Mouse Drivers

Alan Cox

alan@redhat.com

Mouse Drivers

by Alan Cox

Table of Contents

List of Tables

Chapter 1. Introduction

Earlier publication: Parts of this document first appeared in Linux Magazine under a ninety day exclusivity.

Mice are conceptually one of the simplest device interfaces in the Linux operating system. Not all mice are handled by the kernel. Instead there is a two layer abstraction.

The kernel mouse drivers and userspace drivers for the serial mice are all managed by a system daemon called gpm - the general purpose mouse driver. gpm handles cutting and pasting on the text consoles. It provides a general library for mouse-aware applications and it handles the sharing of mouse services with the X Window System user interface.

Sometimes a mouse speaks a sufficiently convoluted protocol that the protocol is handled by Gpm itself. Most of the mouse drivers follow a common interface called the bus mouse protocol.

Each read from a bus mouse interface device returns a block of data. The first three bytes of each read are defined as follows:

Table 1-1. Mouse Data Encoding

An application can choose to read more than 3 bytes. The rest of the bytes will be zero, or may optionally return some additional device-specific information.

The position values are truncated if they exceed the 8bit range (that is -127 = delta = 127). While the value -128 does fit into a byte is not allowed.

The buttons are numbered left to right as 0, 1, 2, 3.. and each button sets the relevant bit. So a user pressing the left and right button of a three button mouse will set bits 0 and 2.

All mice are required to support the `poll` operation. Indeed pretty much every user of a mouse device uses `poll` to wait for mouse events to occur.

Finally the mice support asynchronous I/O. This is a topic we have not yet covered but which I will explain after looking at a simple mouse driver.

Chapter 2. A simple mouse driver

First we will need the set up functions for our mouse device. To keep this simple our imaginary mouse device has three I/O ports fixed at I/O address 0x300 and always lives on interrupt 5. The ports will be the X position, the Y position and the buttons in that order.

```
#define OURMOUSE_BASE        0x300

static struct miscdevice our_mouse = {
        OURMOUSE_MINOR, "ourmouse", our_mouse_fops
};

__init ourmouse_init(void)
{

        if(check_region(OURMOUSE_BASE, 3))
                return -ENODEV;
        request_region(OURMOUSE_BASE, 3, "ourmouse");

        misc_register(our_mouse);
        return 0;
}
```

The miscdevice is new here. Linux normally parcels devices out by major number, and each device has 256 units. For things like mice this is extremely wasteful so a device exists which is used to accumulate all the odd individual devices that computers tend to have.

Minor numbers in this space are allocated by a central source, although you can look in the kernel `Documentation/devices.txt` file and pick a free one for development use. This kernel file also carries instructions for registering a device. This may change over time so it is a good idea to obtain a current copy of this file first.

Our code then is fairly simple. We check nobody else has taken our address space. Having done so we reserve it to ensure nobody stamps on our device while probing for other ISA bus devices. Such a probe might confuse our device.

Then we tell the misc driver that we wish to own a minor number. We also hand it our name (which is used in `/proc/misc`) and a set of file operations that are to be used. The file operations work exactly like the file operations you would register for a normal character device. The misc device itself is simply acting as a redirector for requests.

Next, in order to be able to use and test our code we need to add some module code to support it. This too is fairly simple:

```
#ifdef MODULE

int init_module(void)
{
        if(ourmouse_init()0)
                return -ENODEV:
        return 0;
}

void cleanup_module(void)
{
        misc_deregister(our_mouse);
        free_region(OURMOUSE_BASE, 3);
}


#endif
```

The module code provides the normal two functions. The `init_module` function is called when the module is loaded. In our case it simply calls the initialising function we wrote and returns an error if this fails. This ensures the module will only be loaded if it was successfully set up.

The `cleanup_module` function is called when the module is unloaded. We give the miscellaneous device entry back, and then free our I/O resources. If we didn't free the I/O resources then the next time the module loaded it would think someone else had its I/O space.

