

## 51 | 计算虚拟化之CPU（下）：如何复用集团的人力资源？

2019-07-24 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 19:48 大小 18.14M



上一节 qemu 初始化的 main 函数，我们解析了一个开头，得到了表示体系结构的 MachineClass 以及 MachineState。

### 4. 初始化块设备


我们接着回到 main 函数，接下来初始化的是块设备，调用的是 configure\_blockdev。这里我们需要重点关注上面参数中的硬盘，不过我们放在存储虚拟化那一节再解析。

复制代码

```
1 configure_blockdev(&bdo_queue, machine_class, snapshot);
```

## 5. 初始化计算虚拟化的加速模式

接下来初始化的是计算虚拟化的加速模式，也即要不要使用 KVM。根据参数中的配置是启用 KVM。这里调用的是 `configure_accelerator`。


 复制代码

```
1 configure_accelerator(current_machine, argv[0]);
2
3 void configure_accelerator(MachineState *ms, const char *progrname)
4 {
5     const char *accel;
6     char **accel_list, **tmp;
7     int ret;
8     bool accel_initialised = false;
9     bool init_failed = false;
10    AccelClass *acc = NULL;
11
12    accel = qemu_opt_get(qemu_get_machine_opts(), "accel");
13    accel = "kvm";
14    accel_list = g_strsplit(accel, ":", 0);
15
16    for (tmp = accel_list; !accel_initialised && tmp && *tmp; tmp++) {
17        acc = accel_find(*tmp);
18        ret = accel_init_machine(acc, ms);
19    }
20 }
21
22 static AccelClass *accel_find(const char *opt_name)
23 {
24     char *class_name = g_strdup_printf(ACCEL_CLASS_NAME("%s"), opt_name);
25     AccelClass *ac = ACCEL_CLASS(object_class_by_name(class_name));
26     g_free(class_name);
27     return ac;
28 }
29
30 static int accel_init_machine(AccelClass *acc, MachineState *ms)
31 {
32     ObjectClass *oc = OBJECT_CLASS(acc);
33     const char *cname = object_class_get_name(oc);
34     AccelState *accel = ACCEL(object_new(cname));
35     int ret;
36     ms->accelerator = accel;
37     *(acc->allowed) = true;
38     ret = acc->init_machine(ms);
39     return ret;
40 }
```

在 `configure_accelerator` 中，我们看命令行参数里面的 `accel`，发现是 `kvm`，则调用 `accel_find` 根据名字，得到相应的纸面上的 `class`，并初始化为 `Class` 类。


`MachineClass` 是计算机体系结构的 `Class` 类，同理，`AccelClass` 就是加速器的 `Class` 类，然后调用 `accel_init_machine`，通过 `object_new`，将 `AccelClass` 这个 `Class` 类实例化为 `AccelState`，类似对于体系结构的实例是 `MachineState`。

在 `accel_find` 中，我们会根据名字 `kvm`，找到纸面上的 `class`，也即 `kvm_accel_type`，然后调用 `type_initialize`，里面调用 `kvm_accel_type` 的 `class_init` 方法，也即 `kvm_accel_class_init`。

 复制代码

```
1 static void kvm_accel_class_init(ObjectClass *oc, void *data)
2 {
3     AccelClass *ac = ACCEL_CLASS(oc);
4     ac->name = "KVM";
5     ac->init_machine = kvm_init;
6     ac->allowed = &kvm_allowed;
7 }
```

在 `kvm_accel_class_init` 中，我们创建 `AccelClass`，将 `init_machine` 设置为 `kvm_init`。在 `accel_init_machine` 中其实就调用了这个 `init_machine` 函数，也即调用 `kvm_init` 方法。

 复制代码

```
1 static int kvm_init(MachineState *ms)
2 {
3     MachineClass *mc = MACHINE_GET_CLASS(ms);
4     int soft_vcpus_limit, hard_vcpus_limit;
5     KVMState *s;
6     const KVMCapabilityInfo *missing_cap;
7     int ret;
8     int type = 0;
9     const char *kvm_type;
10
11     s = KVM_STATE(ms->accelerator);
12     s->fd = qemu_open("/dev/kvm", O_RDWR);
13     ret = kvm_ioctl(s, KVM_GET_API_VERSION, 0);
14     .....
15     do {
16         ret = kvm_ioctl(s, KVM_CREATE_VM, type);
```

```

17     } while (ret == -EINTR);
18     .....
19     s->vmfd = ret;
20
21     /* check the vcpu limits */
22     soft_vcpus_limit = kvm_recommended_vcpus(s);
23     hard_vcpus_limit = kvm_max_vcpus(s);
24     .....
25     ret = kvm_arch_init(ms, s);
26     if (ret < 0) {
27         goto err;
28     }
29
30     if (machine_kernel_irqchip_allowed(ms)) {
31         kvm_irqchip_create(ms, s);
32     }
33     .....
34     return 0;
35 }
36

```

这里面的操作就从用户态到内核态的 KVM 了。就像前面原理讲过的一样，用户态使用内核态 KVM 的能力，需要打开一个文件 `/dev/kvm`，这是一个字符设备文件，打开一个字符设备文件的过程我们讲过，这里不再赘述。

 复制代码

```

1 static struct miscdevice kvm_dev = {
2     KVM_MINOR,
3     "kvm",
4     &kvm_chardev_ops,
5 };
6
7 static struct file_operations kvm_chardev_ops = {
8     .unlocked_ioctl = kvm_dev_ioctl,
9     .compat_ioctl   = kvm_dev_ioctl,
10    .llseek         = noop_llseek,
11 };
12

