LDD3, Ch 6, 9, 10

ECEN 427

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Ch 6 IOCTL

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING


SALOON
IOCTL

Why ioctl?

- Most drivers need—in addition to the ability to read and write the device—the ability to perform various types of hardware control via the device driver. Most devices can perform operations beyond simple data transfers; user space must often be able to request, for example, that the device lock its door, eject its media, report error information, change a baud rate, or self destruct. These operations are usually supported via the ioctl method, which implements the system call by the same name.

ioctl System Call

- In user space, the ioctl system call has the following prototype:

```
int ioctl(int fd, unsigned long cmd, ...);
```

“The dots in the prototype represent not a variable number of arguments but a single optional argument, traditionally identified as char *argp. The dots are simply there to prevent type checking during compilation. The actual nature of the third argument depends on the specific control command being issued (the second argument). Some commands take no arguments, some take an integer value, and some take a pointer to other data. Using a pointer is the way to pass arbitrary data to the ioctl call; the device is then able to exchange any amount of data with user space

Issues with ioctl…

- “The unstructured nature of the ioctl call has caused it to fall out of favor among kernel developers. **Each ioctl command is, essentially, a separate, usually undocumented system call**,”

There is no way to audit these calls in any sort of comprehensive manner. It is also difficult to make the unstructured ioctl arguments work identically on all systems; for example, consider 64-bit systems with a userspace process running in 32-bit mode. As a result, there is strong pressure to implement miscellaneous control operations by just about any other means. Possible alternatives include embedding commands into the data stream or using virtual filesystems, either sysfs or driverspecific filesystems. However, the fact remains that ioctl is often the easiest and most straightforward choice for true device operations.

- The ioctl driver method has a prototype that differs somewhat from the user-space version:

```
int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd,
unsigned long arg);
```

- The inode and filp pointers are the values corresponding to the file descriptor fd passed on by the application and are the same parameters passed to the open method.
- The cmd argument is passed from the user unchanged,
- The optional arg argument is passed in the form of an unsigned long, regardless of whether it was given by the user as an integer or a pointer. If the invoking program doesn't pass a third argument, the arg value received by the driver operation is undefined.

ioctl Command Numbers

“As you might imagine, most ioctl implementations consist of a big switch statement that selects the correct behavior according to the cmd argument. Different commands have different numeric values, which are usually given symbolic names to simplify coding. The symbolic name is assigned by a preprocessor definition.”

“Before writing the code for ioctl, you need to choose the numbers that correspond to commands. The first instinct of many programmers is to choose a set of small numbers starting with 0 or 1 and going up from there. There are, however, good reasons for not doing things that way. The ioctl command numbers should be unique across the system in order to prevent errors caused by issuing the right command to the wrong device.”

The approved way to define ioctl command numbers uses four bitfields, which have the following meanings.

`type`

The magic number. Just choose one number (after consulting *ioctl-number.txt*) and use it throughout the driver. This field is eight bits wide (`_IOC_TYPEBITS`).

`number`

The ordinal (sequential) number. It's eight bits (`_IOC_NRBITS`) wide.

`direction`

The direction of data transfer, if the particular command involves a data transfer. The possible values are `_IOC_NONE` (no data transfer), `_IOC_READ`, `_IOC_WRITE`, and `_IOC_READ|_IOC_WRITE` (data is transferred both ways). Data transfer is seen from the application's point of view; `_IOC_READ` means reading *from* the device, so the driver must write to user space. Note that the field is a bit mask, so `_IOC_READ` and `_IOC_WRITE` can be extracted using a logical AND operation.

