努力成为 linux kernel hacker 的人李万鹏原创作品，为梦而战。转载请标明出处 http://blog.csdn.net/woshixingaaa/archive/2011/05/19/6433337.aspx

input 子系统最重要的部分就是向上层 report 了。这里还是先介绍几个数据结构：

```
struct input_event {
    struct timeval time;  //事件发生的时间
    __u16 type;           //事件类型
    __u16 code;           //子事件
    __s32 value;          //事件的 value
};
```

```
struct evdev_client {
    struct input_event buffer[EVDEV_BUFFER_SIZE];       //可以同时管理 EVDEV_BUFFER_SIZE(64)个事件
    int head;                                           //存储事件从 head 开始
    int tail;                                           //取出事件从 tail 开始
    spinlock_t buffer_lock; /* protects access to buffer, head and tail */
    struct fasync_struct *fasync;                       //异步通知事件发生
    struct evdev *evdev;                                //指向本 evdev_client 归属的 evdev
    struct list_head node;                              //用于挂载到 evdev 的链表头 client_list 上
};
```

```
static struct input_handler evdev_handler = {
    .event      = evdev_event,
    .connect    = evdev_connect,
    .disconnect = evdev_disconnect,
    .fops       = &evdev_fops,
    .minor      = EVDEV_MINOR_BASE,
    .name       = "evdev",
    .id_table   = evdev_ids,
};
```

这里的次设备号是 EVDEV_MINOR_BASE(64)，也就是说 evdev_handler 所表示的设备文件范围(13,64)~(13,64+32)。
如下一个结构体:evdev_handler 匹配所有设备。

```
static const struct input_device_id evdev_ids[] = {
    { .driver_info = 1 },   /* Matches all devices */
    { },            /* Terminating zero entry */
};
```

这个是 evdev_handler 是 fops，下面的讲解中会用到其中的 open,read 函数。

```
static const struct file_operations evdev_fops = {
    .owner      = THIS_MODULE,
    .read       = evdev_read,
    .write      = evdev_write,
    .poll       = evdev_poll,
    .open       = evdev_open,
    .release    = evdev_release,
    .unlocked_ioctl = evdev_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl   = evdev_ioctl_compat,
#endif
    .fasync     = evdev_fasync,
    .flush      = evdev_flush
};
```

在驱动程序中我们会调用 input_report_abs 等函数：

```
static inline void input_report_abs(struct input_dev *dev, unsigned int code, int value)
{
    input_event(dev, EV_ABS, code, value);
}
```

跟踪 input_event 如下：

```
void input_event(struct input_dev *dev,
         unsigned int type, unsigned int code, int value)
{
    unsigned long flags;

    if (is_event_supported(type, dev->evbit, EV_MAX)) {

        spin_lock_irqsave(&dev->event_lock, flags);
        /*利用输入值调正随机数产生器*/
        add_input_randomness(type, code, value);
        input_handle_event(dev, type, code, value);
        spin_unlock_irqrestore(&dev->event_lock, flags);
    }
}
```

跟踪 input_handle_event 如下：

```
static void input_handle_event(struct input_dev *dev,
                   unsigned int type, unsigned int code, int value)
{
    int disposition = INPUT_IGNORE_EVENT;

    switch (type) {
    。。。。。。。。。。。。。。。。。
    if (disposition != INPUT_IGNORE_EVENT && type != EV_SYN)
        dev->sync = 0;

    if ((disposition & INPUT_PASS_TO_DEVICE) && dev->event)
        dev->event(dev, type, code, value);

    if (disposition & INPUT_PASS_TO_HANDLERS)
        input_pass_event(dev, type, code, value);
}
```

如果该事件需要 input device 来完成，就会将 disposition 设置成 INPUT_PASS_TO_DEVICE，如果需要 input handler 来完成，就会将 disposition 设置成 INPUT_PASS_TO_DEVICE，如果需要两者都参与，则将 disposition 设置成 INPUT_PASS_TO_ALL。

跟踪 input_pass_event 如下：

```
static void input_pass_event(struct input_dev *dev,
                 unsigned int type, unsigned int code, int value)
{
    struct input_handle *handle;

    rcu_read_lock();
    /**/
    handle = rcu_dereference(dev->grab);
    if (handle)
        /*如果 input_dev 的 grab 指向了一个 handle，就用这个 handle 关联的 handler 的 event，否则遍历整个挂在 input_dev 的 h_list 上的 handle 关联的 handler*/
        handle->handler->event(handle, type, code, value);
    else
        list_for_each_entry_rcu(handle, &dev->h_list, d_node)
            if (handle->open)
                handle->handler->event(handle,
                            type, code, value);
    rcu_read_unlock();
}
```