```

KVM 这个字符设备文件定义了一个字符设备文件的操作函数 `kvm_chardev_ops`，这里面只定义了 `ioctl` 的操作。


接下来，用户态就通过 `ioctl` 系统调用，调用到 `kvm_dev_ioctl` 这个函数。这个过程我们在[字符设备](#)那一节也讲了。

 复制代码

```
1 static long kvm_dev_ioctl(struct file *filp,
2     unsigned int ioctl, unsigned long arg)
3 {
4     long r = -EINVAL;
5
6     switch (ioctl) {
7     case KVM_GET_API_VERSION:
8         r = KVM_API_VERSION;
9         break;
10    case KVM_CREATE_VM:
11        r = kvm_dev_ioctl_create_vm(arg);
12        break;
13    case KVM_CHECK_EXTENSION:
14        r = kvm_vm_ioctl_check_extension_generic(NULL, arg);
15        break;
16    case KVM_GET_VCPU_MMAP_SIZE:
17        r = PAGE_SIZE;    /* struct kvm_run */
18        break;
19    .....
20    }
21 out:
22    return r;
23 }
24
```

我们可以看到，在用户态 `qemu` 中，调用 `KVM_GET_API_VERSION` 查看版本号，内核就有相应的分支，返回版本号，如果能够匹配上，则调用 `KVM_CREATE_VM` 创建虚拟机。

创建虚拟机，需要调用 `kvm_dev_ioctl_create_vm`。

 复制代码

```
1 static int kvm_dev_ioctl_create_vm(unsigned long type)
2 {
3     int r;
4     struct kvm *kvm;
5     struct file *file;
6
7     kvm = kvm_create_vm(type);
8     .....
9     r = get_unused_fd_flags(O_CLOEXEC);
```

```

10 .....
11     file = anon_inode_getfile("kvm-vm", &kvm_vm_fops, kvm, O_RDWR);
12 .....
13     fd_install(r, file);
14     return r;
15 }

```

在 `kvm_dev_ioctl_create_vm` 中，首先调用 `kvm_create_vm` 创建一个 `struct kvm` 结构。这个结构在内核里面代表一个虚拟机。

从下面结构的定义里，我们可以看到，这里面有 `vcpu`，有 `mm_struct` 结构。这个结构本来用来管理进程的内存的。虚拟机也是一个进程，所以虚拟机的用户进程空间也是用它来表示。虚拟机里面的操作系统以及应用的进程空间不归它管。

在 `kvm_dev_ioctl_create_vm` 中，第二件事情就是创建一个文件描述符，和 `struct file` 关联起来，这个 `struct file` 的 `file_operations` 会被设置为 `kvm_vm_fops`。

 复制代码

```


1 struct kvm {
2     struct mm_struct *mm; /* userspace tied to this vm */
3     struct kvm_memslots __rcu *memslots[KVM_ADDRESS_SPACE_NUM];
4     struct kvm_vcpu *vcpus[KVM_MAX_VCPUS];
5     atomic_t online_vcpus;
6     int created_vcpus;
7     int last_boosted_vcpu;
8     struct list_head vm_list;
9     struct mutex lock;
10    struct kvm_io_bus __rcu *buses[KVM_NR_BUSES];
11 .....
12    struct kvm_vm_stat stat;
13    struct kvm_arch arch;
14    refcount_t users_count;
15 .....
16    long tlbs_dirty;
17    struct list_head devices;
18    pid_t userspace_pid;
19 };
20
21 static struct file_operations kvm_vm_fops = {
22     .release          = kvm_vm_release,
23     .unlocked_ioctl  = kvm_vm_ioctl,
24     .llseek          = noop_llseek,
25 };

```

kvm\_dev\_ioctl\_create\_vm 结束之后，对于一台虚拟机而言，只是在内核中有一个数据结构，对于相应的资源还没有分配，所以我们还需要接着看。

## 6. 初始化网络设备


接下来，调用 net\_init\_clients 进行网络设备的初始化。我们可以解析 net 参数，也会在 net\_init\_clients 中解析 netdev 参数。这属于网络虚拟化的部分，我们先暂时放一下。

 复制代码

```
1 int net_init_clients(Error **errp)
2 {
3     QTAILQ_INIT(&net_clients);
4     if (qemu_opts_foreach(qemu_find_opts("netdev"),
5                           net_init_netdev, NULL, errp)) {
6         return -1;
7     }
8     if (qemu_opts_foreach(qemu_find_opts("nic"), net_param_nic, NULL, errp)) {
9         return -1;
10    }
11    if (qemu_opts_foreach(qemu_find_opts("net"), net_init_client, NULL, errp)) {
12        return -1;
13    }
14    return 0;
15 }
```

## 7. CPU 虚拟化

接下来，我们要调用 machine\_run\_board\_init。这里面调用了 MachineClass 的 init 函数。盼啊盼才到了它，这才调用了 pc\_init1。


 复制代码

```
1 void machine_run_board_init(MachineState *machine)
2 {
3     MachineClass *machine_class = MACHINE_GET_CLASS(machine);
4     numa_complete_configuration(machine);
5     if (nb_numa_nodes) {
6         machine_numa_finish_cpu_init(machine);
7     }
8     .....
9     machine_class->init(machine);
```



```
10 }
```

在 `pc_init1` 里面，我们重点关注两件重要的事情，一个的 CPU 的虚拟化，主要调用 `pc_cpus_init`；另外就是内存的虚拟化，主要调用 `pc_memory_init`。这一节我们重点关注 CPU 的虚拟化，下一节，我们来看内存的虚拟化。


 复制代码

```
1 void pc_cpus_init(PCMachineState *pcms)
2 {
3     .....
4     for (i = 0; i < smp_cpus; i++) {
5         pc_new_cpu(possible_cpus->cpus[i].type, possible_cpus->cpus[i].arch_id, &error_
6     }
7 }
8
9 static void pc_new_cpu(const char *typename, int64_t apic_id, Error **errp)
10 {
11     Object *cpu = NULL;
12     cpu = object_new(typename);
13     object_property_set_uint(cpu, apic_id, "apic-id", &local_err);
14     object_property_set_bool(cpu, true, "realized", &local_err); // 调用 object_property_
15     .....
16 }
```

在 `pc_cpus_init` 中，对于每一个 CPU，都调用 `pc_new_cpu`，在这里，我们又看到了 `object_new`，这又是一个从 `TypeImpl` 到 `Class` 类再到对象的一个过程。