Once the `misc_deregister` has been called any attempts to open the mouse device will fail with the error ENODEV (No such device).

Next we need to fill in our file operations. A mouse doesn't need many of these. We need to provide open, release, read and poll. That makes for a nice simple structure:

```
struct file_operations our_mouse_fops = {
        owner: THIS_MODULE,             /* Automatic usage management */
        read:  read_mouse,              /* You can read a mouse */
        write: write_mouse,             /* This won't do a lot */
        poll:  poll_mouse,              /* Poll */
        open:  open_mouse,              /* Called on open */
        release: close_mouse,           /* Called on close */
```

```
};
```

There is nothing particularly special needed here. We provide functions for all the relevant or required operations and little else. There is nothing stopping us providing an ioctl function for this mouse. Indeed if you have a configurable mouse it may be very appropriate to provide configuration interfaces via ioctl calls.

The syntax we use is not standard C as such. GCC provides the ability to initialise fields by name, and this generally makes the method table much easier to read than counting through NULL pointers and remembering the order by hand.

The owner field is used to manage the locking of module load an unloading. It is obviously important that a module is not unloaded while in use. When your device is opened the module specified by "owner" is locked. When it is finally released the module is unlocked.

The open and close routines need to manage enabling and disabling the interrupts for the mouse as well as stopping the mouse being unloaded when it is no longer required.

```
static int mouse_users = 0;               /* User count */
static int mouse_dx = 0;                  /* Position changes */
static int mouse_dy = 0;
static int mouse_event = 0;               /* Mouse has moved */

static int open_mouse(struct inode *inode, struct file *file)
{
        if(mouse_users++)
                return 0;

        if(request_irq(mouse_intr, OURMOUSE_IRQ, 0, "ourmouse", NULL))
        {
                mouse_users--;
                return -EBUSY;
        }
        mouse_dx = 0;
        mouse_dy = 0;
        mouse_event = 0;
        mouse_buttons = 0;
 return 0;
}
```

The open function has to do a small amount of housework. We keep a count of the number of times the mouse is open. This is because we do not want to request the

interrupt multiple times. If the mouse has at least one user then it is set up and we simply add to the user count and return 0 for success.

We grab the interrupt and thus start mouse interrupts. If the interrupt has been borrowed by some other driver then `request_irq` will fail and we will return an error. If we were capable of sharing an interrupt line we would specify `SA_SHIRQ` instead of `zero`. Provided that everyone claiming an interrupt sets this flag, they get to share the line. PCI can share interrupts, ISA normally however cannot.

We do the housekeeping. We make the current mouse position the starting point for accumulated changes and declare that nothing has happened since the mouse driver was opened.

The release function needs to unwind all these:

```
static int close_mouse(struct inode *inode, struct file *file)
{
        if(--mouse_users)
                return 0;
        free_irq(OURMOUSE_IRQ, NULL);
        return 0;
}
```

We count off a user and provided that there are still other users need take no further action. The last person closing the mouse causes us to free up the interrupt. This stops interrupts from the mouse from using our CPU time, and ensures that the mouse can now be unloaded.

We can fill in the write handler at this point as the write function for our mouse simply declines to allow writes:

```
static ssize_t write_mouse(struct file *file, const char *buffer, size_t
                                count, loff_t *ppos)
{
        return -EINVAL;
}
```

This is pretty much self-explanatory. Whenever you write you get told it was an invalid function.

