QEMU/KVM 基本实现

以 RISC-V 为例

李宇

2023 年 6 月 16 日

1 概述
- KVM 是什么
- QEMU 是什么
- QEMU 与 KVM 的交互

2 KVM 代码走读
- KVM 初始化
- 虚拟机创建
- vCPU 运行

3 QEMU 代码摘要
- Machine 初始化
- QEMU 的面向对象实现

KVM 是什么

KVM: Kernel-based Virtual Machine

KVM 最初版本的描述为:

> *KVM (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko.*

简单来说，KVM 是一个 Linux 中的虚拟化解决方案。
KVM 以内核模块的形式存在，基于硬件的虚拟化扩展实现，并为用户态程序提供基本的接口。

QEMU 是什么

QEMU: Quick Emulator。其官方文档中的描述为:

“ QEMU is a generic and open source machine & userspace emulator and virtualizer. ”

QEMU 可以模拟裸机，也可以模拟用户态程序。其支持多种不同的架构，也提供多种不同的加速方式。

QEMU 支持的 target-list

```
Usage: configure [options]
Options: [defaults in brackets after descriptions]

Standard options:
  --help                  print this message
  --prefix=PREFIX         install in PREFIX [/usr/local]
  --target-list=LIST      set target list (default: build all)
        Available targets: aarch64-softmmu alpha-softmmu
        arm-softmmu avr-softmmu cris-softmmu hppa-softmmu
        i386-softmmu loongarch64-softmmu m68k-softmmu
            ......
        aarch64-linux-user aarch64_be-linux-user
        alpha-linux-user arm-linux-user armeb-linux-user
        cris-linux-user hexagon-linux-user hppa-linux-user
        i386-linux-user loongarch64-linux-user
            ......
```

QEMU 支持的加速方式

```
root@root:~/qemu# qemu-system-riscv64 -help
QEMU emulator version 7.2.0 (v7.2.0)
Copyright (c) 2003-2022 Fabrice Bellard and the QEMU Project developers
usage: qemu-system-riscv64 [options] [disk_image]

Standard options:
-h or -help     display this help and exit
-version        display version information and exit
        ......
-accel [accel=]accelerator[,prop[=value][,...]]
    select accelerator
        (kvm, xen, hax, hvf, nvmm, whpx or tcg; use 'help' for a list)
    igd-passthru=on|off
        (enable Xen integrated Intel graphics passthrough, default=off)
    kernel-irqchip=on|off|split controls accelerated irqchip support
        (default=on)
    kvm-shadow-mem=size of KVM shadow MMU in bytes
    split-wx=on|off (enable TCG split w^x mapping)
    tb-size=n (TCG translation block cache size)
    dirty-ring-size=n (KVM dirty ring GFN count, default 0)
    notify-vmexit=run|internal-error|disable,notify-window=n
        (enable notify VM exit and set notify window, x86 only)
    thread=single|multi (enable multi-threaded TCG)
```

QEMU 与 KVM 的交互方式

KVM 模块被加载后，会出现一个字符设备 `/dev/kvm`，QEMU 就是通过这个字符设备与 KVM 交互的。

与 KVM 的通信主要由 `SYS_ioctl` 这一系统调用实现，比如

- 通过 `/dev/kvm` 的 `fd` 获取 KVM 的 API 版本

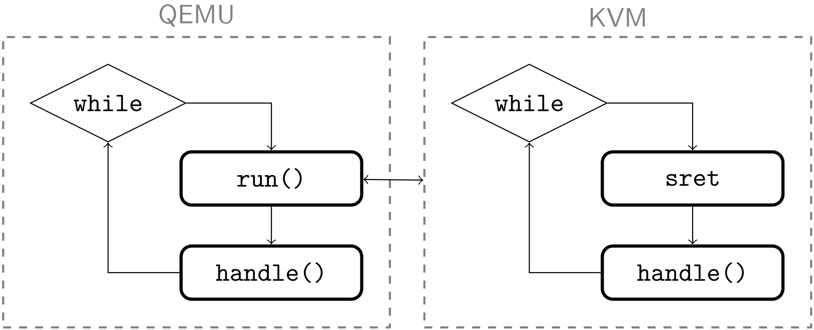
  `ret = ioctl(kvmfd, KVM_GET_API_VERSION, 0);`
- 通过 `/dev/kvm` 的 `fd` 创建一个 VM 并得到其 `fd`

