linux IO多路复用POLL机制深入分析

POLL机制的作用这里就不进行介绍，根据linux man手册，解释为在一个文件描述符上等待某个事件。按照抽象一点的理解，当某个事件被触发（条件被满足），文件描述符变为有状态，那么用户空间可以根据此进行操作，结合多个文件描述符，可以实现文件描述符的无阻塞访问。其实个人感觉这里的无阻塞主要是在监听多个文件描述符的情况下，把多个文件描述符放在一起管理，哪个有状态了就处理哪个，这样好像在调用具体的处理函数前比如read调用前，加了一层管理层用于检查是否可以进行操作，当所有的文件描述符都不可用，还是要阻塞，poll函数原型如下：

```
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

首个参数fds是一个数组指针，指向文件描述符集合，每个文件描述符对应一个pollfd 结构，nfds表示fd的个数，timeout指定阻塞的时间，单位为毫秒。用户空间调用后该函数后，经过poll系统调用进入内核，看下内核的实现在select.c文件中

```
SYSCALL_DEFINE3(poll, struct pollfd __user *, ufds, unsigned int,
nfds,int, timeout_msecs)
```

主要还是调用了do_sys_poll（）函数

```
int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds,
        struct timespec *end_time)
{
    struct poll_wqueues table;
     int err = -EFAULT, fdcount, len, size;
    /* Allocate small arguments on the stack to save memory and be
       faster - use long to make sure the buffer is aligned properly
       on 64 bit archs to avoid unaligned access */
    long stack_pps[POLL_STACK_ALLOC/sizeof(long)];
    struct poll_list *const head = (struct poll_list *)stack_pps;
     struct poll_list *walk = head;
     unsigned long todo = nfds;

    if (nfds > rlimit(RLIMIT_NOFILE))
        return -EINVAL;

    len = min_t(unsigned int, nfds, N_STACK_PPS);
    /*fd通过walk管理，多个walk形成链表*/
    for (;;) {
        walk->next = NULL;
        walk->len = len;
        if (!len)
            break;
        /*每次copy len个fd*/
        if (copy_from_user(walk->entries, ufds + nfds-todo,
                    sizeof(struct pollfd) * walk->len))
            goto out_fds;

        todo -= walk->len;
        if (!todo)
            break;

        len = min(todo, POLLFD_PER_PAGE);
        size = sizeof(struct poll_list) + sizeof(struct pollfd) * len;
        walk = walk->next = kmalloc(size, GFP_KERNEL);
        if (!walk) {
            err = -ENOMEM;
            goto out_fds;
        }
    }
```


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(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)

```
    /*通过一次poll提交的fd共用一个table*/
    poll_initwait(&table);
    fdcount = do_poll(nfds, head, &table, end_time);
    poll_freewait(&table);
    /*向用户空间返回监控结果，主要是当前的可用事件*/
    for (walk = head; walk; walk = walk->next) {
        struct pollfd *fds = walk->entries;
        int j;

        for (j = 0; j < walk->len; j++, ufds++)
            /*把实际的结果返回给用户空间*/
            if (__put_user(fds[j].revents, &ufds->revents))
                goto out_fds;
      }

    err = fdcount;
out_fds:
    /*释放内存*/
    walk = head->next;
    while (walk) {
        struct poll_list *pos = walk;
        walk = walk->next;
        kfree(pos);
    }
    /*正常情况下返回fd 的数目*/
    return err;
}
```

文件描述符在内核中通过poll_list结构管理，每个poll_list管理一定数目的fd，多个poll_list形成一条链表，首个poll_list是在栈上分配的，用以加速访问。但是如果一个poll_list不够用，则必须要进行再次分配，再次就通过kmalloc进行分配了，看下代码，首先在栈上 申请了POLL_STACK_ALLOC个字节的空间，按照8个字节对齐，然后转化成了poll_list指针。如果参数中指定的nfds大于RLIMIT_NOFILE，则返回错误，否则在N_STACK_PPS和nfds中取小者作为首个poll_list的长度，即fd的个数。下面进入一个死循环，主要目的是分批次拷贝fd.没什么好说的，看下poll_list结构