To make the poll and read functions work we have to consider how we handle the mouse interrupt.

```
static struct wait_queue *mouse_wait;
```

```
static spinlock_t mouse_lock = SPIN_LOCK_UNLOCKED;

static void ourmouse_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        char delta_x;
        char delta_y;
        unsigned char new_buttons;

        delta_x = inb(OURMOUSE_BASE);
        delta_y = inb(OURMOUSE_BASE+1);
        new_buttons = inb(OURMOUSE_BASE+2);

        if(delta_x || delta_y || new_buttons != mouse_buttons)
        {
                /* Something happened */

                spin_lock(mouse_lock);
                mouse_event = 1;
                mouse_dx += delta_x;
                mouse_dy += delta_y;
                mouse_buttons = new_buttons;
                spin_unlock(mouse_lock);

                wake_up_interruptible(mouse_wait);
        }
}
```

The interrupt handler reads the mouse status. The next thing we do is to check whether something has changed. If the mouse was smart it would only interrupt us if something had changed, but let’s assume our mouse is stupid as most mice actually tend to be.

If the mouse has changed we need to update the status variables. What we don’t want is the mouse functions reading these variables to read them during a change. We add a spinlock that protects these variables while we play with them.

If a change has occurred we also need to wake sleeping processes, so we add a wakeup call and a wait_queue to use when we wish to await a mouse event.

Now we have the wait queue we can implement the poll function for the mouse relatively easily:

```
static unsigned int mouse_poll(struct file *file, poll_table *wait)
{
        poll_wait(file, mouse_wait, wait);
```

```
        if(mouse_event)
                return POLLIN | POLLRDNORM;
        return 0;
}
```

This is fairly standard poll code. First we add the wait queue to the list of queues we want to monitor for an event. Secondly we check if an event has occurred. We only have one kind of event - the `mouse_event` flag tells us that something happened. We know that this something can only be mouse data. We return the flags indicating input and normal reading will succeed.

You may be wondering what happens if the function returns saying 'no event yet'. In this case the wake up from the wait queue we added to the poll table will cause the function to be called again. Eventually we will be woken up and have an event ready. At this point the `poll` call will exit back to the user.

After the poll completes the user will want to read the data. We now need to think about how our `mouse_read` function will work:

```
static ssize_t mouse_read(struct file *file, char *buffer,
                size_t count, loff_t *pos)
{
        int dx, dy;
        unsigned char button;
        unsigned long flags;
        int n;

        if(count3)
                return -EINVAL;

        /*
          *        Wait for an event
         */

        while(!mouse_event)
        {
                if(file-f_flagsO_NDELAY)
                        return -EAGAIN;
                interruptible_sleep_on(mouse_wait);
                if(signal_pending(current))
                        return -ERESTARTSYS;
        }
```

We start by validating that the user is reading enough data. We could handle partial reads if we wanted but it isn't terribly useful and the mouse drivers don't bother to try.

Next we wait for an event to occur. The loop is fairly standard event waiting in Linux. Having checked that the event has not yet occurred, we then check if an event is pending and if not we need to sleep.

A user process can set the `O_NDELAY` flag on a file to indicate that it wishes to be told immediately if no event is pending. We check this and give the appropriate error if so.

Next we sleep until the mouse or a signal awakens us. A signal will awaken us as we have used `wakeup_interruptible`. This is important as it means a user can kill processes waiting for the mouse - clearly a desirable property. If we are interrupted we exit the call and the kernel will then process signals and maybe restart the call again - from the beginning.

This code contains a classic Linux bug. All will be revealed later in this article as well as explanations for how to avoid it.

```
        /* Grab the event */

        spinlock_irqsave(mouse_lock, flags);

        dx = mouse_dx;
        dy = mouse_dy;
        button = mouse_buttons;

        if(dx=-127)
                dx=-127;
        if(dx=127)
                dx=127;
        if(dy=-127)
                dy=-127;
        if(dy=127)
                dy=127;

        mouse_dx -= dx;
        mouse_dy -= dy;

        if(mouse_dx == 0  mouse_dy == 0)
                mouse_event = 0;

        spin_unlock_irqrestore(mouse_lock, flags);
```

This is the next stage. Having established that there is an event going, we capture it. To be sure that the event is not being updated as we capture it we also take the spinlock and thus prevent parallel updates. Note here we use `spinlock_irqsave`. We need to disable interrupts on the local processor otherwise bad things will happen.