这个时候，我们就要看 CPU 的类是怎么组织的了。

在上面的参数里面，CPU 的配置是这样的：

 复制代码

```
1 -cpu SandyBridge,+erms,+smep,+fsgsbase,+pdpe1gb,+rdrand,+f16c,+osxsave,+dca,+pcid,+pdc
```

在这里我们知道，`SandyBridge` 是 CPU 的一种类型。在 `hw/i386/pc.c` 中，我们能看到这种 CPU 的定义。



```
1 { "SandyBridge" "-" TYPE_X86_CPU, "min-xlevel", "0x8000000a" }
```


接下来，我们就来看"SandyBridge"，也即 TYPE\_X86\_CPU 这种 CPU 的类，是一个什么样的结构。

```
1 static const TypeInfo device_type_info = {
2     .name = TYPE_DEVICE,
3     .parent = TYPE_OBJECT,
4     .instance_size = sizeof(DeviceState),
5     .instance_init = device_initfn,
6     .instance_post_init = device_post_init,
7     .instance_finalize = device_finalize,
8     .class_base_init = device_class_base_init,
9     .class_init = device_class_init,
10    .abstract = true,
11    .class_size = sizeof(DeviceClass),
12 };
13
14 static const TypeInfo cpu_type_info = {
15     .name = TYPE_CPU,
16     .parent = TYPE_DEVICE,
17     .instance_size = sizeof(CPUState),
18     .instance_init = cpu_common_initfn,
19     .instance_finalize = cpu_common_finalize,
20     .abstract = true,
21     .class_size = sizeof(CPUClass),
22     .class_init = cpu_class_init,
23 };
24
25 static const TypeInfo x86_cpu_type_info = {
26     .name = TYPE_X86_CPU,
27     .parent = TYPE_CPU,
28     .instance_size = sizeof(X86CPU),
29     .instance_init = x86_cpu_initfn,
30     .abstract = true,
31     .class_size = sizeof(X86CPUClass),
32     .class_init = x86_cpu_common_class_init,
33 };
```

CPU 这种类的定义是有多层继承关系的。TYPE\_X86\_CPU 的父类是 TYPE\_CPU，TYPE\_CPU 的父类是 TYPE\_DEVICE，TYPE\_DEVICE 的父类是 TYPE\_OBJECT。到头了。


这里面每一层都有 class\_init，用于从 TypeImpl 生产 xxxClass，也有 instance\_init 将 xxxClass 初始化为实例。

在 TYPE\_X86\_CPU 这一层的 class\_init 中，也即 x86\_cpu\_common\_class\_init 中，设置了 DeviceClass 的 realize 函数为 x86\_cpu\_realizefn。这个函数很重要，马上就能用到。

 复制代码

```
1 static void x86_cpu_common_class_init(ObjectClass *oc, void *data)
2 {
3     X86CPUClass *xcc = X86_CPU_CLASS(oc);
4     CPUClass *cc = CPU_CLASS(oc);
5     DeviceClass *dc = DEVICE_CLASS(oc);
6
7     device_class_set_parent_realize(dc, x86_cpu_realizefn,
8                                     &xcc->parent_realize);
9     .....
10 }
```


在 TYPE\_DEVICE 这一层的 instance\_init 函数 device\_initfn，会为此设备添加一个属性"realized"，要设置这个属性，需要用函数 device\_set\_realized。

 复制代码

```
1 static void device_initfn(Object *obj)
2 {
3     DeviceState *dev = DEVICE(obj);
4     ObjectClass *class;
5     Property *prop;
6     dev->realized = false;
7     object_property_add_bool(obj, "realized",
8                               device_get_realized, device_set_realized, NULL);
9     .....
10 }
```

我们回到 pc\_new\_cpu 函数，这里面就是通过 object\_property\_set\_bool 设置这个属性为 true，所以 device\_set\_realized 函数会被调用。

在 `device_set_realized` 中，`DeviceClass` 的 `realize` 函数 `x86_cpu_realizefn` 会被调用。这里面 `qemu_init_vcpu` 会调用 `qemu_kvm_start_vcpu`。

 复制代码

```
1 static void qemu_kvm_start_vcpu(CPUState *cpu)
2 {
3     char thread_name[VCPU_THREAD_NAME_SIZE];
4     cpu->thread = g_malloc0(sizeof(QemuThread));
5     cpu->halt_cond = g_malloc0(sizeof(QemuCond));
6     qemu_cond_init(cpu->halt_cond);
7     qemu_thread_create(cpu->thread, thread_name, qemu_kvm_cpu_thread_fn, cpu, QEMU_THRE
8 }
```

在这里面，为这个 `vcpu` 创建一个线程，也即虚拟机里面的一个 `vcpu` 对应物理机上的一个线程，然后这个线程被调度到某个物理 CPU 上。

我们来看这个 `vcpu` 的线程执行函数。

 复制代码


```
1 static void *qemu_kvm_cpu_thread_fn(void *arg)
2 {
3     CPUState *cpu = arg;
4     int r;
5
6     rcu_register_thread();
7
8     qemu_mutex_lock_iothread();
9     qemu_thread_get_self(cpu->thread);
10    cpu->thread_id = qemu_get_thread_id();
11    cpu->can_do_io = 1;
12    current_cpu = cpu;
13
14    r = kvm_init_vcpu(cpu);
15    kvm_init_cpu_signals(cpu);
16
17    /* signal CPU creation */
18    cpu->created = true;
19    qemu_cond_signal(&qemu_cpu_cond);
20
21    do {
22        if (cpu_can_run(cpu)) {
23            r = kvm_cpu_exec(cpu);
24        }
25        qemu_wait_io_event(cpu);
```

```

26     } while (!cpu->unplug || cpu_can_run(cpu));
27
28     qemu_kvm_destroy_vcpu(cpu);
29     cpu->created = false;
30     qemu_cond_signal(&qemu_cpu_cond);
31     qemu_mutex_unlock_iothread();
32     rcu_unregister_thread();
33     return NULL;
34 }

