KVM API

总览

KVM提供的API可以分为三类

API 类型	功能说明
system 指令	针对虚拟化系统的全局性参数设置和用于虚拟机创建等控制操作
VM 指令	针对具体的 VM 虚拟机进行控制，如进行内存设置、创建 vCPU 等。注意：VM 指令不是进程安全的
vcpu 指令	针对具体的 vCPU 进行参数设置（MRU 寄存器读写、中断控制等）

- KVM的 API是通过`/dev/kvm`接口设备进行访问的。`/dev/kvm`是一个**字符型设备**。所有对KVM的操作都是通过`ioctl`发送相应的**控制字**实现的。

1. 系统调用open()之后会获得针对对kvm子系统的一个fd文件描述符

```
//路径：qemu/accel/kvm/kvm-all.c
static int kvm_init(MachineState *ms){
    KVMState *s;
    s->fd = qemu_open_old("/dev/kvm", O_RDWR);    //fd是Int
    ret = kvm_ioctl(s, KVM_GET_API_VERSION, 0);    //查询KVM API版本，如果版本较低会打印eror

    do {
        ret = kvm_ioctl(s, KVM_CREATE_VM, type);  //创建虚拟机，获取到虚拟机的句柄
    } while (ret == -EINTR);

    /* check the vcpu limits */
    soft_vcpus_limit = kvm_recommended_vcpus(s);    //内核推荐调度策略下支持的cpu数量
    hard_vcpus_limit = kvm_max_vcpus(s);
}
```

2. 通过ioctl系统指令对该文件描述符进行进一步的操作

3. 通过KVM_CREATE_VM指令将创建一个虚拟机并返回该虚拟机对应的fd文件描述符，然后根据该描述符来控制虚拟机的行为
4. 通过KVM_CREATE_VCPU指令，将创建一个虚拟CPU并且返回该vCPU对应的fd

system ioctls调用

system ioctls系统调用用于控制KVM运行环节的参数，相关工作包括全局性的参数设置和虚拟机创建等工作，其主要指令字如表所示

指令字	功能说明
KVM_GET_API_VERSION	查询当前 KVM API 的版本
KVM_CREATE_VM	创建 KVM 虚拟机
KVM_GET_MSR_INDEX_LIST	获得 MSR 索引列表
KVM_CHECK_EXTENSION	检查扩展支持情况
KVM_GET_VCPU_MMAP_SIZE	运行虚拟机和用户态空间共享的一片内存区域的大小

KVM_CREATE_VM 是其中比较重要的指令字。通过该参数，KVM 将返回一个文件描述符，该文件描述符指向内核控件中一个新的虚拟机。

全新创建的虚拟机没有vCPU，也没有内存，需要通过后续的ioctl指令进行配置。使用mmap()系统调用，则会直接返回该虚拟机对应的虚拟内存空间，并且内存的偏移量为 0。

vm ioctl系统调用

vm ioctl系统调用实现了对虚拟机的控制。vm ioctl控制指令的参数大多需要从 KVM_CREATE_VM 中返回的 fd 文件描述符来进行操作，涉及的操作主要针对某一个虚拟机进行控制，如配置内存、配置vCPU等。

指令字	功能说明
KVM_CREATE_VCPU	为已经创建好的 VM 添加 vCPU
KVM_GET_DIRTY_LOG	返回经过上次操作之后，改变的内存页的位图
KVM_CREATE_IRQCHIP	创建一个虚拟的 APIC，并且随后创建的 vCPU 都将连接到该 APIC
KVM_IRQ_LINE	对某虚拟的 APIC 引发中断信号
KVM_GET_IRQCHIP	读取 APIC 的中断标志信息
KVM_SET_IRQCHIP	写入 APIC 的中断标志信息
KVM_RUN	根据 kvm_run 结构体的信息，启动 VM 虚拟机

KVM_RUN和KVM_CREATE_VCPU是vm ioctl系统调用的两个重要指令字。在通过KVM_CREATE_VCPU为 VM虚拟机创建 vCPU，并且获得 vCPU对应的fd文件描述符！！之后，可以进行KVM_RUN启动虚拟机！！！ 的操作

KVM_RUN 指令字虽然没有任何参数，但是在调用 KVM_RUN 启动了**虚拟机**之后，可以通过 mmap()系统调用映射 vCPU 的 fd 所在的内存空间来获得kvm_run结构体信息。

kvm_run结构体的定义在`include/uapi/linux/kvm.h`中，通过读取该结构体可以了解 KVM 内部的运行状态。

字段名	功能说明
request_interrupt_window	往 vCPU 中发出一个中断插入请求，让 vCPU 做好相关的准备工作
ready_for_interrupt_injection	响应 request_interrupt_windows 中的中断请求，当此位有效时，说明可以进行中断
if_flag	中断标识，如果使用了 APIC，则不起作用
hardware_exit_reason	当 vCPU 因为各种不明原因退出时，该字段保存了失败的描述信息(硬件失效)
io	该字段为一个结构体。当 KVM 产生硬件出错的原因是因为 I/O 输出时(KVM_EXIT_IO)，该结构体将保存导致出错的 I/O 请求的原因
mmio	该字段为一个结构体。当 KVM 产生出错的原因是因为内存 I/O 映射导致的（KVM_EXIT_MMIO），该结构体中将保存导致出错的内存 I/O 映射请求的数据

vcpu ioctl系统调用

vcpu ioctl系统调用主要针对具体的**每一个虚拟的vCPU**进行配置，包括**寄存器读/写**、**中断设置**、**内存设置**、**调试开关**、**时钟管理**等功能，能够对 KVM的虚拟机进行精确的运行时配置。