```
struct poll_list {
    struct poll_list *next;//连接下一个poll_list
    int len;//该list中fd的数目
    struct pollfd entries[0];//每个entry对应一个fd
};
```

拷贝完之后，需要进行核心处理了。首先初始化一个poll_wqueues结构

```
struct poll_wqueues {
    poll_table pt;
    struct poll_table_page *table;
    struct task_struct *polling_task;
    int triggered;
    int error;
    int inline_index;
    struct poll_table_entry inline_entries[N_INLINE_POLL_ENTRIES];
};
```

pt是一个poll_table_struct结构，该结构内容如下，前者_qproc是一个函数指针，在驱动的poll函数中会对齐进行调用，主要是负责吧当前进程加入到等待队列中，key是请求的事件掩码，默认请求所有可用事件。

```
typedef struct poll_table_struct {
    poll_queue_proc _qproc;//函数指针，一般负责吧当前进程加入到等待队列
    unsigned long _key;//请求的事件掩码
} poll_table;
```

table指向poll_table_page结构，该结构管理poll_table_entry，每一个entry对应一个fd，所有的poll_table_page形成链表，不过在poll_table_page结构之前，优先使用的是poll_wqueues结构中的inline_entries，该数组是伴随poll_wqueues一起申请，inline_index指向其中的最后一个已经使用的元

素，用以加速分配，OK结构描述就到这里，看函数代码。对poll_wqueues初始化后就调用了do_poll，该函数的核心功能就是对每一个fd，调用驱动的poll函数，从而获取各个fd的状态。代码简单列举下

```
for (;;) {
        struct poll_list *walk;

        for (walk = list; walk != NULL; walk = walk->next) {
            struct pollfd * pfd, * pfd_end;

            pfd = walk->entries;
            pfd_end = pfd + walk->len;
            for (; pfd != pfd_end; pfd++) {
                /*
                 * Fish for events. If we found one, record it
                 * and kill poll_table->_qproc, so we don't
                 * needlessly register any other waiters after
                 * this. They'll get immediately deregistered
                 * when we break out and return.
                 */
                if (do_pollfd(pfd, pt)) {
                    count++;
                    pt->_qproc = NULL;
                }
            }
        }
        /*
         * All waiters have already been registered, so don't provide
         * a poll_table->_qproc to them on the next loop iteration.
         */
        pt->_qproc = NULL;
        if (!count) {
            count = wait->error;
            if (signal_pending(current))
                count = -EINTR;
        }
        if (count || timed_out)
            break;

        /*
         * If this is the first loop and we have a timeout
         * given, then we convert to ktime_t and set the to
         * pointer to the expiry value.
         */
        if (end_time && !to) {
            expire = timespec_to_ktime(*end_time);
            to = &expire;
        }

        if (!poll_schedule_timeout(wait, TASK_INTERRUPTIBLE, to, slack))
            timed_out = 1;
    }
```

核心业务都在这一个for循环中实现，前半部分正是针对每一个pfd调用do_pollfd函数，该函数是一个内敛函数，用以调用具体的poll函数来获取fd状态，并记录到pollfd->revents中。需要注意的是，虽然对各个fd均调用了poll函数，但是并不是所有的调用都会把进程加到等待队列，只要找到任何一个有状态的fd，即do_pollfd返回非0，则之后的poll调用均不会把进程加入到等待队列。因为加入到等待队列的本质目的是让该进程在阻塞的时候可以被正常唤醒。但是现在既然存在fd有状态，那么本次调用就不会阻塞，所以就不用在加入等待队列了。count记录有状态的fd的个数。如果count为0，就有三种可能性

1、如果有信号需要处理，则返回处理信号

2、如果超时了，则返回

3、阻塞，这是会触发调度器，下次调度还是会挨个检查fd状态，不过不会加入等待队列了。

而如果count不为0，则直接返回了。

以马内利

参考资料

linux3.10.1内核源码

分类: linux 内核源码分析

jack.chen
关注 - 12
粉丝 - 44
+加关注

0 0

« 上一篇： Linux eventfd分析
» 下一篇： virtIO之VHOST工作原理简析

posted @ 2017-07-31 19:50 jack.chen Views(658) Comments(0) Edit 收藏

以马内利