genirq - generic irq

uio device的interrupt handler是drivers/uio/uio.c/uio_interrupt()

```
/**
 * uio_interrupt - hardware interrupt handler
 * @irq: IRQ number, can be UIO_IRQ_CYCLIC for cyclic timer
 * @dev_id: Pointer to the devices uio_device structure
 */
static irqreturn_t uio_interrupt(int irq, void *dev_id)
{
        struct uio_device *idev = (struct uio_device *)dev_id;
        irqreturn_t ret = idev->info->handler(irq, idev->info);        ①

        if (ret == IRQ_HANDLED)           ②
                uio_event_notify(idev->info);

        return ret;
}
```

对于Generic IRQ uio device，ui_pdrv_genirq driver的都会在binding某个uio device时注册该device的interrupt handler.

driver framwork binding uio device with ui_pdrv_genirq driver driver

--> uio_pdrv_genirq_probe()

--> uio_register_device()   // create uio_device

--> devm_request_irq()   // binding uio_interrupt() to the irq of uio device

即所有uio device的interrupt的1st interrupt handler都是uio.c/uio_interrupt().

①

irqreturn_t ret = idev->info->handler(irq, idev->info);

idev->info->handler()这里的struct uio_info的.handler是相对与不同类型的uio_device的driver的。比

如

Generic IRQ型的uio device的driver的uio_pdrv_genirq_handler()是2nd interrupt handler。

in uio_pdrv_genirq.c/uio_pdrv_genirq_probe()

```
	/* This driver requires no hardware specific kernel code to handle
	 * interrupts. Instead, the interrupt handler simply disables the
	 * interrupt in the interrupt controller. User space is responsible
	 * for performing hardware specific acknowledge and re-enabling of
	 * the interrupt in the interrupt controller.
	 *
	 * Interrupt sharing is not supported.
	 */

	uioinfo->handler = uio_pdrv_genirq_handler;
	uioinfo->irqcontrol = uio_pdrv_genirq_irqcontrol;
	uioinfo->open = uio_pdrv_genirq_open;
	uioinfo->release = uio_pdrv_genirq_release;
	uioinfo->priv = priv;
```

```
static irqreturn_t uio_pdrv_genirq_handler(int irq, struct uio_info *dev_info)
{
	struct uio_pdrv_genirq_platdata *priv = dev_info->priv;

	/* Just disable the interrupt in the interrupt controller, and
	 * remember the state so we can allow user space to enable it later.
	 */

	spin_lock(&priv->lock);
	if (!__test_and_set_bit(UIO_IRQ_DISABLED, &priv->flags))
		disable_irq_nosync(irq);
	spin_unlock(&priv->lock);

	return IRQ_HANDLED;
}
```

对Generic IRQ uio device的处理是如此简单，就是disable该device进一步产生interrupt。

然后返回IRQ_HANDLED，标记已经处理该interrupt.

②

如果返回IRQ_HANDLED，则uio_event_notify(),显然是要通知更上层的interrupt handler来处理

in uio.c

```
/**
 * uio_event_notify - trigger an interrupt event
 * @info: UIO device capabilities
 */
void uio_event_notify(struct uio_info *info)
{
        struct uio_device *idev = info->uio_dev;

        atomic_inc(&idev->event);                    ③
        wake_up_interruptible(&idev->wait);    ④
        kill_fasync(&idev->async_queue, SIGIO, POLL_IN);
}
```

③

来一次interrupt就递增一次

④

唤醒等待在wait queue上的process。该process应该就是用户态真正处理该interrupt的application。

```
struct uio_device {
        struct module           *owner;
        struct device           *dev;
        int                     minor;
        atomic_t                event;
        struct fasync_struct    *async_queue;
        wait_queue_head_t       wait;
        struct uio_info         *info;
        struct kobject          *map_dir;
        struct kobject          *portio_dir;
};
```

uio kernel driver部分与user mode driver就是通过这个wait queue来同步的。

真正能感知hardware interrupt的code在uio kernel driver中(uio and uio_pdrv_genirq) ,而真正处理该

interrupt的handler在user mode application中。

user mode interrupt handler wait在该wait queue上，等待着kernel部分的code来wakeup它。只要被唤醒，就表示有interrupt来了。