```

在 `qemu_kvm_cpu_thread_fn` 中，先是 `kvm_init_vcpu` 初始化这个 `vcpu`。

 复制代码

```

1 int kvm_init_vcpu(CPUState *cpu)
2 {
3     KVMState *s = kvm_state;
4     long mmap_size;
5     int ret;
6     .....
7     ret = kvm_get_vcpu(s, kvm_arch_vcpu_id(cpu));
8     .....
9     cpu->kvm_fd = ret;
10    cpu->kvm_state = s;
11    cpu->vcpu_dirty = true;
12
13    mmap_size = kvm_ioctl(s, KVM_GET_VCPU_MMAP_SIZE, 0);
14    .....
15    cpu->kvm_run = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_SHARED, cpu->kvm_fd, 0);
16    .....
17    ret = kvm_arch_init_vcpu(cpu);
18 err:
19    return ret;
20 }

```

在 `kvm_get_vcpu` 中，我们会调用 `kvm_vm_ioctl(s, KVM_CREATE_VCPU, (void *)vcpu_id)`，在内核里面创建一个 `vcpu`。在上面创建 `KVM_CREATE_VM` 的时候，我们已经创建了一个 `struct file`，它的 `file_operations` 被设置为 `kvm_vm_fops`，这个内核文件也是可以响应 `ioctl` 的。

如果我们切换到内核 KVM，在 `kvm_vm_ioctl` 函数中，有对于 `KVM_CREATE_VCPU` 的处理，调用的是 `kvm_vm_ioctl_create_vcpu`。

```

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     int r;
7     switch (ioctl) {
8     case KVM_CREATE_VCPU:
9         r = kvm_vm_ioctl_create_vcpu(kvm, arg);
10        break;
11    case KVM_SET_USER_MEMORY_REGION: {
12        struct kvm_userspace_memory_region kvm_userspace_mem;
13        if (copy_from_user(&kvm_userspace_mem, argp,
14                          sizeof(kvm_userspace_mem)))
15            goto out;
16        r = kvm_vm_ioctl_set_memory_region(kvm, &kvm_userspace_mem);
17        break;
18    }
19    .....
20    case KVM_CREATE_DEVICE: {
21        struct kvm_create_device cd;
22        if (copy_from_user(&cd, argp, sizeof(cd)))
23            goto out;
24        r = kvm_ioctl_create_device(kvm, &cd);
25        if (copy_to_user(argp, &cd, sizeof(cd)))
26            goto out;
27        break;
28    }
29    case KVM_CHECK_EXTENSION:
30        r = kvm_vm_ioctl_check_extension_generic(kvm, arg);
31        break;
32    default:
33        r = kvm_arch_vm_ioctl(filp, ioctl, arg);
34    }
35 out:
36    return r;
37 }

```

在 `kvm_vm_ioctl_create_vcpu` 中，`kvm_arch_vcpu_create` 调用 `kvm_x86_ops` 的 `vcpu_create` 函数来创建 CPU。

```

1 static int kvm_vm_ioctl_create_vcpu(struct kvm *kvm, u32 id)
2 {
3     int r;

```


```

4     struct kvm_vcpu *vcpu;
5     kvm->created_vcpus++;
6     .....
7     vcpu = kvm_arch_vcpu_create(kvm, id);
8     preempt_notifier_init(&vcpu->preempt_notifier, &kvm_preempt_ops);
9     r = kvm_arch_vcpu_setup(vcpu);
10    .....
11    /* Now it's all set up, let userspace reach it */
12    kvm_get_kvm(kvm);
13    r = create_vcpu_fd(vcpu);
14    kvm->vcpus[atomic_read(&kvm->online_vcpus)] = vcpu;
15    .....
16 }
17
18 struct kvm_vcpu *kvm_arch_vcpu_create(struct kvm *kvm,
19                                     unsigned int id)
20 {
21     struct kvm_vcpu *vcpu;
22     vcpu = kvm_x86_ops->vcpu_create(kvm, id);
23     return vcpu;
24 }
25
26 static int create_vcpu_fd(struct kvm_vcpu *vcpu)
27 {
28     return anon_inode_getfd("kvm-vcpu", &kvm_vcpu_fops, vcpu, O_RDWR | O_CLOEXEC);
29 }

```

然后，`create_vcpu_fd` 又创建了一个 `struct file`，它的 `file_operations` 指向 `kvm_vcpu_fops`。从这里可以看出，KVM 的内核模块是一个文件，可以通过 `ioctl` 进行操作。基于这个内核模块创建的 VM 也是一个文件，也可以通过 `ioctl` 进行操作。在这个 VM 上创建的 `vcpu` 同样是一个文件，同样可以通过 `ioctl` 进行操作。

我们回过头来看，`kvm_x86_ops` 的 `vcpu_create` 函数。`kvm_x86_ops` 对于不同的硬件加速虚拟化指向不同的结构，如果是 `vmx`，则指向 `vmx_x86_ops`；如果是 `svm`，则指向 `svm_x86_ops`。我们这里看 `vmx_x86_ops`。这个结构很长，里面有非常多的操作，我们用一个看一个。

 复制代码

```

1 static struct kvm_x86_ops vmx_x86_ops __ro_after_init = {
2     .....
3     .vcpu_create = vmx_create_vcpu,
4     .....
5 }
6

