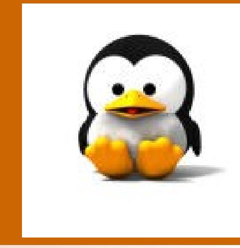
Linux Device Driver
(Enhanced Char Driver)

Amir Hossein Payberah
payberah@yahoo.com

Contents



- ioctl
- Seeking a Device
- Blocking I/O and non-blocking I/O

ioctl

- int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
- The cmd argument is passed from the user unchanged.
- The optional arg argument is passed in the form of an unsigned long.

ioctl

- Most ioctl implementations consist of a switch statement.
  - It selects the correct behavior according to the cmd argument.
- Different commands have different numeric values, which are usually given symbolic names to simplify coding.

ioctl commands

- Before writing the code for ioctl, you need to choose the numbers that correspond to commands.
  - Unfortunately, the simple choice of using small numbers starting from 1 and going up doesn't work well.

ioctl commands

- The command numbers should be unique across the system.
  - In order to prevent errors caused by issuing the right command to the wrong device.

Choosing ioctl commands

- ioctl command codes have been split up into several bitfields.
- The first versions of Linux used 16-bit numbers:
  - The top eight were the magic number associated with the device.
  - The bottom eight were a sequential number, unique within the device.

Choosing ioctl commands

- To choose ioctl numbers for your driver according to the new convention, you should first check include/asm/ioctl.h and Documentation/ioctl-number.txt.
- The header defines the bitfields you will be using.
  - type (magic number), ordinal number, direction of transfer, and size of argument.
- The ioctl-number.txt file lists the magic numbers used throughout the kernel.

Old ioctl commands

- The old way of choosing an ioctl number:
  - Authors chose a magic eight-bit number, such as ‘‘k’’ (hex 0x6b), and added an ordinal number, like this:

```
#define SCULL_IOCTL1 0x6b01
#define SCULL_IOCTL2 0x6b02
/* .... */
```

New ioctl commands

- Any new symbols are defined in <linux/ioctl.h>.
- Type
  - The magic number. Just choose one number (after consulting *ioc*tl-number.txt). 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).
- size
  - The size of user data involved. The width of this field is architecture dependent, and currently ranges from 8 to 14 bits. You can find its value for your specific architecture in the macro _IOC_SIZEBITS.

New ioctl commands

- The header file <asm/ioctl.h>, which is included by <linux/ioctl.h>, defines these macros:
  - _IO(type, number): for no data transferring command
  - _IOR(type, number, size)
  - _IOW(type, number, size)
  - _IOWR(type, number, size)
- The header also defines macros to decode the numbers:
  - _IOC_DIR(nr), _IOC_TYPE(nr), _IOC_NR(nr), and _IOC_SIZE(nr).

New ioctl commands sample

- #define SCULL_IOC_MAGIC 'k'
- #define SCULL_IOCRESET _IO(SCULL_IOC_MAGIC, 0)
- #define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC, 1, scull_quantum)
- #define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC, 5, scull_quantum)
- #define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, scull_quantum)

Predefined Commands

- kernel reserved some numbers for its use
  - Those for any file type including devices
  - Those only for regular files
  - Those specific to a file system type
- device driver is concerned with the first group
  - magic number “T”
  - FIOCLEX: set the close-on-exec flag
  - FIONCLEX: reset
  - FIOASYNC: set/reset synchronous write
  - FIONBIO: set/reset O_NONBLOCK of filp->f_flags

ioctl return value

- The implementation of ioctl is usually a switch statement based on the command number.
- But what should the default selection be when the command number doesn’t match a valid operation?
- Several kernel functions return -EINVAL (‘‘Invalid argument’’).
- The POSIX standard, however, states that if an inappropriate ioctl command has been issued, then -ENOTTY should be returned.

Using the ioctl Argument

- int access_ok(int type, const void *addr, unsigned long size);
  - The first argument should be either VERIFY_READ or VERIFY_WRITE.
  - The addr argument holds a user-space address.
  - The size is a byte count.

Sample

```
int err = 0, tmp;
if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC)
    return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR)
    return -ENOTTY;
if (_IOC_DIR(cmd) & _IOC_READ)
    err = !access_ok(VERIFY_WRITE, (void *)arg,
    _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
    err = !access_ok(VERIFY_READ, (void *)arg,
    _IOC_SIZE(cmd));
if (err)
    return -EFAULT;
```

Using the ioctl Argument

- put_user(datum, ptr) and __put_user(datum, ptr)
  - These macros write the datum to user space
  - They are relatively fast, and should be called instead of copy_to_user whenever single values are being transferred.
- get_user(local, ptr) and __get_user(local, ptr)
  - These macros are used to retrieve a single datum from user space.
- __put_user and __get_user should only be used if the memory region has already been verified with access_ok.

Sample

```
switch(cmd)
{
    case SCULL_IOCSQUANTUM:
        ret = __get_user(scull_quantum, (int *)arg);
        break;
    case SCULL_IOCGQUANTUM:
        ret = __put_user(scull_quantum, (int *)arg);
        break;
    default:
        return -ENOTTY;
}
```

User space sample

```
int quantum;

ioctl(fd,SCULL_IOCSQUANTUM, &quantum);

ioctl(fd,SCULL_IOCGQUANTUM, &quantum);
```

Contents

- ioctl
- Seeking a Device
- Blocking I/O and non-blocking I/O

llseek

- The llseek method implements the lseek and llseek system calls.
- We have already stated that if the llseek method is missing from the device’s operations:
  - The default implementation in the kernel performs seeks from the beginning of the file
  - And from the current position by modifying filp->f_pos, the current reading/writing position within the file.

llseek

- loff_t scull_llseek(struct file *filp, loff_t off, int whence);

Sample

```
loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
    Scull_Dev *dev = filp->private_data;
    loff_t newpos;
    switch(whence)
    {
        case 0: /* SEEK_SET */
                newpos = off;
                break;
        case 1: /* SEEK_CUR */
                newpos = filp->f_pos + off;
                break;
        case 2: /* SEEK_END */
                newpos = dev->size + off;
                break;
        default: /* can’t happen */
                return -EINVAL;
    }
    filp->f_pos = newpos;
    return newpos;
}
```

Contents

- ioctl
- Seeking a Device
- Blocking I/O and non-blocking I/O

Blocking I/O

- One problem that might arise with read is what to do when there’s no data yet, but we’re not at end-of-file.
  - The default answer is ‘‘go to sleep waiting for data’’.

