    mem.flags = slot->flags;
11    .....
12    mem.memory_size = slot->memory_size;
13    ret = kvm_vm_ioctl(s, KVM_SET_USER_MEMORY_REGION, &mem);


```

```

14     slot->old_flags = mem.flags;
15     .....
16     return ret;
17 }

```

终于，在这里，我们又看到了可以和内核通信的 `kvm_vm_ioctl`。我们来看内核收到 `KVM_SET_USER_MEMORY_REGION` 会做哪些事情。


 复制代码

```

1 static long kvm_vm_ioctl(struct file *filp,
2                          unsigned int ioctl, unsigned long arg)
3 {
4     struct kvm *kvm = filp->private_data;
5     void __user *argp = (void __user *)arg;
6     switch (ioctl) {
7     case KVM_SET_USER_MEMORY_REGION: {
8         struct kvm_userspace_memory_region kvm_userspace_mem;
9         if (copy_from_user(&kvm_userspace_mem, argp,
10                          sizeof(kvm_userspace_mem)))
11             goto out;
12         r = kvm_vm_ioctl_set_memory_region(kvm, &kvm_userspace_mem);
13         break;
14     }
15     .....
16 }

```

接下来的调用链为：`kvm_vm_ioctl_set_memory_region`->`kvm_set_memory_region`->`__kvm_set_memory_region`。

 复制代码

```

1 int __kvm_set_memory_region(struct kvm *kvm,
2                             const struct kvm_userspace_memory_region *mem)
3 {
4     int r;
5     gfn_t base_gfn;
6     unsigned long npages;
7     struct kvm_memory_slot *slot;
8     struct kvm_memory_slot old, new;
9     struct kvm_memslots *slots = NULL, *old_memslots;
10    int as_id, id;
11    enum kvm_mr_change change;
12    .....

```

```

13     as_id = mem->slot >> 16;
14     id = (u16)mem->slot;
15
16     slot = id_to_memslot(__kvm_memslots(kvm, as_id), id);
17     base_gfn = mem->guest_phys_addr >> PAGE_SHIFT;
18     npages = mem->memory_size >> PAGE_SHIFT;
19     .....
20     new = old = *slot;
21
22     new.id = id;
23     new.base_gfn = base_gfn;
24     new.npages = npages;
25     new.flags = mem->flags;
26     .....
27     if (change == KVM_MR_CREATE) {
28         new.userspace_addr = mem->userspace_addr;
29
30         if (kvm_arch_create_memslot(kvm, &new, npages))
31             goto out_free;
32     }
33     .....
34     slots = kvzalloc(sizeof(struct kvm_memslots), GFP_KERNEL);
35     memcpy(slots, __kvm_memslots(kvm, as_id), sizeof(struct kvm_memslots));
36     .....
37     r = kvm_arch_prepare_memory_region(kvm, &new, mem, change);
38
39     update_memslots(slots, &new);
40     old_memslots = install_new_memslots(kvm, as_id, slots);
41
42     kvm_arch_commit_memory_region(kvm, mem, &old, &new, change);
43     return 0;
44     .....
45 }

```

在用户态每个 KVMState 有多个 KVMSlot，在内核里面，同样每个 struct kvm 也有多个 struct kvm_memory_slot，两者是对应起来的。

 复制代码

```

1 // 用户态
2 struct KVMState
3 {
4     .....
5     int nr_slots;
6     .....
7     KVMMemoryListener memory_listener;
8     .....
9 };

```