`size`

The size of user data involved. The width of this field is architecture dependent,

The header file <*asm/ioctl.h*>, which is included by <*linux/ioctl.h*>, defines macros that help set up the command numbers as follows: `_IO(type,nr)` (for a command that has no argument), `_IOR(type,nr,datatype)` (for reading data from the driver), `_IOW(type,nr,datatype)` (for writing data), and `_IOWR(type,nr,datatype)` (for bidirectional transfers). The type and number fields are passed as arguments, and the size field is derived by applying *sizeof* to the datatype argument.

```
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC,  1, int)
#define SCULL_IOCSQSET    _IOW(SCULL_IOC_MAGIC,  2, int)

#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC,   3)
#define SCULL_IOCTQSET    _IO(SCULL_IOC_MAGIC,   4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC,  5, int)
#define SCULL_IOCGQSET    _IOR(SCULL_IOC_MAGIC,  6, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC,   7)
#define SCULL_IOCQQSET    _IO(SCULL_IOC_MAGIC,   8)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
#define SCULL_IOCXQSET    _IOWR(SCULL_IOC_MAGIC,10, int)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC,  11)
#define SCULL_IOCHQSET    _IO(SCULL_IOC_MAGIC,  12)
```

ioctl and pointers

“When a pointer is used to refer to user space, we must ensure that the user address is valid. An attempt to access an unverified user-supplied pointer can lead to incorrect behavior, a kernel oops, system corruption, or security problems. It is the driver’s responsibility to make proper checks on every user-space address it uses and to return an error if it is invalid.”

In Chapter 3, we looked at the copy_from_user and copy_to_user functions, which can be used to safely move data to and from user space. Those functions can be used in ioctl methods as well, but ioctl calls often involve small data items that can be more efficiently manipulated through other means. To start, address verification (without transferring data) is implemented by the function access_ok, which is declared in <asm/uaccess.h>:

```
int access_ok(int type, const void *addr, unsigned long size);
```

Ch 9: Communicating With Hardware

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Memory vs Registers

What is the different between read/writing device registers and memory?

What issues could arise?

“The main difference between I/O registers and RAM is that I/O operations have side effects, while memory operations have none: the only effect of a memory write is storing a value to a location, and a memory read returns the last value written there. Because memory access speed is so critical to CPU performance, the no-side-effects case has been optimized in several ways: values are cached and read/write instructions are reordered.”

“The compiler can cache data values into CPU registers without writing them to memory, and even if it stores them, both write and read operations can operate on cache memory without ever reaching physical RAM. Reordering can also happen both at the compiler level and at the hardware level: often a sequence of instructions can be executed more quickly if it is run in an order different from that which appears in the program text, for example, to prevent interlocks in the RISC pipeline. On CISC processors, operations that take a significant amount of time can be executed concurrently with other, quicker ones.”

“These optimizations are transparent and benign when applied to conventional memory (at least on uniprocessor systems), but they can be fatal to correct I/O operations, because they interfere with those “side effects” that are the main reason why a driver accesses I/O registers. The processor cannot anticipate a situation in which some other process (running on a separate processor, or something happening inside an I/O controller) depends on the order of memory access. The compiler or the CPU may just try to outsmart you and reorder the operations you request; the result can be strange errors that are very difficult to debug. Therefore, a driver must ensure that no caching is performed and no read or write reordering takes place when accessing registers.”

“The problem with hardware caching is the easiest to face: the underlying hardware is already configured (either automatically or by Linux initialization code) to disable any hardware cache when accessing I/O regions (whether they are memory or port regions).”

"The solution to compiler optimization and hardware reordering is to place a memory barrier between operations that must be visible to the hardware (or to another processor) in a particular order. Linux provides four macros to cover all possible ordering needs:"

`void barrier(void)`

This function tells the compiler to insert a memory barrier but has no effect on the hardware. Compiled code stores to memory all values that are currently modified and resident in CPU registers, and rereads them later when they are needed. A call to *barrier* prevents compiler optimizations across the barrier but leaves the hardware free to do its own reordering.

```
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);
```

These functions insert hardware memory barriers in the compiled instruction flow; their actual instantiation is platform dependent. An *rmb* (read memory barrier) guarantees that any reads appearing before the barrier are completed prior to the execution of any subsequent read. *wmb* guarantees ordering in write operations, and the *mb* instruction guarantees both. Each of these functions is a superset of *barrier*.

Allocating device memory

“As you might expect, you should not go off and start pounding on I/O ports (or memory) without first ensuring that you have exclusive access to those ports. The kernel provides a registration interface that allows your driver to claim the ports/memory it needs. The core function in that interface is request_region/request_mem_region”

```
#include <linux/ioport.h>

struct resource *request_mem_region(unsigned long start, unsigned long
len, char *name);
```

This isn’t strictly necessary, but it should be done to “play nice” with other drivers.

ioremap

“Allocation of I/O memory is not the only required step before that memory may be accessed. You must also ensure that this I/O memory has been made accessible to the kernel. Getting at I/O memory is not just a matter of dereferencing a pointer; on many systems, I/O memory is not directly accessible in this way at all. So a mapping must be set up first.”

