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

Linux eventfd分析

2017-07-20

eventfd在linux中是一个较新的进程通信方式，和信号量等不同的是event不仅可以用于进程间的通信，还可以用户内核发信号给用户层的进程。eventfd在virtIO后端驱动vHost的实现中作为vhost和KVM交互的媒介，起到了重大作用。本节结合linux源码就eventfd的具体实现坐下简要分析。

eventfd在用户层下有函数

```
#include <sys/eventfd.h>

int eventfd(unsigned int initval, int flags);
```

该函数返回一个文件描述符，类似于其他的文件描述符操作，可以对该描述符进行一系列的操作，如读、写、poll、select等，当然这里我们仅仅考虑read、write。看下该函数的内核实现

```
SYSCALL_DEFINE2(eventfd2, unsigned int, count, int, flags)
{
    int fd, error;
    struct file *file;
    error = get_unused_fd_flags(flags & EFD_SHARED_FCNTL_FLAGS);
    if (error < 0)
        return error;
    fd = error;
    file = eventfd_file_create(count, flags);
    if (IS_ERR(file)) {
        error = PTR_ERR(file);
        goto err_put_unused_fd;
    }
    fd_install(fd, file);
    return fd;
err_put_unused_fd:
    put_unused_fd(fd);
    return error;
}
```

代码本身很是简单，首先 获取一个空闲的文件描述符，这个和普通的文件描述符没有区别。然后调用eventfd_file_create创建了一个file结构。该函数中有针对eventfd的一系列操作，看下该函数

```
struct file *eventfd_file_create(unsigned int count, int flags)
{
    struct file *file;
    struct eventfd_ctx *ctx;

    /* Check the EFD_* constants for consistency.  */
    BUILD_BUG_ON(EFD_CLOEXEC != O_CLOEXEC);
    BUILD_BUG_ON(EFD_NONBLOCK != O_NONBLOCK);

    if (flags & ~EFD_FLAGS_SET)
        return ERR_PTR(-EINVAL);

    ctx = kmalloc(sizeof(*ctx), GFP_KERNEL);
    if (!ctx)
        return ERR_PTR(-ENOMEM);

    kref_init(&ctx->kref);
    init_waitqueue_head(&ctx->wqh);
    ctx->count = count;
```

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(21155)
2. PCI 设备详解一(15808)
3. 进程的挂起、阻塞和睡眠(13715)
4. Linux下桥接模式详解一(13468)
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)

```
    ctx->flags = flags;

    file = anon_inode_getfile("[eventfd]", &eventfd_fops, ctx,
                  O_RDWR | (flags & EFD_SHARED_FCNTL_FLAGS));
    if (IS_ERR(file))
        eventfd_free_ctx(ctx);

    return file;
}
```

这里说明下，每个eventfd在内核对应一个eventfd_ctx结构，该结构后面咱们再细讲，函数中首先给该结构分配 了内存然后做初始化，注意有个等待队列和count，等待队列就是当进程需要阻塞的时候挂在对应evnetfd的等待队列上，而count就是read、write操作的值。接着就调用anon_inode_getfile获取一个file对象，具体也没什么好说的，只是注意这里把刚才分配好的eventfd_ctx作为file结构的私有成员即private_data，并且关联了eventfd自身的操作函数表eventfd_fops， 里面实现的函数不多，如下

```
static const struct file_operations eventfd_fops = {
#ifdef CONFIG_PROC_FS
    .show_fdinfo    = eventfd_show_fdinfo,
#endif
    .release    = eventfd_release,
    .poll        = eventfd_poll,
    .read        = eventfd_read,
    .write        = eventfd_write,
    .llseek        = noop_llseek,
};
```

我们重点看read和write函数。当用户空间对eventfd文件描述符发起read操作时，最终要调用到上面函数表中的eventfd_read函数，

```
static ssize_t eventfd_read(struct file *file, char __user *buf, size_t count,
                loff_t *ppos)
{
    struct eventfd_ctx *ctx = file->private_data;
    ssize_t res;
    __u64 cnt;
    if (count < sizeof(cnt))
        return -EINVAL;
    res = eventfd_ctx_read(ctx, file->f_flags & O_NONBLOCK, &cnt);
    if (res < 0)
        return res;
    return put_user(cnt, (__u64 __user *) buf) ? -EFAULT : sizeof(cnt);
}
```