  `ret = ioctl(kvmfd, KVM_CREATE_VM, vm_type);`
- 通过 VM 的 `fd` 创建一个 vCPU 并得到其 `fd`

  `ret = ioctl(vmfd, KVM_CREATE_VCPU, vcpu_id);`

QEMU 与 KVM 的运行过程

为了让 vCPU 持续运行，QEMU 和 KVM 都需要有一个死循环来让 vCPU 不停地进入 Guest 中，仅当发生 VM exit 时才会退出。



1 概述
- KVM 是什么
- QEMU 是什么
- QEMU 与 KVM 的交互

2 KVM 代码走读
- KVM 初始化
- 虚拟机创建
- vCPU 运行

3 QEMU 代码摘要
- Machine 初始化
- QEMU 的面向对象实现

1 概述
- KVM 是什么
- QEMU 是什么
- QEMU 与 KVM 的交互

2 KVM 代码走读
- KVM 初始化
- 虚拟机创建
- vCPU 运行

3 QEMU 代码摘要
- Machine 初始化
- QEMU 的面向对象实现

KVM 初始化[1]

```
static int __init riscv_kvm_init(void)
{
    if (!riscv_isa_extension_available(NULL, h)) {
        kvm_info("hypervisor extension not available\n");
        return -ENODEV;
    }

    kvm_riscv_gstage_mode_detect();

    kvm_riscv_gstage_vmid_detect();

    return kvm_init(sizeof(struct kvm_vcpu), 0, THIS_MODULE);
}
module_init(riscv_kvm_init);
```

[1] 由于篇幅限制，本节会省略大量代码，仅保留核心逻辑

探测分页模式

```
void __init kvm_riscv_gstage_mode_detect(void)
{
#ifdef CONFIG_64BIT
    /* Try Sv57x4 G-stage mode */
    csr_write(CSR_HGATP, HGATP_MODE_SV57X4 << HGATP_MODE_SHIFT);
    if ((csr_read(CSR_HGATP) >> HGATP_MODE_SHIFT) == HGATP_MODE_SV57X4) {
        gstage_mode = (HGATP_MODE_SV57X4 << HGATP_MODE_SHIFT);
        gstage_pgd_levels = 5;
        goto skip_sv48x4_test;
    }

    /* Try Sv48x4 G-stage mode */
    csr_write(CSR_HGATP, HGATP_MODE_SV48X4 << HGATP_MODE_SHIFT);
    if ((csr_read(CSR_HGATP) >> HGATP_MODE_SHIFT) == HGATP_MODE_SV48X4) {
        gstage_mode = (HGATP_MODE_SV48X4 << HGATP_MODE_SHIFT);
        gstage_pgd_levels = 4;
    }

skip_sv48x4_test:
    csr_write(CSR_HGATP, 0);
    kvm_riscv_local_hfence_gvma_all();
#endif
}
```

KVM ioctl

```
static long kvm_dev_ioctl(struct file *filp,
                          unsigned int ioctl,
                          unsigned long arg)
{
    int r = -EINVAL;

    switch (ioctl) {
    /* ...... */
    case KVM_CREATE_VM:
        r = kvm_dev_ioctl_create_vm(arg);
        break;
    /* ...... */
    default:
        return kvm_arch_dev_ioctl(filp, ioctl, arg);
    }
out:
    return r;
}
```