```

10
11 typedef struct KVMMemoryListener {
12     MemoryListener listener;
13     KVMSlot *slots;
14     int as_id;
15 } KVMMemoryListener
16
17 typedef struct KVMSlot
18 {
19     hwaddr start_addr;
20     ram_addr_t memory_size;
21     void *ram;
22     int slot;
23     int flags;
24     int old_flags;
25 } KVMSlot;
26
27 // 内核态
28 struct kvm {
29     spinlock_t mmu_lock;
30     struct mutex slots_lock;
31     struct mm_struct *mm; /* userspace tied to this vm */
32     struct kvm_memslots __rcu *memslots[KVM_ADDRESS_SPACE_NUM];
33     .....
34 }
35
36 struct kvm_memslots {
37     u64 generation;
38     struct kvm_memory_slot memslots[KVM_MEM_SLOTS_NUM];
39     /* The mapping table from slot id to the index in memslots[]. */
40     short id_to_index[KVM_MEM_SLOTS_NUM];
41     atomic_t lru_slot;
42     int used_slots;
43 };
44
45 struct kvm_memory_slot {
46     gfn_t base_gfn; // 根据 guest_phys_addr 计算
47     unsigned long npages;
48     unsigned long *dirty_bitmap;
49     struct kvm_arch_memory_slot arch;
50     unsigned long userspace_addr;
51     u32 flags;
52     short id;
53 };

```

并且，`id_to_memslot` 函数可以根据用户态的 slot 号得到内核态的 slot 结构。

如果传进来的参数是 KVM_MR_CREATE，表示要创建一个新的内存条，就会调用 kvm_arch_create_memslot 来创建 kvm_memory_slot 的成员 kvm_arch_memory_slot。

接下来就是创建 kvm_memslots 结构，填充这个结构，然后通过 install_new_memslots 将这个新的内存条，添加到 struct kvm 结构中。

至此，用户态的内存结构和内核态的内存结构算是对应了起来。

页面分配和映射

上面对于内存的管理，还只是停留在元数据的管理。对于内存的分配与映射，我们还没有涉及，接下来，我们就来看看，页面是如何进行分配和映射的。

上面咱们说了，内存映射对于虚拟机来讲是一件非常麻烦的事情，从 GVA 到 GPA 到 HVA 到 HPA，性能很差，为了解决这个问题，有两种主要的思路。

影子页表

第一种方式就是软件的方式，**影子页表**（Shadow Page Table）。

按照咱们在内存管理那一节讲的，内存映射要通过页表来管理，页表地址应该放在 cr3 寄存器里面。本来的过程是，客户机要通过 cr3 找到客户机的页表，实现从 GVA 到 GPA 的转换，然后在宿主机上，要通过 cr3 找到宿主机的页表，实现从 HVA 到 HPA 的转换。

为了实现客户机虚拟地址空间到宿主机物理地址空间的直接映射。客户机中每个进程都有自己的虚拟地址空间，所以 KVM 需要为客户机中的每个进程页表都要维护一套相应的影子页表。

在客户机访问内存时，使用的不是客户机的原来的页表，而是这个页表对应的影子页表，从而实现了从客户机虚拟地址到宿主机物理地址的直接转换。而且，在 TLB 和 CPU 缓存上缓存的是来自影子页表中客户机虚拟地址和宿主机物理地址之间的映射，也因此提高了缓存的效率。

但是影子页表的引入也意味着 KVM 需要为每个客户机的每个进程的页表都要维护一套相应的影子页表，内存占用比较大，而且客户机页表和影子页表也需要进行实时同步。

扩展页表

于是就有了第二种方式，就是硬件的方式，Intel 的 EPT (Extent Page Table , 扩展页表) 技术。

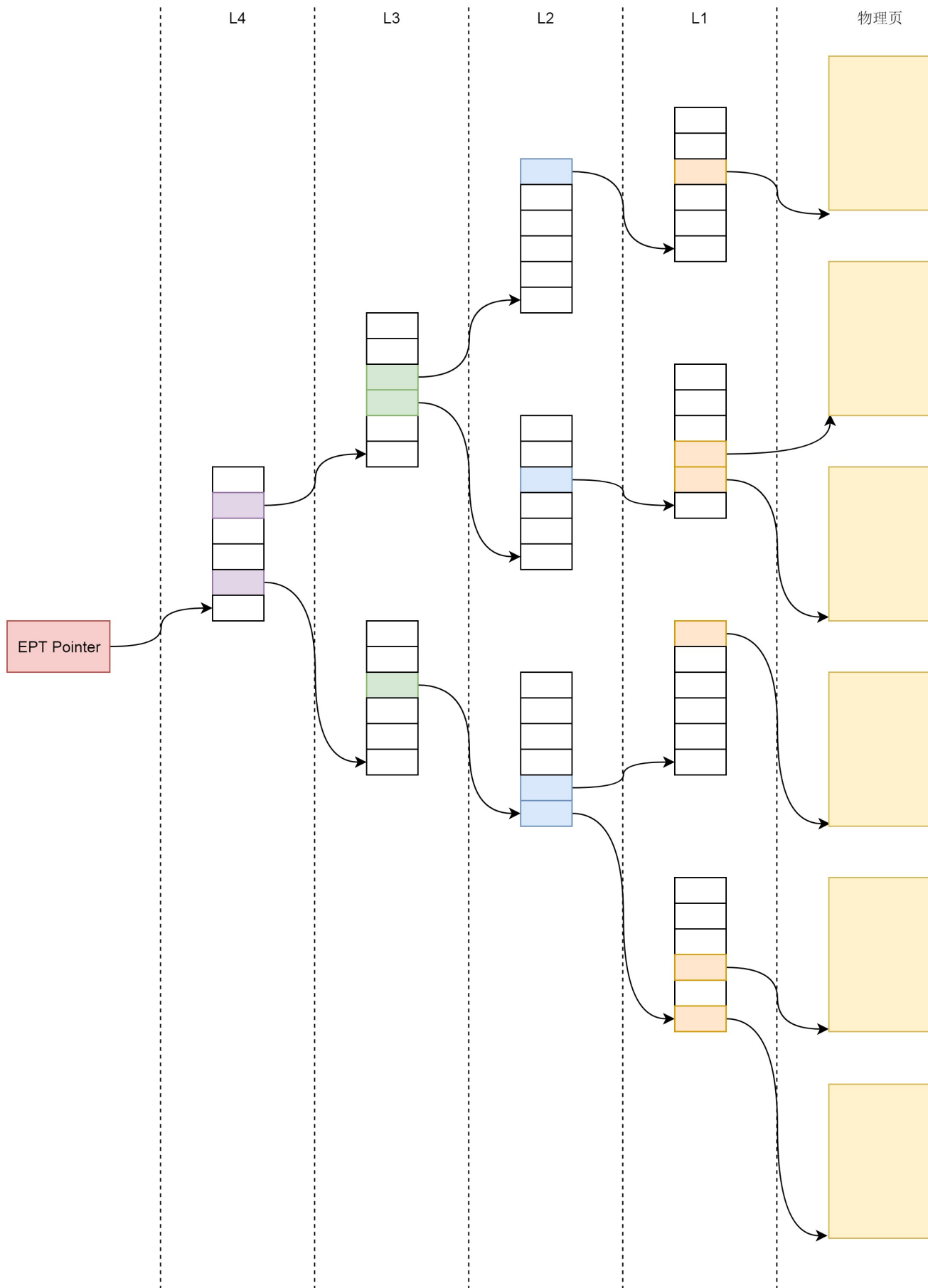
EPT 在原有客户机页表对客户机虚拟地址到客户机物理地址映射的基础上，又引入了 EPT 页表来实现客户机物理地址到宿主机物理地址的另一次映射。客户机运行时，客户机页表被载入 CR3，而 EPT 页表被载入专门的 EPT 页表指针寄存器 EPTP。

有了 EPT，在客户机物理地址到宿主机物理地址转换的过程中，缺页会产生 EPT 缺页异常。KVM 首先根据引起异常的客户机物理地址，映射到对应的宿主机虚拟地址，然后为此虚拟地址分配新的物理页，最后 KVM 再更新 EPT 页表，建立起引起异常的客户机物理地址到宿主机物理地址之间的映射。

KVM 只需为每个客户机维护一套 EPT 页表，也大大减少了内存的开销。


这里，我们重点看第二种方式。因为使用了 EPT 之后，客户机里面的页表映射，也即从 GVA 到 GPA 的转换，还是用传统的方式，和在内存管理那一章讲的没有什么区别。而 EPT 重点帮我们解决的就是从 GPA 到 HPA 的转换问题。因为要经过两次页表，所以 EPT 又 tdp(two dimentional paging)。

EPT 的页表结构也是分为四层，EPT Pointer (EPTP) 指向 PML4 的首地址。



管理物理页面的 Page 结构和咱们讲内存管理那一章是一样的。EPT 页表也需要存放在一个页中，这些页要用 `kvm_mmu_page` 这个结构来管理。


当一个虚拟机运行，进入客户机模式的时候，我们上一节解析过，它会调用 `vcpu_enter_guest` 函数，这里面会调用 `kvm_mmu_reload->kvm_mmu_load`。

 复制代码