KVM启动过程

虚拟机启动过程

1. 获取到KVM fd句柄 `kvmfd = open("/dev/kvm", O_RDWR);`
2. 创建虚拟机，获取到虚拟机句柄 `ret = kvm_ioctl(s, KVM_CREATE_VM, type);`
3. 为虚拟机映射内存，还有其他的PCI，信号处理的初始化。刚创建的VM是没有cpu和内存的，需要QEMU进程利用mmap系统调用映射一块内存给VM的描述符，其实也就是给VM创建内存的过程。

```
static int kvm_set_user_memory_region(KVMMemoryListener *kml, KVMSlot
*slot, bool new){
    KVMState *s = kvm_state;
    struct kvm_userspace_memory_region mem;
    int ret;
    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 && !new && (mem.flags ^ slot->old_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;
        ret = kvm_vm_ioctl(s, KVM_SET_USER_MEMORY_REGION, &mem);
        if (ret < 0) {
            goto err;
        }
    }
```

```
    mem.memory_size = slot->memory_size;
    ret = kvm_vm_ioctl(s, KVM_SET_USER_MEMORY_REGION, &mem);
    slot->old_flags = mem.flags;
err:
    trace_kvm_set_user_memory(mem.slot, mem.flags,
mem.guest_phys_addr,
                              mem.memory_size, mem.userspace_addr,
ret);
    if (ret < 0) {
        error_report("%s: KVM_SET_USER_MEMORY_REGION failed, slot=%d,"
                     " start=0x%" PRIx64 ", size=0x%" PRIx64 ": %s",
                     __func__, mem.slot, slot->start_addr,
                     (uint64_t)mem.memory_size, strerror(errno));
    }
    return ret;
}
```

4. 将虚拟机镜像映射到内存，相当于物理机的boot过程，把镜像映射到内存。
5. 创建vCPU，并为vCPU分配内存空间

```
int kvm_init_vcpu(CPUState *cpu, Error **errp){
    KVMState *s = kvm_state;
    long mmap_size;
    int ret;

    trace_kvm_init_vcpu(cpu->cpu_index, kvm_arch_vcpu_id(cpu));
    ret = kvm_get_vcpu(s, kvm_arch_vcpu_id(cpu));   //这个函数中会创建
vcpu

    mmap_size = kvm_ioctl(s, KVM_GET_VCPU_MMAP_SIZE, 0);    //为vcpu分
配内存空间
}
```

```
static int kvm_get_vcpu(KVMState *s, unsigned long vcpu_id)
{
    struct KVMParkedVcpu *cpu;

    QLIST_FOREACH(cpu, &s->kvm_parked_vcpus, node) {
        if (cpu->vcpu_id == vcpu_id) {
            int kvm_fd;

            QLIST_REMOVE(cpu, node);
            kvm_fd = cpu->kvm_fd;
            g_free(cpu);
            return kvm_fd;
```

```
        }
    }

    return kvm_vm_ioctl(s, KVM_CREATE_VCPU, (void *)vcpu_id);
}
```

6. 创建vCPU个数的线程并运行虚拟机。

```
void qemu_init_vcpu(CPUState *cpu)
{
    MachineState *ms = MACHINE(qdev_get_machine());

    cpu->nr_cores = ms->smp.cores;
    cpu->nr_threads =  ms->smp.threads;
    cpu->stopped = true;
    cpu->random_seed = qemu_guest_random_seed_thread_part1();

    if (!cpu->as) {
        /* If the target cpu hasn't set up any address spaces itself,
         * give it the default one.
         */
        cpu->num_ases = 1;
        cpu_address_space_init(cpu, 0, "cpu-memory", cpu->memory);
    }

    /* accelerators all implement the CpusAccel interface */
    g_assert(cpus_accel != NULL && cpus_accel->create_vcpu_thread !=
NULL);
    cpus_accel->create_vcpu_thread(cpu);

    while (!cpu->created) {
        qemu_cond_wait(&qemu_cpu_cond, &qemu_global_mutex);
    }
}
```

7. 第六步，线程进入循环，并捕获虚拟机退出原因，做相应的处理。在虚机启动之后正常运行的过程中，虚机就是在不断的经历ioctl进入，返回，进入，返回的循环过程。

总结

虚拟机的启动过程：

创建kvm句柄 -> 创建vm -> 分配内存 -> 加载镜像到内存 -> 启动线程执行KVM_RUN。

虚拟机的内存是由宿主机通过mmap调用映射给虚拟机的，而vCPU是宿主机的一个线程，这个线程通过设置相应的vCPU的寄存器指定了虚拟机的程序加载地址后，开始运行虚拟机的指令，当虚拟机执行了IO操作后，CPU捕获到中断并把执行权又交回给宿主机。