创建 VM

```
static struct kvm *kvm_create_vm(unsigned long type, const char *fdname)
{
    struct kvm *kvm = kvm_arch_alloc_vm();

    if (!kvm)
        return ERR_PTR(-ENOMEM);

    r = kvm_arch_init_vm(kvm, type);
    if (r)
        goto out_err_no_arch_destroy_vm;

    r = hardware_enable_all();
    if (r)
        goto out_err_no_disable;

    r = kvm_arch_post_init_vm(kvm);
    if (r)
        goto out_err;

    return kvm;
}
```

创建 VM

```c
int kvm_arch_init_vm(struct kvm *kvm, unsigned long type)
{
    int r;

    r = kvm_riscv_gstage_alloc_pgd(kvm);
    if (r)
        return r;

    r = kvm_riscv_gstage_vmid_init(kvm);
    if (r) {
        kvm_riscv_gstage_free_pgd(kvm);
        return r;
    }

    kvm_riscv_guest_timer_init(kvm);

    return 0;
}
```

硬件功能初始化

```
int kvm_arch_hardware_enable(void)
{
    hedeleg = 0;
    hedeleg |= (1UL << EXC_INST_MISALIGNED);
    hedeleg |= (1UL << EXC_BREAKPOINT);
    hedeleg |= (1UL << EXC_SYSCALL);
    hedeleg |= (1UL << EXC_INST_PAGE_FAULT);
    hedeleg |= (1UL << EXC_LOAD_PAGE_FAULT);
    hedeleg |= (1UL << EXC_STORE_PAGE_FAULT);
    csr_write(CSR_HEDELEG, hedeleg);

    hideleg = 0;
    hideleg |= (1UL << IRQ_VS_SOFT);
    hideleg |= (1UL << IRQ_VS_TIMER);
    hideleg |= (1UL << IRQ_VS_EXT);
    csr_write(CSR_HIDELEG, hideleg);

    /* VS should access only the time counter directly.
     * Everything else should trap */
    csr_write(CSR_HCOUNTEREN, 0x02);

    csr_write(CSR_HVIP, 0);
}
```

VM ioctl

```
static long kvm_vm_ioctl(struct file *filp,
                         unsigned int ioctl, unsigned long arg)
{
    switch (ioctl) {
    case KVM_CREATE_VCPU:
        r = kvm_vm_ioctl_create_vcpu(kvm, arg);
        break;
    case KVM_SET_USER_MEMORY_REGION: {
        struct kvm_userspace_memory_region kvm_userspace_mem;

        r = -EFAULT;
        if (copy_from_user(&kvm_userspace_mem, argp,
                           sizeof(kvm_userspace_mem)))
            goto out;

        r = kvm_vm_ioctl_set_memory_region(kvm, &kvm_userspace_mem);
        break;
    }
    /* ...... */
    default:
        r = kvm_arch_vm_ioctl(filp, ioctl, arg);
    }
out:
    return r;
}
```

映射 Memory Region

```c
static void gstage_wp_range(struct kvm *kvm, gpa_t start, gpa_t end)
{
    pte_t *ptep;
    u32 ptep_level;
    bool found_leaf;
    gpa_t addr = start;
    unsigned long page_size;

    while (addr < end) {
        found_leaf = gstage_get_leaf_entry(kvm, addr,
                                           &ptep, &ptep_level);
        ret = gstage_level_to_page_size(ptep_level, &page_size);
        if (ret)
            break;

        if (!found_leaf)
            goto next;

        if (!(addr & (page_size - 1)) && ((end - addr) >= page_size))
            gstage_op_pte(kvm, addr, ptep, ptep_level,
                          /* Write-protect */ GSTAGE_OP_WP);
next:
        addr += page_size;
    }
}
```