What will occur is that we take the spinlock. While we hold the lock an interrupt will occur. At this point our interrupt handler will try and take the spinlock. It will sit in a loop waiting for the read routine to release the lock. However because we are sitting in a loop in the interrupt handler we will never release the lock. The machine hangs and the user gets upset.

By blocking the interrupt on this processor we ensure that the lock holder will always give the lock back without deadlocking.

There is a little cleverness in the reporting mechanism too. We can only report a move of 127 per read. We don't however want to lose information by throwing away further movement. Instead we keep returning as much information as possible. Each time we return a report we remove the amount from the pending movement in `mouse_dx` and `mouse_dy`. Eventually when these counts hit zero we clear the `mouse_event` flag as there is nothing else left to report.

```
        if(put_user(button|0x80, buffer))
                return -EFAULT;
        if(put_user((char)dx, buffer+1))
                return -EFAULT;
        if(put_user((char)dy, buffer+2))
                return -EFAULT;

        for(n=3; n < count; n++)
                if(put_user(0x00, buffer+n))
                        return -EFAULT;

        return count;
}
```

Finally we must put the results in the user supplied buffer. We cannot do this while holding the lock as a write to user memory may sleep. For example the user memory may be residing on disk at this instant. Thus we did our computation beforehand and now copy the data. Each `put_user call` is filling in one byte of the buffer. If it returns an error we inform the program that it passed us an invalid buffer and abort.

Having written the data we blank the rest of the buffer that was read and report the read as being successful.

Chapter 3. Debugging the mouse driver

We now have an almost perfectly usable mouse driver. If you were to actually try and use it however you would eventually find a couple of problems with it. A few programs will also not work with as it does not yet support asynchronous I/O.

First let us look at the bugs. The most obvious one isn't really a driver bug but a failure to consider the consequences. Imagine you bumped the mouse hard by accident and sent it skittering across the desk. The mouse interrupt routine will add up all that movement and report it in steps of 127 until it has reported all of it. Clearly there is a point beyond which mouse movement isn't worth reporting. We need to add this as a limit to the interrupt handler:

```
static void ourmouse_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        char delta_x;
        char delta_y;
        unsigned char new_buttons;

        delta_x = inb(OURMOUSE_BASE);
        delta_y = inb(OURMOUSE_BASE+1);
        new_buttons = inb(OURMOUSE_BASE+2);

        if(delta_x || delta_y || new_buttons != mouse_buttons)
        {
                /* Something happened */

                spin_lock(mouse_lock);
                mouse_event = 1;
                mouse_dx += delta_x;
                mouse_dy += delta_y;

                if(mouse_dx  -4096)
                        mouse_dx = -4096;
                if(mouse_dx  4096)
                        mouse_dx = 4096;

                if(mouse_dy  -4096)
                        mouse_dy = -4096;
                if(mouse_dy  4096)
                        mouse_dy = 4096;

                mouse_buttons = new_buttons;
```

```
                spin_unlock(mouse_lock);

                wake_up_interruptible(mouse_wait);
        }
}
```

By adding these checks we limit the range of accumulated movement to something sensible.

The second bug is a bit more subtle, and that is perhaps why this is such a common mistake. Remember, I said the waiting loop for the read handler had a bug in it. Think about what happens when we execute:

```
while(!mouse_event)
{
```

and an interrupt occurs at this point here. This causes a mouse movement and wakes up the queue.

```
                interruptible_sleep_on(mouse_wait);
```

Now we sleep on the queue. We missed the wake up and the application will not see an event until the next mouse event occurs. This will lead to just the odd instance when a mouse button gets delayed. The consequences to the user will probably be almost undetectable with a mouse driver. With other drivers this bug could be a lot more severe.