```

```

7 static struct kvm_vcpu *vmx_create_vcpu(struct kvm *kvm, unsigned int id)
8 {
9     int err;
10    struct vcpu_vmx *vmx = kmem_cache_zalloc(kvm_vcpu_cache, GFP_KERNEL);
11    int cpu;
12    vmx->vpid = allocate_vpid();
13    err = kvm_vcpu_init(&vmx->vcpu, kvm, id);
14    vmx->guest_msrs = kmalloc(PAGE_SIZE, GFP_KERNEL);
15    vmx->loaded_vmcs = &vmx->vmcs01;
16    vmx->loaded_vmcs->vmcs = alloc_vmcs();
17    vmx->loaded_vmcs->shadow_vmcs = NULL;
18    loaded_vmcs_init(vmx->loaded_vmcs);
19
20    cpu = get_cpu();
21    vmx_vcpu_load(&vmx->vcpu, cpu);
22    vmx->vcpu.cpu = cpu;
23    err = vmx_vcpu_setup(vmx);
24    vmx_vcpu_put(&vmx->vcpu);
25    put_cpu();
26
27    if (enable_ept) {
28        if (!kvm->arch.ept_identity_map_addr)
29            kvm->arch.ept_identity_map_addr =
30                VMX_EPT_IDENTITY_PAGETABLE_ADDR;
31        err = init_rmode_identity_map(kvm);
32    }
33
34    return &vmx->vcpu;
35 }

```

`vmx_create_vcpu` 创建用于表示 vcpu 的结构 `struct vcpu_vmx`，并填写里面的内容。例如 `guest_msrs`，咱们在讲系统调用的时候提过 `msr` 寄存器，虚拟机也需要有这样的寄存器。

`enable_ept` 是和内存虚拟化相关的，EPT 全称 Extended Page Table，顾名思义，是优化内存虚拟化的，这个功能我们放到内存的那一节讲。

最最重要的就是 `loaded_vmcs` 了。VMCS 是什么呢？它的全称是 Virtual Machine Control Structure。它是来干什么呢？

前面咱们将进程调度的时候讲过，为了支持进程在 CPU 上的切换，CPU 硬件要求有一个 TSS 结构，用于保存进程运行时的所有寄存器的状态，进程切换的时候，需要根据 TSS 恢复寄存器。



虚拟机也是一个进程，也需要切换，而且切换更加的复杂，可能是两个虚拟机之间切换，也可能是虚拟机切换给内核，虚拟机因为里面还有另一个操作系统，要保存的信息比普通的进程多得多。那就需要有一个结构来保存虚拟机运行的上下文，VMCS 就是 Intel 实现 CPU 虚拟化，记录 vCPU 状态的一个关键数据结构。

VMCS 数据结构主要包含以下信息。

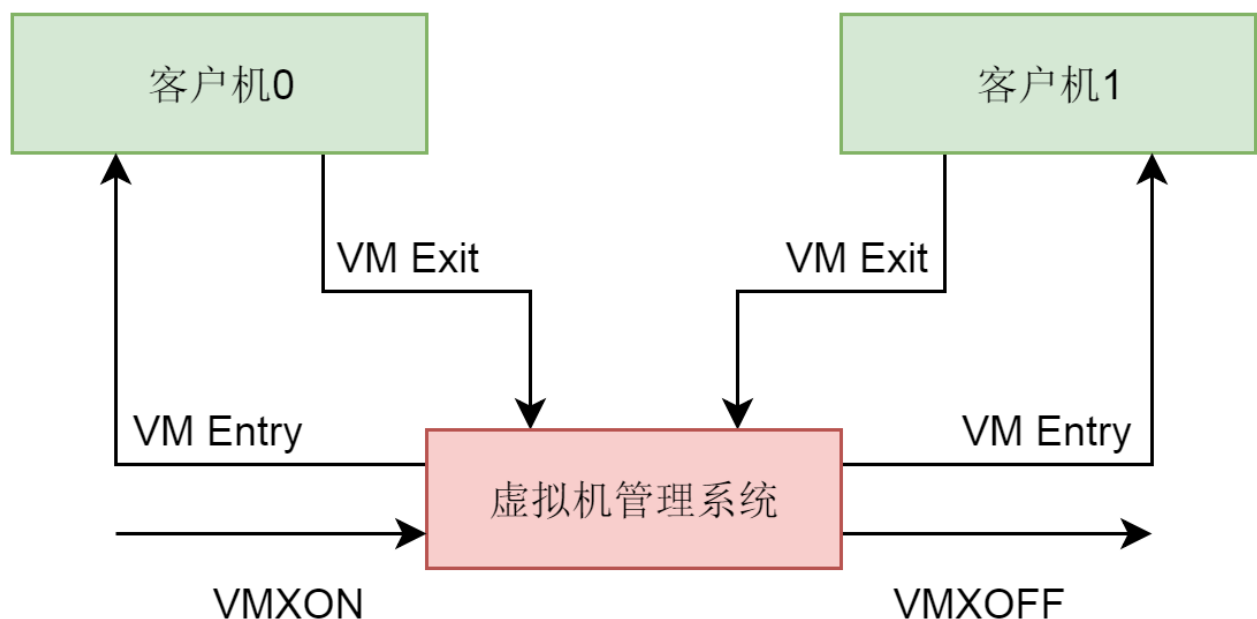
Guest-state area，即 vCPU 的状态信息，包括 vCPU 的基本运行环境，例如寄存器等。

Host-state area，是物理 CPU 的状态信息。物理 CPU 和 vCPU 之间也会来回切换，所以，VMCS 中既要记录 vCPU 的状态，也要记录物理 CPU 的状态。

VM-execution control fields，对 vCPU 的运行行为进行控制。例如，发生中断怎么办，是否使用 EPT ( Extended Page Table ) 功能等。

接下来，对于 VMCS，有两个重要的操作。

VM-Entry，我们称为从根模式切换到非根模式，也即切换到 guest 上，这个时候 CPU 上运行的是虚拟机。VM-Exit 我们称为 CPU 从非根模式切换到根模式，也即从 guest 切换到宿主机。例如，当要执行一些虚拟机没有权限的敏感指令时。



为了维护这两个动作，VMCS 里面还有几项内容：

VM-exit control fields , 对 VM Exit 的行为进行控制。比如 , VM Exit 的时候对 vCPU 来说需要保存哪些 MSR 寄存器 , 对于主机 CPU 来说需要恢复哪些 MSR 寄存器。