This is the role of the ioremap function.

> The functions are called according to the following definition:
>
> ```
> #include <asm/io.h>
> void *ioremap(unsigned long phys_addr, unsigned long size);
> void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
> void iounmap(void * addr);
> ```

Reading/Writing Device Memory

Remember, though, that the addresses returned from ioremap should not be dereferenced directly; instead, accessor functions provided by the kernel should be used.

On some platforms, you may get away with using the return value from ioremap as a pointer. Such use is not portable, and, increasingly, the kernel developers have been working to eliminate any such use. The proper way of getting at I/O memory is via a set of functions (defined via <asm/io.h>) provided for that purpose.

To read from I/O memory, use one of the following:

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

Ch 10: Interrupt Handling

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

“Although some devices can be controlled using nothing but their I/O regions, most real devices are a bit more complicated than that. Devices have to deal with the external world, which often includes things such as spinning disks, moving tape, wires to distant places, and so on. Much has to be done in a time frame that is different from, and far slower than, that of the processor. Since it is almost always undesirable to have the processor wait on external events, there must be a way for a device to let the processor know when something has happened. “

“That way, of course, is interrupts. An interrupt is simply a signal that the hardware can send when it wants the processor’s attention. Linux handles interrupts in much the same way that it handles signals in user space. For the most part, a driver need only register a handler for its device’s interrupts, and handle them properly when they arrive. Of course, underneath that simple picture there is some complexity; in particular, interrupt handlers are somewhat limited in the actions they can perform as a result of how they are run.”

- Register an interrupt handler

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                const char *dev_name,
                void *dev_id);
```

`unsigned int irq`
The interrupt number being requested.

`irqreturn_t (*handler)(int, void *, struct pt_regs *)`
The pointer to the handling function being installed.

`const char *dev_name`
The string passed to *request_irq* is used in */proc/interrupts* to show the owner of the interrupt (see the next section).

- The correct place to call request_irq is when the device is first opened, before the hardware is instructed to generate interrupts. The place to call free_irq is the last time the device is closed, after the hardware is told not to interrupt the processor any more.

The /proc Interface

Whenever a hardware interrupt reaches the processor, an internal counter is incremented, providing a way to check whether the device is working as expected. Reported interrupts are shown in */proc/interrupts*. The following snapshot was taken on a two-processor Pentium system:

```
root@montalcino:/bike/corbet/write/ldd3/src/short# m /proc/interrupts
           CPU0       CPU1
  0:    4848108         34    IO-APIC-edge  timer
  2:          0          0          XT-PIC  cascade
  8:          3          1    IO-APIC-edge  rtc
 10:       4335          1   IO-APIC-level  aic7xxx
 11:       8903          0   IO-APIC-level  uhci_hcd
 12:         49          1    IO-APIC-edge  i8042
NMI:          0          0
LOC:    4848187    4848186
ERR:          0
MIS:          0
```

ISR Arguments


```
static irqreturn_t sample_interrupt(int irq, void *dev_id, struct pt_regs
                                    *regs)
{
```

- The interrupt number (int irq) is useful as information you may print in your log messages, if any.
- The second argument, void *dev_id, is a sort of client data; a void * argument is passed to request_irq, and this same pointer is then passed backas an argument to the handler when the interrupt happens. You usually pass a pointer to your device data structure in dev_id, so a driver that manages several instances of the same device doesn't need any extra code in the interrupt handler to find out which device is in charge of the current interrupt event.
- The last argument, struct pt_regs *regs, is rarely used. It holds a snapshot of the processor's context before the processor entered interrupt code. The registers can be used for monitoring and debugging; they are not normally needed for regular device driver tasks.
- Interrupt handlers should return a value indicating whether there was actually an interrupt to handle. If the handler found that its device did, indeed, need attention, it should return IRQ_HANDLED