Wait queue

- A wait queue is exactly a queue of processes that are waiting for an event.
- wait_queue_head_t my_queue;
- init_waitqueue_head (&my_queue);
- DECLARE_WAIT_QUEUE_HEAD (my_queue);
  - When a wait queue is declared statically, it is also possible to initialize the queue at compile time.

Sleeping

- sleep_on(wait_queue_head_t *queue);
- interruptible_sleep_on(wait_queue_head_t *queue);

- sleep_on_timeout(wait_queue_head_t *queue, long timeout);
- interruptible_sleep_on_timeout(wait_queue_head_t *queue, long timeout);

- wait_event(wait_queue_head_t queue, int condition);
- wait_event_interruptible(wait_queue_head_t queue, int condition);

Wake up

- wake_up(wait_queue_head_t *queue);
- wake_up_interruptible(wait_queue_head_t *queue);

- wake_up_sync(wait_queue_head_t *queue);
- wake_up_interruptible_sync(wait_queue_head_t *queue);

Sample

```
DECLARE_WAIT_QUEUE_HEAD(wq);

ssize_t sleepy_read (struct file *filp, char *buf, size_t count, loff_t
    *pos)
{
    interruptible_sleep_on(&wq);
    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current-
    >comm);
    return 0; /* EOF */
}

ssize_t sleepy_write (struct file *filp, const char *buf, size_t count,
loff_t *pos)
{
    wake_up_interruptible(&wq);
    return count; /* succeed, to avoid retrial */
}
```

Nonblocking I/O

- Another point we need to touch on before we look at the implementation of full featured read and write methods is the role of the O_NONBLOCK flag in filp->f_flags.
- The flag is defined in <linux/fcntl.h>, which is automatically included by <linux/fs.h>.
- O_NDELAY is an alternate name for O_NONBLOCK,

Nonblocking I/O

- The behavior of read and write is different if O_NONBLOCK is specified.
- In this case, the calls simply return -EAGAIN if a process calls read when no data is available or if it calls write when there’s no space in the buffer.

Sample

```
ssize_t scull_p_read (struct file *filp, char *buf, size_t count,
loff_t *f_pos)
{
    if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
}

static inline int spacefree(Scull_Pipe *dev)
{
    if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
}
```

User space sample

```
int main(int argc, char **argv)
{
    int delay=1, n, m=0;
    fcntl(0, F_SETFL, fcntl(0,F_GETFL) | O_NONBLOCK); /* stdin */
    fcntl(1, F_SETFL, fcntl(1,F_GETFL) | O_NONBLOCK); /* stdout */
    while (1) {
        n=read(0, buffer, 4096);
        if (n>=0)
        m=write(1, buffer, n);
        if ((n<0 || m<0) && (errno != EAGAIN))
                break;
        sleep(delay);
    }
    perror( n<0 ? "stdin" : "stdout");
    exit(1);
}
```

poll and select

- Applications that use nonblocking I/O often use the poll and select system calls.
- poll and select have essentially the same functionality:
  - both allow a process to determine whether it can read from or write to one or more open files without blocking.
  - They are thus often used in applications that must use multiple input or output streams without blocking on any one of them.

poll

- unsigned int (*poll) (struct file *, poll_table *);
- The poll_table structure is used within the kernel to implement the poll and select calls; it is declared in <linux/poll.h>.

poll_wait

- Every event queue that could wake up the process and change the status of the poll operation can be added to the poll_table structure by calling the function poll_wait:
- void poll_wait (struct file *, wait_queue_head_t *, poll_table *);

poll return value

- Another task performed by the poll method is returning the bit mask describing which operations could be completed immediately:
  - POLLIN
  - POLLRDNORM
  - POLLOUT
  - POLLWRNORM

Sample

```
unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
    Scull_Pipe *dev = filp->private_data;
    unsigned int mask = 0;
    int left = (dev->rp + dev->buffersize - dev->wp) % dev->buffersize;
    poll_wait(filp, &dev->inq, wait);
    poll_wait(filp, &dev->outq, wait);
    if (dev->rp != dev->wp)
        mask |= POLLIN | POLLRDNORM;
    if (left != 1)
        mask |= POLLOUT | POLLWRNORM;
    return mask;
}
```

Userspace poll

- #include <sys/poll.h>
- int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
- struct pollfd

```
 {
   int fd; /* file descriptor */
   short events; /* requested events */
   short revents; /* returned events */
};
```

Userspace sample

```
struct pollfd pfd[2];
pfd[0].fd = 0;
pfd[0].events = POLLIN;
pfd[0].revents = 0;
pfd[1].fd = 1;
pfd[1].events = POLLOUT;
pfd[1].revents = 0;

switch (poll (pfd, 2, 10000))
{
    case -1:
            perror ("poll()");
            break;
    case 0:
            printf (“none is ready.\n");
            break;
    default:
            if (pfd[0].revents == POLLIN || pfd[1].revents == POLLOUT)
                        printf (“event\n”);
            break
}
```

Question?