```
1 int kvm_mmu_load(struct kvm_vcpu *vcpu)
2 {
3     .....
4     r = mmu_topup_memory_caches(vcpu);
5     r = mmu_alloc_roots(vcpu);
6     kvm_mmu_sync_roots(vcpu);
7     /* set_cr3() should ensure TLB has been flushed */
8     vcpu->arch.mmu.set_cr3(vcpu, vcpu->arch.mmu.root_hpa);
9     .....
10 }
11
12 static int mmu_alloc_roots(struct kvm_vcpu *vcpu)
13 {
14     if (vcpu->arch.mmu.direct_map)
15         return mmu_alloc_direct_roots(vcpu);
16     else
17         return mmu_alloc_shadow_roots(vcpu);
18 }
19
20 static int mmu_alloc_direct_roots(struct kvm_vcpu *vcpu)
21 {
22     struct kvm_mmu_page *sp;
23     unsigned i;
24
25     if (vcpu->arch.mmu.shadow_root_level == PT64_ROOT_LEVEL) {
26         spin_lock(&vcpu->kvm->mmu_lock);
27         make_mmu_pages_available(vcpu);
28         sp = kvm_mmu_get_page(vcpu, 0, 0, PT64_ROOT_LEVEL, 1, ACC_ALL);
29         ++sp->root_count;
30         spin_unlock(&vcpu->kvm->mmu_lock);
31         vcpu->arch.mmu.root_hpa = __pa(sp->spt);
32     }
33     .....
34 }
```


这里构建的是页表的根部，也即顶级页表，并且设置 `cr3` 来刷新 TLB。`mmu_alloc_roots` 会调用 `mmu_alloc_direct_roots`，因为我们用的是 EPT 模式，而非影子表。在 `mmu_alloc_direct_roots` 中，`kvm_mmu_get_page` 会分配一个 `kvm_mmu_page`，来存放顶级页表项。

接下来，当虚拟机真的要访问内存的时候，会发现有的页表没有建立，有的物理页没有分配，这都会触发缺页异常，在 KVM 里面会发送 VM-Exit，从客户机模式转换为宿主机模式，来修复这个缺失的页表或者物理页。

 复制代码