VM-entry control fields , 对 VM Entry 的行为进行控制。比如 , 需要保存和恢复哪些 MSR 寄存器等。

VM-exit information fields , 记录下发生 VM Exit 发生的原因及一些必要的信息 , 方便对 VM Exit 事件进行处理。

至此 , 内核准备完毕。

我们再回到 qemu 的 kvm\_init\_vcpu 函数 , 这里面除了创建内核中的 vcpu 结构之外 , 还通过 mmap 将内核的 vcpu 结构 , 映射到 qemu 中 CPUState 的 kvm\_run 中 , 为什么能用 mmap 呢 , 上面咱们不是说过了吗 , vcpu 也是一个文件。

我们再回到这个 vcpu 的线程函数 qemu\_kvm\_cpu\_thread\_fn , 他在执行 kvm\_init\_vcpu 创建 vcpu 之后 , 接下来是一个 do-while 循环 , 也即一直运行 , 并且通过调用 kvm\_cpu\_exec , 运行这个虚拟机。

 复制代码

```
1 int kvm_cpu_exec(CPUState *cpu)
2 {
3     struct kvm_run *run = cpu->kvm_run;
4     int ret, run_ret;
5     .....
6     do {
7         .....
8         run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
9         .....
10        switch (run->exit_reason) {
11        case KVM_EXIT_IO:
12            kvm_handle_io(run->io.port, attrs,
13                          (uint8_t *)run + run->io.data_offset,
14                          run->io.direction,
15                          run->io.size,
16                          run->io.count);
17            break;
18        case KVM_EXIT_IRQ_WINDOW_OPEN:
19            ret = EXCP_INTERRUPT;
20            break;
21        case KVM_EXIT_SHUTDOWN:
22            qemu_system_reset_request(SHUTDOWN_CAUSE_GUEST_RESET);
23            ret = EXCP_INTERRUPT;
24            break;
```

```

25         case KVM_EXIT_UNKNOWN:
26             fprintf(stderr, "KVM: unknown exit, hardware reason %" PRIx64 "\n", (uint64_t)exit_reason);
27             ret = -1;
28             break;
29         case KVM_EXIT_INTERNAL_ERROR:
30             ret = kvm_handle_internal_error(cpu, run);
31             break;
32         .....
33     }
34 } while (ret == 0);
35 .....
36 return ret;
37 }


```

在 `kvm_cpu_exec` 中，我们能看到一个循环，在循环中，`kvm_vcpu_ioctl(KVM_RUN)` 运行这个虚拟机，这个时候 CPU 进入 VM-Entry，也即进入客户机模式。

如果一直是客户机的操作系统占用这个 CPU，则会一直停留在这一行运行，一旦这个调用返回了，就说明 CPU 进入 VM-Exit 退出客户机模式，将 CPU 交还给宿主机。在循环中，我们会对退出的原因 `exit_reason` 进行分析处理，因为有了 I/O，还有了中断等，做相应的处理。处理完毕之后，再次循环，再次通过 VM-Entry，进入客户机模式。如此循环，直到虚拟机正常或者异常退出。

我们来看 `kvm_vcpu_ioctl(KVM_RUN)` 在内核做了哪些事情。

上面我们也讲了，`vcpu` 在内核也是一个文件，也是通过 `ioctl` 进行用户态和内核态通信的，在内核中，调用的是 `kvm_vcpu_ioctl`。

 复制代码

```

1 static long kvm_vcpu_ioctl(struct file *filp,
2                             unsigned int ioctl, unsigned long arg)
3 {
4     struct kvm_vcpu *vcpu = filp->private_data;
5     void __user *argp = (void __user *)arg;
6     int r;
7     struct kvm_fpu *fpu = NULL;
8     struct kvm_sregs *kvm_sregs = NULL;
9     .....
10    r = vcpu_load(vcpu);
11    switch (ioctl) {
12    case KVM_RUN: {
13        struct pid *oldpid;


```

```

14         r = kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
15         break;
16     }
17     case KVM_GET_REGS: {
18         struct kvm_regs *kvm_regs;
19         kvm_regs = kzalloc(sizeof(struct kvm_regs), GFP_KERNEL);
20         r = kvm_arch_vcpu_ioctl_get_regs(vcpu, kvm_regs);
21         if (copy_to_user(argp, kvm_regs, sizeof(struct kvm_regs)))
22             goto out_free1;
23         break;
24     }
25     case KVM_SET_REGS: {
26         struct kvm_regs *kvm_regs;
27         kvm_regs = memdup_user(argp, sizeof(*kvm_regs));
28         r = kvm_arch_vcpu_ioctl_set_regs(vcpu, kvm_regs);
29         break;
30     }
31     .....
32 }

```

kvm\_arch\_vcpu\_ioctl\_run 会调用 vcpu\_run , 这里面也是一个无限循环。

 复制代码


```

1 static int vcpu_run(struct kvm_vcpu *vcpu)
2 {
3     int r;
4     struct kvm *kvm = vcpu->kvm;
5
6     for (;;) {
7         if (kvm_vcpu_running(vcpu)) {
8             r = vcpu_enter_guest(vcpu);
9         } else {
10             r = vcpu_block(kvm, vcpu);
11         }
12         ....
13         if (signal_pending(current)) {
14             r = -EINTR;
15             vcpu->run->exit_reason = KVM_EXIT_INTR;
16             ++vcpu->stat.signal_exits;
17             break;
18         }
19         if (need_resched()) {
20             cond_resched();
21         }
22     }
23     .....
24     return r;

```

在这个循环中，除了调用 `vcpu_enter_guest` 进入客户机模式运行之外，还有对于信号的响应 `signal_pending`，也即一台虚拟机是可以被 kill 掉的，还有对于调度的响应，这台虚拟机可以被从当前的物理 CPU 上赶下来，换成别的虚拟机或者其他进程。