如果不考虑时间延迟，确实是非常漂亮的方式，因为user mode programming毕竟要远远比lernel mode programming方便多了！

---

in uio.c

```
static unsigned int uio_poll(struct file *filep, poll_table *wait)
{
        struct uio_listener *listener = filep->private_data;
        struct uio_device *idev = listener->dev;

        if (!idev->info->irq)
                return -EIO;

        poll_wait(filep, &idev->wait, wait);        ①
        if (listener->event_count != atomic_read(&idev->event))    ②
                return POLLIN | POLLRDNORM;        ③
        return 0;
}
```

```
static const struct file_operations uio_fops = {
        .owner          = THIS_MODULE,
        .open           = uio_open,
        .release        = uio_release,
        .read           = uio_read,
        .write          = uio_write,
        .mmap           = uio_mmap,
.poll           = uio_poll,
        .fasync         = uio_fasync,
        .llseek         = noop_llseek,
};
```

uio_poll()就是用于response poll / epoll system call的handler。

①

user mode interrupt handler通过调用poll / epoll system call而等待interrupt产生，也就是运行的这里，使得该process wait在该uio device的wait queue上。

②

当uio device产生interrupt，该user mode handler的process被wakeup后，通过比较来确定是否确实有interrupt产生

③

POLLIN There is data to read.

POLLRDNORM Equivalent to POLLIN.

A value of 0 indicates that the call timed out and no file descriptors were ready.

---

user mode interrupt handler

in driver/pip/pip-app/uio_lib.c

```c
void uio_lib_init(void)
{
    int px_status;

    if (uio_epfd == -1)
    {
        DBG_DEBUG("%s\n", __func__);
        uio_epfd = epoll_create(1);
        REL_XASSERT(uio_epfd != -1, errno);

        px_status = posix_create_thread( &uio_thd_id,
                                 UIOIntThread,
                                 0,
                                 "UIO_interrupt_thread",
                                 UIOStack,
                                 UIO_STACK_SIZE,
                                 (POSIX_THR_PRI_ISR));
        REL_XASSERT( px_status==0, px_status );
    }
}
```

```
#define MAX_EVENTS 20
void *UIOIntThread(void *unused)
{
    struct epoll_event events[MAX_EVENTS];

    while (1)
    {
        DBG_DEBUG("Waiting for UIO interrupt event\n");
        int nfds = epoll_wait(uio_epfd, events, MAX_EVENTS, -1);            ①
        if (nfds < 0)
        {
            if (errno != EINTR)
            {
                DPRINTF((DBG_LOUD|DBG_OUTPUT), ("UIOLIB: epoll_wait failed - errno = %d\n", errno));
                posix_sleep_ms(500);
            }
        }
        else
        {
            int i;
            for (i = 0; i < nfds; i++)
            {
                uio_dev_t *dev = events[i].data.ptr;
                int32_t int_count;

                ASSERT(dev);

                DBG_DEBUG("Reading event count from device %s\n", dev->name);
                if (read(dev->fd, &int_count, 4) == 4)              ②
                {
                    if (dev->handler)   ③
                    {
                        DBG_DEBUG("Calling handler count %d for device %s\n", int_count, dev->name);
                        dev->handler(int_count, dev->context);
                    }

                    // re-enable interrupts
                    uio_int_enable(dev);                ④
                }
            }
        }
    }
    return 0;
}
```

①

user mode interrupt thrad将wait在这一行，一直等到有interrupt occur，然后由uio kernel driver wakeup该thread

When successful, epoll_wait() returns the number of file descriptors ready for the requested I/O, or zero if no file descriptor became ready,When an error occurs, epoll_wait() returns -1

②

对/dev/uio/uioX uio device read operation，会调用到uio.c/uio_read()，返回的是该设备的interrupt count

③

user mode interrupt handler

④

记得在kernel mode的2nd interrupt handler(uio_pdrv_genirq_handler())中，该device的interrupt被disable了。所以在次要enable该interrupt。