```
1 static int (*const kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu) = {
2     [EXIT_REASON_EXCEPTION_NMI]          = handle_exception,
3     [EXIT_REASON_EXTERNAL_INTERRUPT]      = handle_external_interrupt,
4     [EXIT_REASON_IO_INSTRUCTION]         = handle_io,
5     .....
6     [EXIT_REASON_EPT_VIOLATION]          = handle_ept_violation,
7     .....
8 }
```

咱们前面讲过，虚拟机退出客户机模式有很多原因，例如接收到中断、接收到 I/O 等，EPT 的缺页异常也是一种类型，我们称为 EXIT_REASON_EPT_VIOLATION，对应的处理函数是 handle_ept_violation。

 复制代码

```
1 static int handle_ept_violation(struct kvm_vcpu *vcpu)
2 {
3     gpa_t gpa;
4     .....
5     gpa = vmcs_read64(GUEST_PHYSICAL_ADDRESS);
6     .....
7     vcpu->arch.gpa_available = true;
8     vcpu->arch.exit_qualification = exit_qualification;
9
10    return kvm_mmu_page_fault(vcpu, gpa, error_code, NULL, 0);
11 }
12
13 int kvm_mmu_page_fault(struct kvm_vcpu *vcpu, gva_t cr2, u64 error_code,
14                        void *insn, int insn_len)
15 {
16     .....
17     r = vcpu->arch.mmu.page_fault(vcpu, cr2, lower_32_bits(error_code), false);
18     .....
19 }
```


在 `handle_ept_violation` 里面，我们从 `VMCS` 中得到没有解析成功的 `GPA`，也即客户机的物理地址，然后调用 `kvm_mmu_page_fault`，看为什么解析不成功。
`kvm_mmu_page_fault` 会调用 `page_fault` 函数，其实是 `tdp_page_fault` 函数。`tdp` 的意思就是 `EPT`，前面我们解释过了。

 复制代码