我们这里重点看 `vcpu_enter_guest`。

 复制代码

```

1 static int vcpu_enter_guest(struct kvm_vcpu *vcpu)
2 {
3     r = kvm_mmu_reload(vcpu);
4     vcpu->mode = IN_GUEST_MODE;
5     kvm_load_guest_xcr0(vcpu);
6     .....
7     guest_enter_irqoff();
8     kvm_x86_ops->run(vcpu);
9     vcpu->mode = OUTSIDE_GUEST_MODE;
10    .....
11    kvm_put_guest_xcr0(vcpu);
12    kvm_x86_ops->handle_external_intr(vcpu);
13    ++vcpu->stat.exits;
14    guest_exit_irqoff();
15    r = kvm_x86_ops->handle_exit(vcpu);
16    return r;
17    .....
18 }
19
20 static struct kvm_x86_ops vmx_x86_ops __ro_after_init = {
21     .....
22     .run = vmx_vcpu_run,
23     .....
24 }
```

在 `vcpu_enter_guest` 中，我们会调用 `vmx_x86_ops` 的 `vmx_vcpu_run` 函数，进入客户机模式。

 复制代码

```

1 static void __noclone vmx_vcpu_run(struct kvm_vcpu *vcpu)
2 {
3     struct vcpu_vmx *vmx = to_vmx(vcpu);
```

```

4      unsigned long debugctlmsr, cr3, cr4;
5      .....
6      cr3 = __get_current_cr3_fast();
7      .....
8      cr4 = cr4_read_shadow();
9      .....
10     vmx->__launched = vmx->loaded_vmcs->launched;
11     asm(
12         /* Store host registers */
13         "push %%_ASM_DX "; push %%_ASM_BP ";
14         "push %%_ASM_CX " \n\t" /* placeholder for guest rcx */
15         "push %%_ASM_CX " \n\t"
16     .....
17         /* Load guest registers. Don't clobber flags. */
18         "mov %crax, %%_ASM_AX " \n\t"
19         "mov %crbx, %%_ASM_BX " \n\t"
20         "mov %crdx, %%_ASM_DX " \n\t"
21         "mov %crsi, %%_ASM_SI " \n\t"
22         "mov %crdi, %%_ASM_DI " \n\t"
23         "mov %crbp, %%_ASM_BP " \n\t"
24 #ifdef CONFIG_X86_64
25         "mov %cr8, %%r8 \n\t"
26         "mov %cr9, %%r9 \n\t"
27         "mov %cr10, %%r10 \n\t"
28         "mov %cr11, %%r11 \n\t"
29         "mov %cr12, %%r12 \n\t"
30         "mov %cr13, %%r13 \n\t"
31         "mov %cr14, %%r14 \n\t"
32         "mov %cr15, %%r15 \n\t"
33 #endif
34         "mov %crcx, %%_ASM_CX " \n\t" /* kills %0 (ecx) */
35
36         /* Enter guest mode */
37         "jne 1f \n\t"
38         __ex(ASM_VMX_VMLAUNCH) "\n\t"
39         "jmp 2f \n\t"
40         "1: " __ex(ASM_VMX_VMRESUME) "\n\t"
41         "2: "
42         /* Save guest registers, load host registers, keep flags */
43         "mov %0, %cwordsize \n\t"
44         "pop %0 \n\t"
45         "mov %%_ASM_AX ", %crax \n\t"
46         "mov %%_ASM_BX ", %crbx \n\t"
47         __ASM_SIZE(pop) " %crcx \n\t"
48         "mov %%_ASM_DX ", %crdx \n\t"
49         "mov %%_ASM_SI ", %crsi \n\t"
50         "mov %%_ASM_DI ", %crdi \n\t"
51         "mov %%_ASM_BP ", %crbp \n\t"
52 #ifdef CONFIG_X86_64
53         "mov %%r8, %cr8 \n\t"
54         "mov %%r9, %cr9 \n\t"
55         "mov %%r10, %cr10 \n\t"

```

```

56         "mov %%r11, %cr11 \n\t"
57         "mov %%r12, %cr12 \n\t"
58         "mov %%r13, %cr13 \n\t"
59         "mov %%r14, %cr14 \n\t"
60         "mov %%r15, %cr15 \n\t"
61 #endif
62         "mov %%cr2, %%_ASM_AX " \n\t"
63         "mov %%_ASM_AX ", %c cr2 \n\t"
64
65         "pop %%_ASM_BP "; pop %%_ASM_DX " \n\t"
66         "setbe %c fail \n\t"
67         ".pushsection .rodata \n\t"
68         ".global vmx_return \n\t"
69         "vmx_return: " _ASM_PTR " 2b \n\t"
70 .....
71         );
72 .....
73         vmx->loaded_vmcs->launched = 1;
74         vmx->exit_reason = vmcs_read32(VM_EXIT_REASON);
75 .....
76 }

```

在 `vmx_vcpu_run` 中，出现了汇编语言的代码，比较难看懂，但是没有关系呀，里面有注释呀，我们可以沿着注释来看。

首先是 Store host registers，要从宿主机模式变为客户机模式了，所以原来宿主机运行时候的寄存器要保存下来。

接下来是 Load guest registers，将原来客户机运行的时候的寄存器加载进来。

接下来是 Enter guest mode，调用 `ASM_VMX_VMLAUNCH` 进入客户机模型运行，或者 `ASM_VMX_VMRESUME` 恢复客户机模型运行。

如果客户机因为某种原因退出，Save guest registers, load host registers，也即保存客户机运行的时候的寄存器，就加载宿主机运行的时候的寄存器。