创建 vCPU

```
/* Creates some virtual cpus.  Good luck creating more than one. */
static int kvm_vm_ioctl_create_vcpu(struct kvm *kvm, u32 id)
{
    struct kvm_vcpu *vcpu;

    r = kvm_arch_vcpu_precreate(kvm, id);
    if (r) {
        mutex_unlock(&kvm->lock);
        return r;
    }
    kvm->created_vcpus++;
    kvm_vcpu_init(vcpu, kvm, id);

    r = kvm_arch_vcpu_create(vcpu);
    if (r)
        goto vcpu_free_run_page;

    /* Now it's all set up, let userspace reach it */
    kvm_get_kvm(kvm);
    r = create_vcpu_fd(vcpu);

    kvm_arch_vcpu_postcreate(vcpu);
    return r;
}
```

vCPU 初始化

```
int kvm_arch_vcpu_create(struct kvm_vcpu *vcpu)
{
    struct kvm_cpu_context *cntx;

    /* Setup vendor, arch, and implementation details */
    vcpu->arch.mvendorid = sbi_get_mvendorid();
    vcpu->arch.marchid = sbi_get_marchid();
    vcpu->arch.mimpid = sbi_get_mimpid();

    /* Setup reset state of shadow SSTATUS and HSTATUS CSRs */
    cntx = &vcpu->arch.guest_reset_context;
    cntx->sstatus = SR_SPP | SR_SPIE;
    cntx->hstatus = 0;
    cntx->hstatus |= HSTATUS_VTW;
    cntx->hstatus |= HSTATUS_SPVP;
    cntx->hstatus |= HSTATUS_SPV;

    /* Setup VCPU timer */
    kvm_riscv_vcpu_timer_init(vcpu);

    /* Reset VCPU */
    kvm_riscv_reset_vcpu(vcpu);
}
```

vCPU 初始化

```
void kvm_arch_vcpu_postcreate(struct kvm_vcpu *vcpu)
{
    /**
     * vcpu with id 0 is the designated boot cpu.
     * Keep all vcpus with non-zero id in power-off state so that
     * they can be brought up using SBI HSM extension.
     */
    if (vcpu->vcpu_idx != 0)
        kvm_riscv_vcpu_power_off(vcpu);
}
```

vCPU 运行

```c
int kvm_arch_vcpu_ioctl_run(struct kvm_vcpu *vcpu)
{
    int ret;
    struct kvm_cpu_trap trap;
    struct kvm_run *run = vcpu->run;

    while (ret > 0) {
        local_irq_disable();
        kvm_riscv_vcpu_enter_exit(vcpu);
        /*
         * Save SCAUSE, STVAL, HTVAL, and HTINST because we might
         * get an interrupt between __kvm_riscv_switch_to() and
         * local_irq_enable() which can potentially change CSRs.
         */
        trap.sepc = vcpu->arch.guest_context.sepc;
        trap.scause = csr_read(CSR_SCAUSE);
        trap.stval = csr_read(CSR_STVAL);
        trap.htval = csr_read(CSR_HTVAL);
        trap.htinst = csr_read(CSR_HTINST);

        local_irq_enable();
        ret = kvm_riscv_vcpu_exit(vcpu, run, &trap);
    }
}
```

切换到 Guest

```
__kvm_riscv_switch_to:
    /* Save Host GPRs (except A0 and T0-T6) */

    /* Load Guest CSR values */
    la  t4, __kvm_switch_return

    /* Save Host STVEC and change it to return path */
    csrrw   t4, CSR_STVEC, t4

    /* Store Host CSR values */

    /* Restore Guest GPRs (except A0) */

    /* Restore Guest A0 */
    REG_L   a0, (KVM_ARCH_GUEST_A0)(a0)

    /* Resume Guest */
    sret

    /* Back to Host */
    .align 2
__kvm_switch_return:
    /* ...... */
```

处理 VM exit

```c
/* Return > 0 to return to guest, < 0 on error, 0 (and set exit_reason)
 * on proper exit to userspace. */
int kvm_riscv_vcpu_exit(struct kvm_vcpu *vcpu, struct kvm_run *run,
                        struct kvm_cpu_trap *trap)
{
    ret = -EFAULT;
    run->exit_reason = KVM_EXIT_UNKNOWN;
    switch (trap->scause) {
    case EXC_INST_GUEST_PAGE_FAULT:
    case EXC_LOAD_GUEST_PAGE_FAULT:
    case EXC_STORE_GUEST_PAGE_FAULT:
        if (vcpu->arch.guest_context.hstatus & HSTATUS_SPV)
            ret = gstage_page_fault(vcpu, run, trap);
        break;
    case EXC_SUPERVISOR_SYSCALL:
        if (vcpu->arch.guest_context.hstatus & HSTATUS_SPV)
            ret = kvm_riscv_vcpu_sbi_ecall(vcpu, run);
        break;
    /* ...... */
    default:
        break;
    }
}
```

1 概述
- KVM 是什么
- QEMU 是什么
- QEMU 与 KVM 的交互

2 KVM 代码走读
- KVM 初始化
- 虚拟机创建
- vCPU 运行

3 QEMU 代码摘要
- Machine 初始化
- QEMU 的面向对象实现

1 概述
- KVM 是什么
- QEMU 是什么
- QEMU 与 KVM 的交互

2 KVM 代码走读
- KVM 初始化
- 虚拟机创建
- vCPU 运行

3 QEMU 代码摘要
- Machine 初始化
- QEMU 的面向对象实现

地址映射

```
static const MemMapEntry virt_memmap[] = {
    [VIRT_DEBUG] =        {        0x0,         0x100 },
    [VIRT_MROM] =         {     0x1000,        0xf000 },
    [VIRT_TEST] =         {   0x100000,        0x1000 },
    [VIRT_RTC] =          {   0x101000,        0x1000 },
    [VIRT_CLINT] =        {  0x2000000,       0x10000 },
    [VIRT_ACLINT_SSWI] =  {  0x2F00000,        0x4000 },
    [VIRT_PCIE_PIO] =     {  0x3000000,       0x10000 },
    [VIRT_PLATFORM_BUS] = {  0x4000000,     0x2000000 },
    [VIRT_PLIC] =         {  0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
    [VIRT_APLIC_M] =      {  0xc000000, APLIC_SIZE(VIRT_CPUS_MAX) },
    [VIRT_APLIC_S] =      {  0xd000000, APLIC_SIZE(VIRT_CPUS_MAX) },
    [VIRT_UART0] =        { 0x10000000,         0x100 },
    [VIRT_VIRTIO] =       { 0x10001000,        0x1000 },
    [VIRT_FW_CFG] =       { 0x10100000,          0x18 },
    [VIRT_FLASH] =        { 0x20000000,     0x4000000 },
    [VIRT_IMSIC_M] =      { 0x24000000, VIRT_IMSIC_MAX_SIZE },
    [VIRT_IMSIC_S] =      { 0x28000000, VIRT_IMSIC_MAX_SIZE },
    [VIRT_PCIE_ECAM] =    { 0x30000000,    0x10000000 },
    [VIRT_PCIE_MMIO] =    { 0x40000000,    0x40000000 },
    [VIRT_DRAM] =         { 0x80000000,           0x0 },
};
```