```
1 static int tdp_page_fault(struct kvm_vcpu *vcpu, gva_t gpa, u32 error_code, bool prefau:
2 {
3     kvm_pfn_t pfn;
4     int r;
5     int level;
6     bool force_pt_level;
7     gfn_t gfn = gpa >> PAGE_SHIFT;
8     unsigned long mmu_seq;
9     int write = error_code & PFERR_WRITE_MASK;
10    bool map_writable;
11
12    r = mmu_topup_memory_caches(vcpu);
13    level = mapping_level(vcpu, gfn, &force_pt_level);
14    .....
15    if (try_async_pf(vcpu, prefault, gfn, gpa, &pfn, write, &map_writable))
16        return 0;
17
18    if (handle_abnormal_pfn(vcpu, 0, gfn, pfn, ACC_ALL, &r))
19        return r;
20
21    make_mmu_pages_available(vcpu);
22    r = __direct_map(vcpu, write, map_writable, level, gfn, pfn, prefault);
23    .....
24 }
```

既然没有映射，就应该加上映射，`tdp_page_fault` 就是干这个事情的。

在 `tdp_page_fault` 这个函数开头，我们通过 `gpa`，也即客户机的物理地址得到客户机的页号 `gfn`。接下来，我们要通过调用 `try_async_pf` 得到宿主机的物理地址对应的页号，也即真正的物理页的页号，然后通过 `__direct_map` 将两者关联起来。

 复制代码

```
1 static bool try_async_pf(struct kvm_vcpu *vcpu, bool prefault, gfn_t gfn, gva_t gva, kvr
2 {
3     struct kvm_memory_slot *slot;
4     bool async;
```



```

3 {
4     struct page *page[1];
5     int npages = 0;
6     .....
7     if (async) {
8         npages = get_user_page_nowait(addr, write_fault, page);
9     } else {
10    .....
11        npages = get_user_pages_unlocked(addr, 1, page, flags);
12    }
13    .....
14    *pfn = page_to_pfn(page[0]);
15    return npages;
16 }


```

在 `hva_to_pfn_slow` 中，我们要先调用 `get_user_page_nowait`，得到一个物理页面，然后再调用 `page_to_pfn` 将物理页面转换成为物理页号。

无论是哪一种 `get_user_pages_XXX`，最终都会调用 `__get_user_pages` 函数。这里面会调用 `faultin_page`，在 `faultin_page` 中我们会调用 `handle_mm_fault`。看到这个是不是很熟悉？这就是咱们内存管理那一章讲的缺页异常的逻辑，分配一个物理内存。

至此，`try_async_pf` 得到了物理页面，并且转换为对应的物理页号。

接下来，`__direct_map` 会关联客户机物理页号和宿主机物理页号。

 复制代码

```

1 static int __direct_map(struct kvm_vcpu *vcpu, int write, int map_writable,
2                        int level, gfn_t gfn, kvm_pfn_t pfn, bool prefault)
3 {
4     struct kvm_shadow_walk_iterator iterator;
5     struct kvm_mmu_page *sp;
6     int emulate = 0;
7     gfn_t pseudo_gfn;
8
9     if (!VALID_PAGE(vcpu->arch.mmu.root_hpa))
10        return 0;
11
12    for_each_shadow_entry(vcpu, (u64)gfn << PAGE_SHIFT, iterator) {
13        if (iterator.level == level) {
14            emulate = mmu_set_spte(vcpu, iterator.sptep, ACC_ALL,
15                                write, level, gfn, pfn, prefault,
16                                map_writable);

```