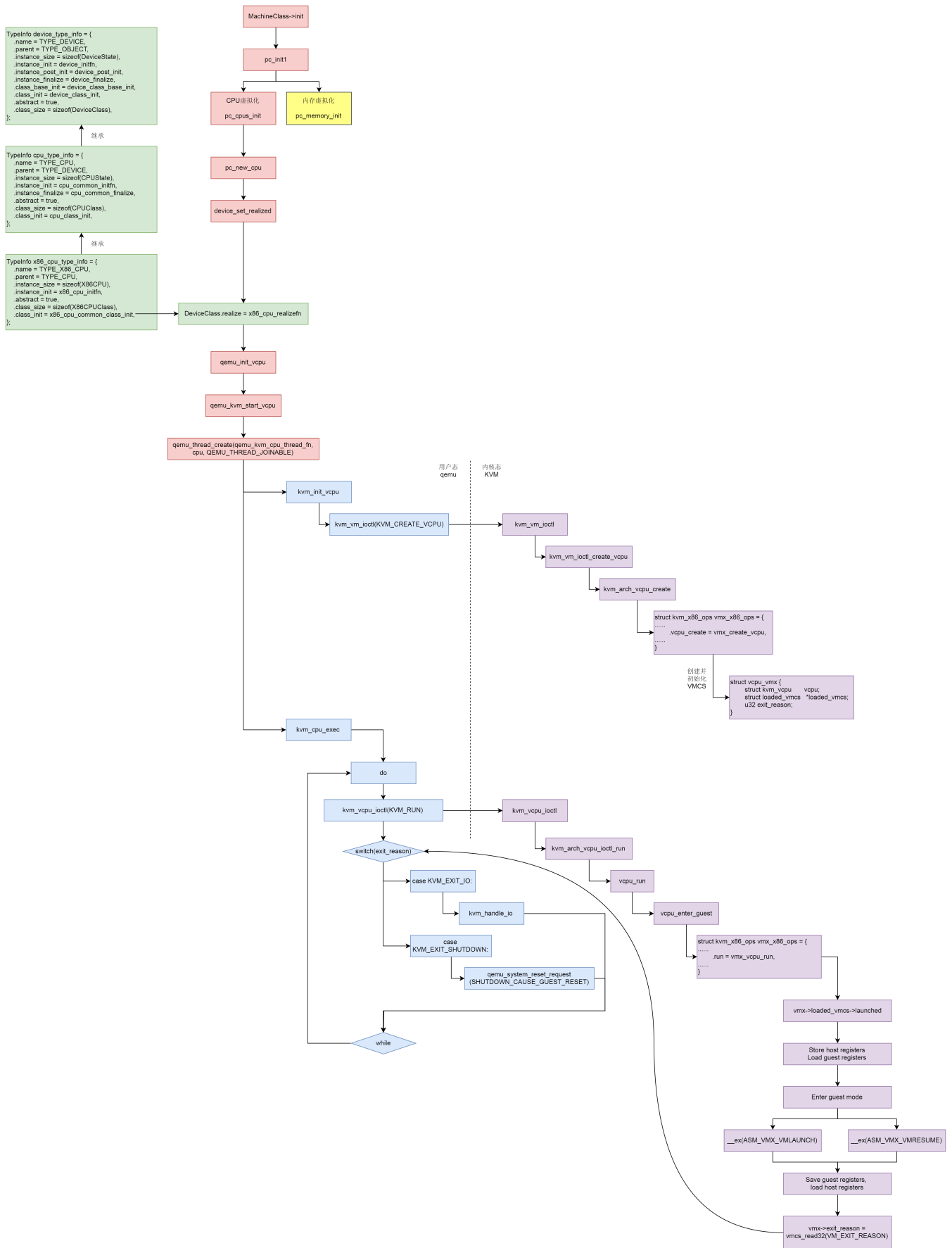
最后将 `exit_reason` 保存在 `vmx` 结构中。

至此，CPU 虚拟化就解析完了。

## 总结时刻

CPU 的虚拟化过程还是很复杂的，我画了一张图总结了一下。





首先，我们要定义 CPU 这种类型的 TypeInfo 和 TypeImpl、继承关系，并且声明它的类初始化函数。

在 qemu 的 main 函数中调用 MachineClass 的 init 函数，这个函数既会初始化 CPU，也会初始化内存。

CPU 初始化的时候，会调用 pc\_new\_cpu 创建一个虚拟 CPU，它会调用 CPU 这个类的初始化函数。

每一个虚拟 CPU 会调用 qemu\_thread\_create 创建一个线程，线程的执行函数为 qemu\_kvm\_cpu\_thread\_fn。

在虚拟 CPU 对应的线程执行函数中，我们先是调用 kvm\_vm\_ioctl(KVM\_CREATE\_VCPU)，在内核的 KVM 里面，创建一个结构 struct vcpu\_vmx，表示这个虚拟 CPU。在这个结构里面，有一个 VMCS，用于保存当前虚拟机 CPU 的运行时的状态，用于状态切换。

在虚拟 CPU 对应的线程执行函数中，我们接着调用 kvm\_vcpu\_ioctl(KVM\_RUN)，在内核的 KVM 里面运行这个虚拟机 CPU。运行的方式是保存宿主机的寄存器，加载客户机的寄存器，然后调用 \_\_ex(ASM\_VMX\_VMLAUNCH) 或者 \_\_ex(ASM\_VMX\_VMRESUME)，进入客户机模式运行。一旦退出客户机模式，就会保存客户机寄存器，加载宿主机寄存器，进入宿主机模式运行，并且会记录退出虚拟机模式的原因。大部分的原因是等待 I/O，因而宿主机调用 kvm\_handle\_io 进行处理。

## 课堂练习

在咱们上面操作 KVM 的过程中，出现了好几次文件系统。不愧是“Linux 中一切皆文件”。那你能否整理一下这些文件系统之间的关系呢？

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 50 | 计算虚拟化之CPU（上）：如何复用集团的人力资源？

下一篇 52 | 计算虚拟化之内存：如何建立独立的办公室？

## 精选留言 (3)

 写留言



安排

2019-07-28

老师，那在虚拟机里面创建的多个核其实是假的是码？即使创建4个核的虚拟机，那么对应到kvm里面其实也是一个线程，也就是从虚拟机os这个层面它是无法真正利用多核的。其实它虚拟机os利用多核也没有意义。只要保证宿主os能正常利用多核就足够了，不知道这样理解是否正确？

展开 ∨



一笔一画

2019-07-27

.unlocked\_ioctl = kvm\_dev\_ioctl,

`.compat_ioctl = kvm_dev_ioctl,`  
请问下这两个ioctl有什么区别？在什么时候会调到  
展开 ▾



**小龙的城堡**

2019-07-24

深入内核以后，发现一切都是那么简洁，美妙？

展开 ▾