ROM 初始化

```
void riscv_setup_rom_reset_vec(/* ... */)
{
    start_addr_hi32 = start_addr >> 32;
    fdt_load_addr_hi32 = fdt_load_addr >> 32;
    /* reset vector */
    uint32_t reset_vec[10] = {
        0x00000297,           /* 1:  auipc  t0, %pcrel_hi(fw_dyn) */
        0x02828613,           /*     addi   a2, t0, %pcrel_lo(1b) */
        0xf1402573,           /*     csrr   a0, mhartid  */
        0x0202b583,           /*     ld     a1, 32(t0) */
        0x0182b283,           /*     ld     t0, 24(t0) */
        0x00028067,           /*     jr     t0 */
        start_addr,           /* start: .dword */
        start_addr_hi32,
        fdt_load_addr,        /* fdt_laddr: .dword */
        fdt_load_addr_hi32,
                              /* fw_dyn: */
    };
    /* copy in the reset vector in little_endian byte order */
    for (i = 0; i < ARRAY_SIZE(reset_vec); i++) {
        reset_vec[i] = cpu_to_le32(reset_vec[i]);
    }
    rom_add_blob_fixed_as("mrom.reset", reset_vec, sizeof(reset_vec),
                          rom_base, &address_space_memory);
}
```

1 概述
- KVM 是什么
- QEMU 是什么
- QEMU 与 KVM 的交互

2 KVM 代码走读
- KVM 初始化
- 虚拟机创建
- vCPU 运行

3 QEMU 代码摘要
- Machine 初始化
- QEMU 的面向对象实现

QEMU 的面向对象实现

QEMU 中用 C 实现了一套面向对象的框架，包括类、实例、接口等功能，支持继承、多态。

所有的类都继承自 Object：

```
struct Object
{
    /* private: */
    ObjectClass *class;
    ObjectFree *free;
    GHashTable *properties;
    uint32_t ref;
    Object *parent;
};
```

QEMU 的面向对象实现

每个类都有一个对应的 Class 结构体，以 Object 为例:

```
struct ObjectClass
{
    /* private: */
    Type type;
    GSList *interfaces;

    const char *object_cast_cache[OBJECT_CLASS_CAST_CACHE];
    const char *class_cast_cache[OBJECT_CLASS_CAST_CACHE];

    ObjectUnparent *unparent;

    GHashTable *properties;
};
```

Object 用来存储实例中的成员，ObjectClass 用来存储类的静态成员

QEMU 的面向对象实现

一个类必须将自己父类的结构体放在自己结构体的最前面:

```
struct DeviceState {
    /*< private >*/
    Object parent_obj;
    /*< public >*/
    char *id;
    char *canonical_path;
    bool realized;
    /* ...... */
};
```

这样可以让每个类的实例都可以强制转换成 `Object` 的指针，并得到其 metadata

QEMU 的面向对象实现

定义一个类后必须显式地注册，如:

```
static const TypeInfo led_info = {
    .name = TYPE_LED,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(LEDState),
    .class_init = led_class_init
};

static void led_register_types(void)
{
    type_register_static(&led_info);
}

type_init(led_register_types)
```

QEMU 的面向对象实现

类的注册通过 C 语言的 `__attribute__((constructor))` 实现。
标有 constructor 的函数会在 main 函数开始前执行

```
#define module_init(function, type) \
static void __attribute__((constructor)) \
    do_qemu_init_ ## function(void) \
{ \
    register_module_init(function, type); \
}

#define type_init(function) \
    module_init(function, MODULE_INIT_QOM)
```

QEMU 的面向对象实现

类的向上和向下转换直接通过指针强制转换实现，但需要经过复杂的检查

```
ObjectClass *object_class_dynamic_cast(ObjectClass *class,
                                       const char *typename)
{
    /* A simple fast path that can trigger a lot for leaf classes.  */
    type = class->type;
    if (type->name == typename) {
        return class;
    }
    target_type = type_get_by_name(typename);
    if (!target_type) {
        /* target class type unknown, so fail the cast */
        return NULL;
    }
    if (type->class->interfaces &&
            type_is_ancestor(target_type, type_interface)) {
        /* ...... */
    } else if (type_is_ancestor(type, target_type)) {
        ret = class;
    }
    return ret;
}
```

附录

KVM API 文档: https://docs.kernel.org/virt/kvm/api.html

KVM 中还有两个比较重要，但较复杂、不方便用代码体现的功能，如 irqfd, ioeventfd，可以查阅文档了解相关细节

Q & A