比如下边的 evdev_handler 的 evdev_event：

```
static void evdev_event(struct input_handle *handle,
            unsigned int type, unsigned int code, int value)
{
    struct evdev *evdev = handle->private;
    struct evdev_client *client;
    struct input_event event;

    do_gettimeofday(&event.time);
    event.type = type;
    event.code = code;
    event.value = value;

    rcu_read_lock();
    client = rcu_dereference(evdev->grab);
    if (client)
    /*如果 evdev->grab 指向一个当前使用的 client 就将 event 放到这个 client 的 buffer 中，否则放到整个 client_list 上的 client 的链表中*/
        evdev_pass_event(client, &event);
    else
        list_for_each_entry_rcu(client, &evdev->client_list, node)
            evdev_pass_event(client, &event);

    rcu_read_unlock();

    wake_up_interruptible(&evdev->wait);
}
```

```
static void evdev_pass_event(struct evdev_client *client,
                 struct input_event *event)
{
    /*
     * Interrupts are disabled, just acquire the lock
     */
    spin_lock(&client->buffer_lock);
    /*将 event 装入 client 的 buffer 中，buffer 是一个环形缓存区*/
    client->buffer[client->head++] = *event;
    client->head &= EVDEV_BUFFER_SIZE - 1;
    spin_unlock(&client->buffer_lock);

    kill_fasync(&client->fasync, SIGIO, POLL_IN);
}
```

这里总结一下事件的传递过程：首先在驱动层中，调用 inport_report_abs，然后他调用了 input core 层的 input_event，input_event 调用了 input_handle_event 对事件进行分派，调用 input_pass_event，在这里他会把事件传递给具体的 handler 层，然后在相应 handler 的 event 处理函数中，封装一个 event，然后把它投入 evdev 的那个 client_list 上的 client 的事件 buffer 中，等待用户空间来读取。

当用户空间打开设备节点/dev/input/event0~/dev/input/event4 的时候，会使用 input_fops 中的 input_open_file()函数，input_open_file()->evdev_open()(如果 handler 是 evdev 的话)->evdev_open_device()->input_open_device()->dev->open()。也就是 struct file_operations input_fops 提供了通用接口，最终会调用具体 input_dev 的 open 函数。下边看一下用户程序打开文件时的过程，首先调用了 input_open_file:

```
static int input_open_file(struct inode *inode, struct file *file)
{
    struct input_handler *handler;
    const struct file_operations *old_fops, *new_fops = NULL;
    int err;

    lock_kernel();
    /* No load-on-demand here? */
    /*因为 32 个 input_dev 公共一个 handler 所以低 5 位应该是相同的*/
    handler = input_table[iminor(inode) >> 5];
    if (!handler || !(new_fops = fops_get(handler->fops))) {
        err = -ENODEV;
        goto out;
    }

    /*
     * That's _really_ odd. Usually NULL ->open means "nothing special",
     * not "no device". Oh, well...
     */
    if (!new_fops->open) {
        fops_put(new_fops);
        err = -ENODEV;
        goto out;
    }
    /*保存以前的 fops，使用相应的 handler 的 fops*/
    old_fops = file->f_op;
    file->f_op = new_fops;

    err = new_fops->open(inode, file);

    if (err) {
        fops_put(file->f_op);
        file->f_op = fops_get(old_fops);
    }
    fops_put(old_fops);
out:
    unlock_kernel();
    return err;
}
```