There are two ways to solve this. The first is to disable interrupts during the testing and the sleep. This works because when a task sleeps it ceases to disable interrupts, and when it resumes it disables them again. Our code thus becomes:

```
save_flags(flags);
cli();

while(!mouse_event)
{
        if(file-f_flagsO_NDELAY)
        {
                restore_flags(flags);
                return -EAGAIN;
        }
        interruptible_sleep_on(mouse_wait);
```

```
        if(signal_pending(current))
        {
                restore_flags(flags);
                return -ERESTARTSYS;
        }
}
restore_flags(flags);
```

This is the sledgehammer approach. It works but it means we spend a lot more time turning interrupts on and off. It also affects interrupts globally and has bad properties on multiprocessor machines where turning interrupts off globally is not a simple operation, but instead involves kicking each processor, waiting for them to disable interrupts and reply.

The real problem is the race between the event testing and the sleeping. We can avoid that by using the scheduling functions more directly. Indeed this is the way they generally should be used for an interrupt.

```
struct wait_queue wait = { current, NULL };

add_wait_queue(mouse_wait, wait);
set_current_state(TASK_INTERRUPTIBLE);

while(!mouse_event)
{
        if(file-f_flagsO_NDELAY)
        {
                remove_wait_queue(mouse_wait, wait);
                set_current_state(TASK_RUNNING);
                return -EWOULDBLOCK;
        }
        if(signal_pending(current))
        {
                remove_wait_queue(mouse_wait, wait);
                current-state = TASK_RUNNING;
                return -ERESTARTSYS;
        }
        schedule();
        set_current_state(TASK_INTERRUPTIBLE);
}

remove_wait_wait(mouse_wait, wait);
set_current_state(TASK_RUNNING);
```

At first sight this probably looks like deep magic. To understand how this works you need to understand how scheduling and events work on Linux. Having a good grasp of this is one of the keys to writing clean efficient device drivers.

`add_wait_queue` does what its name suggests. It adds an entry to the `mouse_wait` list. The entry in this case is the entry for our current process (`current` is the current task pointer).

So we start by adding an entry for ourself onto the `mouse_wait` list. This does not put us to sleep however. We are merely tagged onto the list.

Next we set our status to `TASK_INTERRUPTIBLE`. Again this does not mean we are now asleep. This flag says what should happen next time the process sleeps. `TASK_INTERRUPTIBLE` says that the process should not be rescheduled. It will run from now until it sleeps and then will need to be woken up.

The `wakeup_interruptible` call in the interrupt handler can now be explained in more detail. This function is also very simple. It goes along the list of processes on the queue it is given and any that are marked as `TASK_INTERRUPTIBLE` it changes to `TASK_RUNNING` and tells the kernel that new processes are runnable.

Behind all the wrappers in the original code what is happening is this

1. We add ourself to the mouse wait queue
2. We mark ourself as sleeping
3. We ask the kernel to schedule tasks again
4. The kernel sees we are asleep and schedules someone else.
5. The mouse interrupt sets our state to `TASK_RUNNING` and makes a note that the kernel should reschedule tasks
6. The kernel sees we are running again and continues our execution

This is why the apparent magic works. Because we mark ourself as `TASK_INTERRUPTIBLE` and as we add ourselves to the queue before we check if there are events pending, the race condition is removed.

Now if an interrupt occurs after we check the queue status and before we call the `schedule` function in order to sleep, things work out. Instead of missing an event, we are set back to `TASK_RUNNING` by the mouse interrupt. We still call `schedule` but it will continue running our task. We go back around the loop and this time there may be an event.

There will not always be an event. Thus we set ourselves back to `TASK_INTERRUPTIBLE` before resuming the loop. Another process doing a read may already have cleared the event flag, and if so we will need to go back to sleep again. Eventually we will get our event and escape.

Finally when we exit the loop we remove ourselves from the `mouse_wait` queue as we are no longer interested in mouse events, and we set ourself back to `TASK_RUNNABLE` as we do not wish to go to sleep again just yet.

Note: