KVM VHOST中irqfd的使用

2018-01-18

其实在之前的文章中已经简要介绍了VHOST中通过irqfd通知guest，但是并没有对irqfd的具体工作机制做深入分析，本节简要对irqfd的工作机制分析下。这里暂且不讨论具体中断虚拟化的问题，因为那是另一个内容，这里仅仅讨论vhost如何使用中断的方式对guest进行通知，这里答案就是IRQFD。

在vhost的初始化过程中，qemu会通过ioctl系统调用和KVM交互，注册guestnotifer，见kvm_irqchip_assign_irqfd函数。qemu中对irqfd的定义如下

```
struct kvm_irqfd {
    __u32 fd;
    __u32 gsi;
    __u32 flags;
    __u8  pad[20];
};
```

该结构是32个字节对齐的，fd是irqfd绑定的eventfd的描述符，gsi是给irqfd绑定的一个 全局中断号，flag纪录标志位，可以判定本次请求是irqfd的注册还是消除。qemu把该结构传递给KVM，KVM中做了什么处理呢?

在kvm_vm_ioctl函数中的case KVM_IRQFD分支，这里把kvm_irqfd复制到内核中，调用了kvm_irqfd函数。经过对flags的判断，如果发现是注册的话，就调用kvm_irqfd_assign函数，并把kvm_irqfd结构作为参数传递进去，该函数是注册irqfd的核心函数。在此之前，先看下内核中对于irqfd对应的数据结构

```
struct _irqfd {
    /* Used for MSI fast-path */
    struct kvm *kvm;
    wait_queue_t wait;
    /* Update side is protected by irqfds.lock */
    struct kvm_kernel_irq_routing_entry __rcu *irq_entry;
    /* Used for level IRQ fast-path */
    int gsi;
    struct work_struct inject;
    /* The resampler used by this irqfd (resampler-only) */
    struct _irqfd_resampler *resampler;
    /* Eventfd notified on resample (resampler-only) */
    struct eventfd_ctx *resamplefd;
    /* Entry in list of irqfds for a resampler (resampler-only) */
    struct list_head resampler_link;
    /* Used for setup/shutdown */
    struct eventfd_ctx *eventfd;
    struct list_head list;//对应KVM中的irqfd链表
    poll_table pt;
    struct work_struct shutdown;
};
```

irqfd在内核中必然属于某个KVM结构体，因此首个字段便是对应KVM结构体的指针；在KVM结构体中也有对应的irqfd的链表；第二个是wait，是一个等待队列对象，irqfd需要绑定一个eventfd，且在这个eventfd上等待，当eventfd有信号是，就会处理该irqfd。目前暂且忽略和中断路由相关的字段；GSI正是用户空间传递过来的全局中断号；indect和shutdown是两个工作队列对象；pt是一个poll_table对象，作用后面使用时在进行描述。接下来看kvm_irqfd_assign函数代码，这里我们分成三部分：准备阶段、对resamplefd的处理，对普通irqfd的处理。第二种我们暂时忽略

第一阶段：

邮箱：zhunxun@gmail.com

< 2020年5月 >

日	一	二	三	四	五	六
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

搜索

找找看

谷歌搜索

PostCategories

C语言(2)
IO Virtualization(3)
KVM虚拟化技术(26)
linux 内核源码分析(61)
Linux日常应用(3)
linux时间子系统(3)
qemu(10)
seLinux(1)
windows内核(5)
调试技巧(2)
内存管理(8)
日常技能(3)
容器技术(2)
生活杂谈(1)
网络(5)
文件系统(4)
硬件(4)

PostArchives

2018/4(1)
2018/2(1)
2018/1(3)
2017/12(2)
2017/11(4)
2017/9(3)
2017/8(1)
2017/7(8)
2017/6(6)
2017/5(9)
2017/4(15)
2017/3(5)
2017/2(1)
2016/12(1)
2016/11(11)
2016/10(8)
2016/9(13)

ArticleCategories

时态分析(1)

Recent Comments

1. Re:virtio前端驱动详解
我看了下，Linux-4.18.2中的vp_notify()函数。bool vp_notify(struct virtqueue *vq){ /* we write the queue's selec...
--Linux-inside
2. Re:virtIO之VHOST工作原理简析

再问一个问题，从设置ioeventfd那个流程来看的话是guest发起一个IO，首先会陷入到kvm中，然后由kvm向qemu发送一个IO到来的event，最后IO才被处理，是这样的吗?

