REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Using ioctl()

The system call ioctl() is provided for device-specific custom commands (such as format, reset and shutdown) that are not provided by standard system calls such as read(), write and mmap().

To invoke ioctl commands of a device, the user-space program would open the device first, then send the appropriate ioctl() and any necessary arguments.

```
#include <sys/ioctl.h>

int ioctl(int fd, int command, ...);
```

and on success, **0** is returned and on error, **-1** will be returned and errno will be set to:

EBADF	Bad file descriptor
ENOTTY	File descriptor not associated with character special device, or the request does not apply to the kind of object the file descriptor references.
EINVAL	Invalid **command** or **argp**

In the kernel code of the device, the entry point for ioctl() looks like:

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

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

where arg can be used directly either as a long or a pointer in user-space. In the latter case, the pointer points to user-space data, therefore to access the user-space data, one would use put_user(), get_user(), copy_to_user() and copy_from_user() functions.

Here is an example of an ioctl implementation in a driver:

```
static int mydrvr_ioctl (struct inode *inode, struct file *file,
                         unsigned int cmd, unsigned long arg) {
  if (_IOC_TYPE(cmd) != MYDRBASE) return -EINVAL;

  switch (cmd) {
  case MYDRVR_RESET:
    .....
    return 0;

  case MYDRVR_OFFLINE:
    .....
    return 0;

  case MYDRVR_GETSTATE:
    if (copy_to_user((void *)arg, &mydrvr_state_struct, sizeof(mydrvr_state_struct))) {
      return -EFAULT;
    }
    return 0;

  default:
    return -EINVAL;
  }
}
```

From kernel 2.6.36 onwards, the Big Kernel Lock (a single lock that only allow one process making a system call to the device at a time) is removed, so the member **.ioctl**, which assumes protection from BKL, is also removed from **struct file_operations**.

All system call implementations are now required to use its own synchronization methods such as spinlocks, mutex, semaphores, and atomic operations (discussed in the next lab) to ensure atomic access to shared data.

In **struct file_operations**, your ioctl() implementation must now be registered with the member **.unlocked_ioctl()**. For example:

```
struct file_operations asgn1_fops = {
  .owner = THIS_MODULE,
  ......
  .unlocked_ioctl = asgn1_ioctl,
  ......
};
```

The unlocked version of ioctl should look like:

```
static long asgn1_ioctl (struct file *filp, unsigned int cmd, unsigned long arg);
```

Note the differences of type of return value and number of parameters from the locked version above.

1.1 Defining ioctl() commands

Programmers much choose a number for the integer command representing each command implemented through ioctl. The number should be unique across the system. Picking arbitrary number is a bad idea, because:

Two device nodes may have the same major number. An application could open more than one device and mix up the file descriptors, thereby sending the right command to the wrong device. Sending wrong ioctl commands can have catastrophic consequences, including damage to hardware. A unique magic number should be encoded into the commands with one of the following macros:

```
_IO  (magic, number)
_IOR (magic, number, data_type)
_IOW (magic, number, data_type)
_IORW(magic, number, data_type)
```

where **magic** is the 8-bit magic number unique to the device. For currently-used magic numbers in the kernel (therefore you should not use), please have a look at Documentation/ioctl-number.txt and include/asm-generic/ioctl.h under the Linux source directory. For all tasks in this lab, please use *k* as the magic number of your modules.

number is the sequential number you assign to your command. It is local to your device driver and is at the discretion of the driver developer.

data_type is used to code the size of the data structure passed from/to the user space and kernel space in the ioctl command. Rather than putting the actual size in the field, one must put the actual data type or structure (not pointer to the data structure) in the field. Then a **sizeof()** primitive is applied to it to get the size of the data structure.

For example:

```
MY_IOCTL = _IOWR('k', 1, struct my_data_structure);
```

Also since the field size only has 14-bit, therefore the largest size of the data struct is 16KB.

If you command does not involve in passing data, then you should use _IO(); if your command lets the user-space program read data from the data structure, use _IOR(); if the user-space program writes to the data structure an passes to the kernel, then use _IOW(); otherwise if the data structure is both read and written to by the user-space program, then use _IORW().

Here is an example for encoding ioctl() command numbers:

```
#define MYDRBASE 'k'
#define MYDR_RESET _IO(  MYDRBASE, 1)
#define MYDR_STOP  _IO(  MYDRBASE, 2)
#define MYDR_READ  _IOR( MYDRBASE, 3, my_data_buffer)
```

In your ioctl implementation (in the kernel module), you can use the following macros to decode information from the ioctl command integer:

```
_IOC_TYPE(cmd)            /* gets the magic number of the device
                             this command targets */
_IOC_NR(  cmd)            /* gets the sequential number of the command
                             within your device */
_IOC_SIZE(cmd)            /* gets the size of the data structure */
_IOC_DIR( cmd)            /* gets the direction of data transfer,
                             can be one of the following:
                             _IOC_NONE
                             _IOC_READ
                             _IOC_WRITE
                             _IOC_READ | _IOC_WRITE
                          */
```

1.2 Automatically create device nodes under /dev with the udev system

udev is the device manager for the Linux kernel, which manages device nodes automatically during module insertion and removal, thus preventing the troubles of manually creating and removing devices nodes and matching the major and minor numbers.

Device nodes are commonly created by the init() function. Since once the device node is created, it will be accessible by other modules, or user-space program. Therefore the device node is usually created at the end of init(), where everything is already initialized and the device is ready to be used.

To create a node, first you must create a class using:

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

struct class *class_create(struct module *owner, const char *name);
```

where owner is usually set to THIS_MODULE and name will be the name of the class, which does not have to be same as the module name.

Then you must create the node itself using:

```
struct device *device_create(struct class *cls, struct device *parent, dev_t devt,
                             const char **fmt...);
```

where **cls** is the class you've just created; **parent** is parent node, which is set to **NULL** for our assignments; **dev** is our entry and fmtis the name of the node appears under /dev

Then in exit(), you must remove the node(s) and the class by:

```
void device_destroy(struct class *cls), dev_t dev);
void class_destroy(struct class *cls);
```

Here is an example:

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

struct class *my_class;
dev_t my_dev;

/* ... */

static int __init my_init(void) {
    /* ... */

    /* create node */
    my_class = class_create(THIS_MODULE, "my_class");
    device_create(my_class, NULL, my_dev, "mycdrv");
```

```
    return 0
}


static void __exit my_exit(void) {
    /* remove node */
    device_destroy(my_class, my_dev);
    class_destroy(my_class);

    /* ... */
}

module_init(my_init);
module_exit(my_exit);
```

Important
For all assignments in COSC440, we will use dynamic major number allocation and udev

1.3 Exercise: using ioctl to pass data

Write a simple module that uses the ioctl directional information to pass a data buffer of fixed size back and forth between the driver and the user-space program.

The size and directions(s) of the data transfer should be encoded in the command number.

You will need to write a user-space application to test this.

1.4 Exercise: using ioctl() to pass data of variable length

Extend the previous exercise to send a buffer whose length is determined at run time. You will probably need to use the _IOC macro directly in the user-space program. (See linux/ioctl.h.)