这里还是假设 handler 是 evdev_handler。

```
static int evdev_open(struct inode *inode, struct file *file)
{
    struct evdev *evdev;
    struct evdev_client *client;
    /*因为次设备号是从 EVDEV_MINOR_BASE 开始的*/
    int i = iminor(inode) - EVDEV_MINOR_BASE;
    int error;

    if (i >= EVDEV_MINORS)
        return -ENODEV;

    error = mutex_lock_interruptible(&evdev_table_mutex);
    if (error)
        return error;
    /*evdev_table 一共可容纳 32 个成员，找到次设备号对应的那个*/
    evdev = evdev_table[i];
    if (evdev)
        get_device(&evdev->dev);
    mutex_unlock(&evdev_table_mutex);

    if (!evdev)
        return -ENODEV;
    /*打开的时候创建一个 client*/
    client = kzalloc(sizeof(struct evdev_client), GFP_KERNEL);
    if (!client) {
        error = -ENOMEM;
        goto err_put_evdev;
    }

    spin_lock_init(&client->buffer_lock);
    /*下边两句的作用就是将 evdev 和 client 绑定到一起*/
    client->evdev = evdev;
    evdev_attach_client(evdev, client);

    error = evdev_open_device(evdev);
    if (error)
        goto err_free_client;
    /*将 file->private_data 指向刚刚建的 client，后边会用到的*/
    file->private_data = client;
    return 0;

err_free_client:
    evdev_detach_client(evdev, client);
    kfree(client);
err_put_evdev:
    put_device(&evdev->dev);
    return error;
}
```

```
static int evdev_open_device(struct evdev *evdev)
{
    int retval;

    retval = mutex_lock_interruptible(&evdev->mutex);
    if (retval)
        return retval;
    /*如果设备不存在，返回错误*/
    if (!evdev->exist)
        retval = -ENODEV;
    /*如果是被第一次打开，则调用 input_open_device*/
    else if (!evdev->open++) {
        retval = input_open_device(&evdev->handle);
        if (retval)
            evdev->open--;
    }

    mutex_unlock(&evdev->mutex);
    return retval;
}
```

```
int input_open_device(struct input_handle *handle)
{
    struct input_dev *dev = handle->dev;
    int retval;

    retval = mutex_lock_interruptible(&dev->mutex);
    if (retval)
        return retval;

    if (dev->going_away) {
        retval = -ENODEV;
        goto out;
    }

    handle->open++;

    if (!dev->users++ && dev->open)
        retval = dev->open(dev);

    if (retval) {
        dev->users--;
        if (!--handle->open) {
            /*
             * Make sure we are not delivering any more events
             * through this handle
             */
            synchronize_rcu();
        }
    }

out:
    mutex_unlock(&dev->mutex);
    return retval;
}
```

下面是用户进程读取event的底层实现：

```
static ssize_t evdev_read(struct file *file, char __user *buffer,
              size_t count, loff_t *ppos)
{
    /*这个就是刚才在open函数中*/
    struct evdev_client *client = file->private_data;
    struct evdev *evdev = client->evdev;
    struct input_event event;
    int retval;

    if (count < input_event_size())
        return -EINVAL;
    /*如果client的环形缓冲区中没有数据并且是非阻塞的，那么返回-EAGAIN，也就是try again*/
    if (client->head == client->tail && evdev->exist &&
        (file->f_flags & O_NONBLOCK))
        return -EAGAIN;
    /*如果没有数据，并且是阻塞的，则在等待队列上等待吧*/
    retval = wait_event_interruptible(evdev->wait,
        client->head != client->tail || !evdev->exist);
    if (retval)
        return retval;

    if (!evdev->exist)
        return -ENODEV;
    /*如果获得了数据则取出来，调用evdev_fetch_next_event*/
    while (retval + input_event_size() <= count &&
           evdev_fetch_next_event(client, &event)) {
        /*input_event_to_user调用copy_to_user传入用户程序中，这样读取完成*/
        if (input_event_to_user(buffer + retval, &event))
            return -EFAULT;

        retval += input_event_size();
    }

    return retval;
}
```

```
static int evdev_fetch_next_event(struct evdev_client *client,
                  struct input_event *event)
{
    int have_event;

    spin_lock_irq(&client->buffer_lock);
    /*先判断一下是否有数据*/
    have_event = client->head != client->tail;
    /*如果有就从环形缓冲区的取出来，记得是从 head 存储，tail 取出*/
    if (have_event) {
        *event = client->buffer[client->tail++];
        client->tail &= EVDEV_BUFFER_SIZE - 1;
    }

    spin_unlock_irq(&client->buffer_lock);

    return have_event;
}
```

```
int input_event_to_user(char __user *buffer,
            const struct input_event *event)
{
    /*如果设置了标志 INPUT_COMPAT_TEST 就将事件 event 包装成结构体 compat_event*/
    if (INPUT_COMPAT_TEST) {
        struct input_event_compat compat_event;

        compat_event.time.tv_sec = event->time.tv_sec;
        compat_event.time.tv_usec = event->time.tv_usec;
        compat_event.type = event->type;
        compat_event.code = event->code;
        compat_event.value = event->value;
        /*将包装成的 compat_event 拷贝到用户空间*/
        if (copy_to_user(buffer, &compat_event,
                 sizeof(struct input_event_compat)))
            return -EFAULT;

    } else {
        /*否则，将 event 拷贝到用户空间*/
        if (copy_to_user(buffer, event, sizeof(struct input_event)))
            return -EFAULT;
    }

    return 0;
}
```

这里总结一下：如果两个进程打开同一个文件，每个进程在打开时都会生成一个 evdev_client，evdev_client 被挂在 evdev 的 client_list，在 handle 收到一个事件的时候，会把事件 copy 到挂在 client_list 上的所有 evdev_client 的 buffer 中。这样所有打开同一个设备的进程都会收到这个消息而唤醒。