--Linux-inside

3. Re:virtIO之VHOST工作原理简析

你好。设置ioeventfd这个部分和guest里面的virtio前端驱动有关系吗?
设置ioeventfd和virtio前端驱动是如何发生联系起来的？谢谢。

--Linux-inside

4. Re:QEMU IO事件处理框架

良心博主，怎么停跟了，太可惜了。

--黄铁牛

5. Re:linux 逆向映射机制浅析

小哥哥520脱单了么

--黄铁牛

Top Posts

1. 详解操作系统中断(21154)
2. PCI 设备详解一(15808)
3. 进程的挂起、阻塞和睡眠(13714)
4. Linux下桥接模式详解一(13467)
5. virtio后端驱动详解(10539)

推荐排行榜

1. 进程的挂起、阻塞和睡眠(6)
2. qemu-kvm内存虚拟化1(2)
3. 为何要写博客(2)
4. virtIO前后端notify机制详解(2)
5. 详解操作系统中断(2)

```
    irqfd = kzalloc(sizeof(*irqfd), GFP_KERNEL);
    if (!irqfd)
        return -ENOMEM;

    irqfd->kvm = kvm;
    irqfd->gsi = args->gsi;
    INIT_LIST_HEAD(&irqfd->list);
    /*设置工作队列的处理函数*/
    INIT_WORK(&irqfd->inject, irqfd_inject);
    INIT_WORK(&irqfd->shutdown, irqfd_shutdown);
    /*得到eventfd对应的file结构*/
    file = eventfd_fget(args->fd);
    if (IS_ERR(file)) {
        ret = PTR_ERR(file);
        goto fail;
    }
    /*得到eventfd对应的描述符*/
    eventfd = eventfd_ctx_fileget(file);
    if (IS_ERR(eventfd)) {
        ret = PTR_ERR(eventfd);
        goto fail;
    }
    /*进行绑定*/
    irqfd->eventfd = eventfd;
```

这里的任务比较明确，在内核为irqfd分配内存，设置相关字段。重要的是对于前面提到的inject和shutdown两个工作队列绑定了处理函数。然后根据用户空间传递进来的fd，解析得到对应的eventfd_ct结构，并和irqfd进行绑定。

第三阶段：

```
/*
     * Install our own custom wake-up handling so we are notified via
     * a callback whenever someone signals the underlying eventfd
     */
     /*设置等待队列的处理函数*/
    init_waitqueue_func_entry(&irqfd->wait, irqfd_wakeup);
    //poll函数中会调用,即eventfd_poll中，这里仅仅是绑定irqfd_ptable_queue_proc函数和poll_table
    init_poll_funcptr(&irqfd->pt, irqfd_ptable_queue_proc);

    spin_lock_irq(&kvm->irqfds.lock);

    ret = 0;
    /*检查针对当前eventfd是否已经存在irqfd与之对应*/
    list_for_each_entry(tmp, &kvm->irqfds.items, list) {
        if (irqfd->eventfd != tmp->eventfd)
            continue;
        /* This fd is used for another irq already. */
        ret = -EBUSY;
        spin_unlock_irq(&kvm->irqfds.lock);
        goto fail;
    }
    /*获取kvm的irq路由表*/
    irq_rt = rcu_dereference_protected(kvm->irq_routing,
                       lockdep_is_held(&kvm->irqfds.lock));
    /*更新kvm相关信息，主要是irq路由项目*/
    irqfd_update(kvm, irqfd, irq_rt);
    /*调用poll函数，把irqfd加入到eventfd的等待队列中  eventfd_poll*/
    events = file->f_op->poll(file, &irqfd->pt);
    /*把该irqfd加入到虚拟机kvm的链表中*/
    list_add_tail(&irqfd->list, &kvm->irqfds.items);

    /*
     * Check if there was an event already pending on the eventfd
     * before we registered, and trigger it as if we didn't miss it.
     */
     /*如果有可用事件，就执行注入，把中断注入任务加入到系统全局工作队列*/
    if (events & POLLIN)
        schedule_work(&irqfd->inject);

    spin_unlock_irq(&kvm->irqfds.lock);

    /*
     * do not drop the file until the irqfd is fully initialized, otherwise
```

```
     * we might race against the POLLHUP
     */
    fput(file);

    return 0;
```

第三阶段首先设置了irqfd中等待对象的唤醒函数irqfd_wakeup，然后用init_poll_funcptr对irqfd中的poll_table进行初始化，主要是绑定一个排队函数irqfd_ptable_queue_proc，其实这两步也可以看做是准备工作的一部分，不过由于第二部分的存在，只能安排在第三部分。接下来遍历KVM结构中的irqfd链表，检查是否有irqfd已经绑定了本次需要的eventfd，言外之意是一个eventfd只能绑定一个irqfd。如果检查没有问题，则会调用irqfd_update函数更新中断路由表项目。并调用VFS的poll函数对eventfd进行监听，并把irqfd加入到KVM维护的链表中。如果发现现在已经有可用的信号（可用事件），就立刻调用schedule_work，调度irqfd->inject工作对象，执行中断的注入。否则，中断的注入由系统统一处理。具体怎么处理呢？先看下poll函数做了什么，这里poll函数对应eventfd_poll函数

```
static unsigned int eventfd_poll(struct file *file, poll_table *wait)
{
    struct eventfd_ctx *ctx = file->private_data;
    unsigned int events = 0;
    unsigned long flags;
    /*执行poll_table中的函数，把irqfd加入eventfd的到等待队列中*/
    poll_wait(file, &ctx->wqh, wait);

    spin_lock_irqsave(&ctx->wqh.lock, flags);

    if (ctx->count > 0)
        events |= POLLIN;//表明现在可以read
    if (ctx->count == ULLONG_MAX)
        events |= POLLERR;
    if (ULLONG_MAX - 1 > ctx->count)
        events |= POLLOUT;//现在可以write
    spin_unlock_irqrestore(&ctx->wqh.lock, flags);

    return events;
}
```