首先从private_data获取eventfd_ctx，然后判断请求读取的大小是否满足条件，这里count是64位即8个字节，所以最小读取8个字节，如果不足则错误。没问题就调用eventfd_ctx_read，该函数实际返回eventfd_ctx中的count计数，并清零，如果读取有问题则返回，否则把值写入到用户空间。前面eventfd_ctx_read是读取的核心，什么时候会返回小于0的值呢，我们看下该函数的实现

```
ssize_t eventfd_ctx_read(struct eventfd_ctx *ctx, int no_wait, __u64 *cnt)
{
    ssize_t res;
    DECLARE_WAITQUEUE(wait, current);

    spin_lock_irq(&ctx->wqh.lock);
    *cnt = 0;
    res = -EAGAIN;
    if (ctx->count > 0)
        res = 0;
    else if (!no_wait) {
        /*add to wait queue*/
        __add_wait_queue(&ctx->wqh, &wait);
```

```
        for (;;) {
            /*设置阻塞状态*/
            set_current_state(TASK_INTERRUPTIBLE);
            /*如果信号变为有状态。则break*/
            if (ctx->count > 0) {
                res = 0;
                break;
            }
            /*如果有未处理的信号，也break，进行处理*/
            if (signal_pending(current)) {
                res = -ERESTARTSYS;
                break;
            }
            /*否则触发调度器执行调度*/
            spin_unlock_irq(&ctx->wqh.lock);
            schedule();
            spin_lock_irq(&ctx->wqh.lock);
        }
        /*remove from the wait queue*/
        __remove_wait_queue(&ctx->wqh, &wait);
        /*set processs state*/
        __set_current_state(TASK_RUNNING);
    }
    if (likely(res == 0)) {
        /*read fdcount again*/
        eventfd_ctx_do_read(ctx, cnt);
        /**/
        if (waitqueue_active(&ctx->wqh))
            wake_up_locked_poll(&ctx->wqh, POLLOUT);
    }
    spin_unlock_irq(&ctx->wqh.lock);

    return res;
}
```

该函数比较长，我们慢慢分析，首先操作eventfd_ctx要加锁保证安全。起初res初始化为-EAGAIN，如果count计数大于0，那么对res置0，否则意味着count=0（count不会小于0），这种情况下看传递进来的参数标志，如果设置了O_NONBLOCK，则就不需等待，直接返回res.这正是前面说的返回值小于0的情况。如果没有指定O_NONBLOCK标志，此时由于读取不到count值（count值为0），就会在这里阻塞。具体把当前进程加入到eventfd_ctx的等待队列，这里有必要说下DECLARE_WAITQUEUE(wait, current)，该宏声明并初始化一个wait_queue_t对象，其关联的函数为default_wake_function，是作为唤醒函数存在。OK，接下上面，加入到队列后进入一个死循环，设置当前进程状态为TASK_INTERRUPTIBLE，并不断检查count值，如果count大于0了，意味着有信号了，就设置res=0，然后break，然后把进程从等待队列去掉，然后设置状态TASK_RUNNING。如果count值为0，则检查是否有挂起的信号，如果有信号，同样需要先对信号进行处理，不过这就以为这read失败了。都么有的话就正常阻塞，调用调度器进行调度。break之后，如果res==0，对count值进行读取，这里对应上面循环中判断count值大于0的情况。具体读取通过eventfd_ctx_do_read函数，该函数很简单

```
static void eventfd_ctx_do_read(struct eventfd_ctx *ctx, __u64 *cnt)
{
    *cnt = (ctx->flags & EFD_SEMAPHORE) ? 1 : ctx->count;
    ctx->count -= *cnt;
}
```

如果没有指定EFD_SEMAPHORE标志就返回count值，该标志是指定eventfd像信号量一样使用，不过在2.6之后的内核都设置为0了。然后对count做减法，实际上减去之后就为0了。在读取值之后count值就变小了，之前如果有在该eventfd上阻塞的write进程，现在就可以唤醒了，所以这里检查了下，如果等待队列还有进程，则调用wake_up_locked_poll对对应的进程进行唤醒。

用户空间的write操作最终要调用到eventfd_write，不过该函数的实现和上面read操作类似，这里就不重复，感兴趣可以自行分析源码。前面说内核也可以主动的对eventfd发送信号，这里就是通过eventfd_signal函数实现

```
__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*/
    if (waitqueue_active(&ctx->wqh))
        wake_up_locked_poll(&ctx->wqh, POLLIN);
    spin_unlock_irqrestore(&ctx->wqh.lock, flags);

    return n;
}
```

该函数和write函数类似，不过不会阻塞，如果指定的n太大导致count加上之后超过ULLONG_MAX，就去n为当前count和ULLONG_MAX的差值，即不会让count溢出。然后如果等待队列有等待的进程，则对其进程唤醒，当然唤醒的应该是需要读操作的进程。

到这里对于eventfd的介绍基本就完成了，总的来说很简单的一个东西，不过经过上面分析不难发现，eventfd应该归结于低级通信行列，即不适用于传递大量数据，仅仅用于通知或者同步操作，还要注意的是，该文件描述符并不对应固定的磁盘文件，故类似于无名管道，这里也仅仅用于有亲缘关系之间的进程通信！。

关于eventfd的使用方法，参考手册：https://linux.die.net/man/2/eventfd

以马内利

参考资料：

linux内核3.10.1源码

分类：C语言, linux 内核源码分析

jack.chen
关注 - 12
粉丝 - 44
+加关注

1 0

« 上一篇： linux内存管理之vmalloc函数分析
» 下一篇： linux IO多路复用POLL机制深入分析

posted @ 2017-07-20 20:22 jack.chen Views(7494) Comments(0) Edit 收藏

最新 IT 新闻:
· 契合Chrome深色模式 谷歌搜索结果页面现可跟随深色显示
· 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
» 更多新闻...

以马内利