```

17         direct_pte_prefetch(vcpu, iterator.sptep);
18         ++vcpu->stat.pf_fixed;
19         break;
20     }
21
22     drop_large_spte(vcpu, iterator.sptep);
23     if (!is_shadow_present_pte(*iterator.sptep)) {
24         u64 base_addr = iterator.addr;
25
26         base_addr &= PT64_LVL_ADDR_MASK(iterator.level);
27         pseudo_gfn = base_addr >> PAGE_SHIFT;
28         sp = kvm_mmu_get_page(vcpu, pseudo_gfn, iterator.addr,
29                               iterator.level - 1, 1, ACC_ALL);
30
31         link_shadow_page(vcpu, iterator.sptep, sp);
32     }
33 }
34 return emulate;
35 }

```

`__direct_map` 首先判断页表的根是否存在，当然存在，我们刚才初始化了。

接下来是 `for_each_shadow_entry` 一个循环。每一个循环中，先是会判断需要映射的 `level`，是否正是当前循环的这个 `iterator.level`。如果是，则说明是叶子节点，直接映射真正的物理页面 `pfn`，然后退出。接着是非叶子节点的情形，判断如果这一项指向的页表项不存在，就要建立页表项，通过 `kvm_mmu_get_page` 得到保存页表项的页面，然后将这一项指向下一级的页表页面。

至此，内存映射就结束了。

总结时刻

我们这里来总结一下，虚拟机的内存管理也是需要用户态的 `qemu` 和内核态的 `KVM` 共同完成。为了加速内存映射，需要借助硬件的 `EPT` 技术。

在用户态 `qemu` 中，有一个结构 `AddressSpace address_space_memory` 来表示虚拟机的系统内存，这个内存可能包含多个内存区域 `struct MemoryRegion`，组成树形结构，指向由 `mmap` 分配的虚拟内存。

在 AddressSpace 结构中，有一个 struct KVMMemoryListener，当有新的内存区域添加的时候，会被通知调用 kvm_region_add 来通知内核。

在用户态 qemu 中，对于虚拟机有一个结构 struct KVMState 表示这个虚拟机，这个结构会指向一个数组的 struct KVMSlot 表示这个虚拟机的多个内存条，KVMSlot 中有一个 void *ram 指针指向 mmap 分配的那块虚拟内存。

kvm_region_add 是通过 ioctl 来通知内核 KVM 的，会给内核 KVM 发送一个 KVM_SET_USER_MEMORY_REGION 消息，表示用户态 qemu 添加了一个内存区域，内核 KVM 也应该添加一个相应的内存区域。

和用户态 qemu 对应的内核 KVM，对于虚拟机有一个结构 struct kvm 表示这个虚拟机，这个结构会指向一个数组的 struct kvm_memory_slot 表示这个虚拟机的多个内存条，kvm_memory_slot 中有起始页号，页面数目，表示这个虚拟机的物理内存空间。

虚拟机的物理内存空间里面的页面当然不是一开始就映射到物理页面的，只有当虚拟机的内存被访问的时候，也即 mmap 分配的虚拟内存空间被访问的时候，先查看 EPT 页表，是否已经映射过，如果已经映射过，则经过四级页表映射，就能访问到物理页面。

如果没有映射过，则虚拟机会通过 VM-Exit 指令回到宿主机模式，通过 handle_ept_violation 补充页表映射。先是通过 handle_mm_fault 为虚拟机的物理内存空间分配真正的物理页面，然后通过 __direct_map 添加 EPT 页表映射。

欢迎留言和我分享你的疑惑和见解，也欢迎收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 51 | 计算虚拟化之CPU（下）：如何复用集团的人力资源？

下一篇 53 | 存储虚拟化（上）：如何建立自己保管的单独档案库？

精选留言 (5)

写留言



没心没肺

2019-07-28

快要被劝退了，硬着头皮看。

展开



LDxy

2019-07-27

真难

展开





浪子

2019-07-27

超哥会讲overcommit吗

展开 ▾



sunyunjian

2019-07-26

这配图是网易吗

展开 ▾



小龙的城堡

2019-07-26

把c语言写出面相对象，确实有意思！

展开 ▾