该函数的可以分成两部分，首先是通过poll_wait函数执行poll_table中的函数，即前面提到的irqfd_ptable_queue_proc，该函数把irqfd中的wait对象加入到eventfd的等待队列中。这样irqfd和eventfd的双向关系就建立起来了。接下来的任务是判断eventfd的当前状态，并返回结果。如果count大于0，则表示现在可读，即有信号；如果count小于ULLONG_MAX-1，则表示目前该描述符还可以接受信号触发。当然这种情况绝大部分是满足的。那么在evnetfd上等待的对象什么时候会被处理呢？答案是当有操作试图给evnetfd发送信号时，即eventfd_signal函数

```
__u64 eventfd_signal(struct eventfd_ctx *ctx, __u64 n)
{
    unsigned long flags;
    spin_lock_irqsave(&ctx->wqh.lock, flags);
    if (ULLONG_MAX - ctx->count < n)
        n = ULLONG_MAX - ctx->count;
    ctx->count += n;
    /*mainly judge if wait is empty,如果等待队列不为空，则进行处理*/
    if (waitqueue_active(&ctx->wqh))
        wake_up_locked_poll(&ctx->wqh, POLLIN);
    spin_unlock_irqrestore(&ctx->wqh.lock, flags);

    return n;
}
```

函数首要任务是向对应的eventfd传送信号，实质就是增加count值。因为此时count值一定大于0，即状态可用，则检查等待队列中时候有等待对象，如果有，则调用wake_up_locked_poll函数进行处理

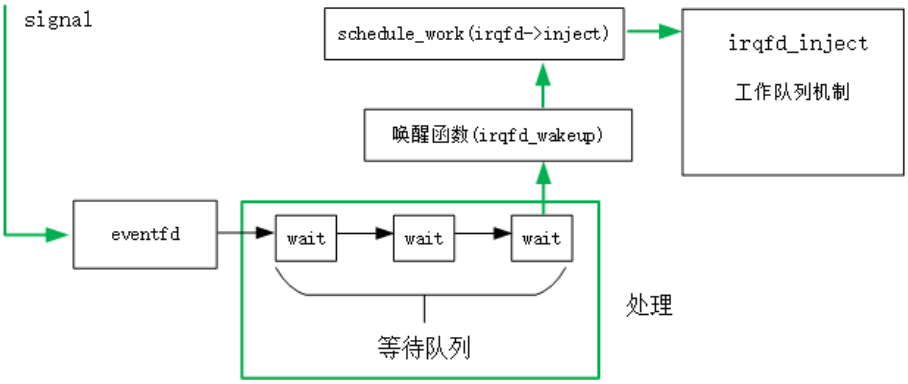
```
#define wake_up_locked_poll(x, m)                \
    __wake_up_locked_key((x), TASK_NORMAL, (void *) (m))

void __wake_up_locked_key(wait_queue_head_t *q, unsigned int mode, void *key)
{
    __wake_up_common(q, mode, 1, 0, key);
}

static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
            int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;

    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
        unsigned flags = curr->flags;

        if (curr->func(curr, mode, wake_flags, key) &&
                (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;
    }
}
```

具体处理过程就是遍历等待队列上所有等待对象，并执行对应的唤醒函数。针对irqfd的唤醒函数前面已经提到，是irqfd_wakeup，在该函数中会对普通中断执行schedule_work(&irqfd->inject);这样对于irqfd注册的inject工作对象处理函数就会得到执行，于是，中断就会被执行注入。到这里不妨在看看schedule_work发生了什么

```
static inline bool schedule_work(struct work_struct *work)
{
    return queue_work(system_wq, work);
}
```

原来如此，该函数把工作对象加入到内核全局的工作队列中，接下来的处理就有内核自身完成了。

从给eventfd发送信号，到中断注入的执行流大致如下所示：



以马内利：

参考资料：

- qemu2.7源码
- Linux 内核3.10.1源码

分类：KVM虚拟化技术, linux 内核源码分析

jack.chen
关注 - 12
粉丝 - 44
+加关注

« 上一篇： 翻墙软件推荐
» 下一篇： 聊一聊Linux中的工作队列

注册用户登录后才能发表评论，请 登录 或 注册， 访问 网站首页。

最新 IT 新闻:
· 腾讯在列！微软宣布超140家工作室为Xbox Series X开发游戏
· 黑客声称从微软GitHub私人数据库当中盗取500GB数据
· IBM开源用于简化AI模型开发的Elyra工具包
· 中国网民人均安装63个App：腾讯系一家独大
· Lyft颁布新规：强制要求乘客和司机佩戴口罩
» 更多新闻...

以马内利