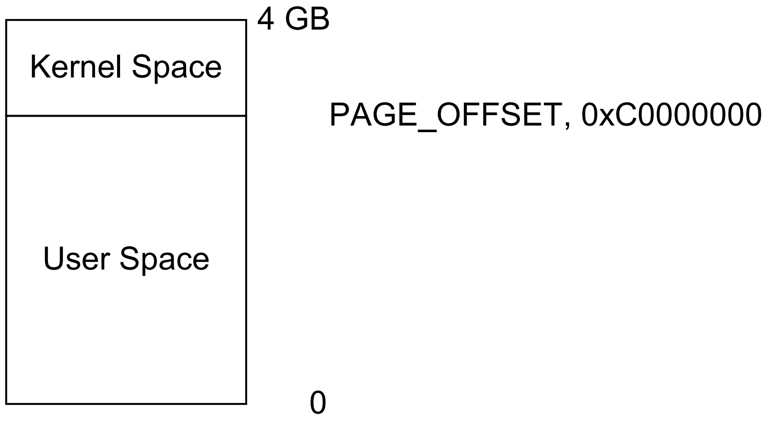
Notes on Linux Drivers

Duy-Ky Nguyen, PhD

Dec 10 2004

1. Introduction

In Linux, application programs runs in user space and kernel in kernel space. Only kernel have direct access to HW/SW device via drivers, so user programs must access devices via drivers loadable/unloadable to kernel.



The entire addressable area of memory (4 GB on 32-bit platforms) is split into 2 major areas – kernel space and user (or application) space. PAGE_OFFSET defines this split and is actually configurable in asm/page.h. The kernel space is located above the offset, and user space is kept below. The default for PAGE_OFFSET on the Intel platform is 0xc0000000 and thus provides the kernel with approximately 1 GB of memory, leaving 3 GB for user space consumption.

The Simplest Module

```
// foo_drv.c
#include <linux/module.h>   // init, exit, MODVERSION, …

#include <linux/kernel.h>   // KERN_INFO, KERN_DEBUG, …

static __init int init_module(void)    // __init for function to init module
{
   printk (KERN_DEBUG “Foo_drv init!\n”);
return 0;
}

static __exit void cleanup_module (void)    // __exit  for function to exit module
{
   printk(KERN_DEBUG “Foo_drv exit!\n”);
}
```

To compile, assume /usr/src/linux/include/linux/module.h

```
gcc -D__KERNEL__ -I/usr/src/linux/include -DMODULE _Wall -O2 -c foo_drv.c -o
foo_drv.o
```

To load driver into kernel,

```
insmod foo_drv.o
```

It's now idle in the kernel, but not used until invoked by `open()` by user. It's then back idle by user function `close()`.

To check if driver loaded

```
lsmod
```

or

```
modinfo -p foo_drv.o
```

To remove driver out from kernel

```
rmmod foo_drv   // module name hello happens to be base name ( without .o )
```

Devices are of 3 types : character with basic unit of 1 byte, block with basic unit of block, and packet for network device.

Device is another type of file, so the next section is about file.

2. Linux Files

In Linux, almost *everything is a file*. This means that, in general, programs can use disk files, serial ports, printers, and other devices in exactly the same way they would use a file. Except network devices, we need to use only 5 basic functions: open, close, read, write, and ioctl (to pass control info to a device driver).

A file has a name and some properties (creation/modification date, r-w-x permissions, …). The properties are stored in the files's *inode*, a special block of data in the file system that also contains the length of the file and its storage location.

A directoriy is a file that holds the inode numbers and names of other files.

Even HW devices are represented by files. For example, as the superuser, you mount a CDROM drive as a file

```
# mount -t iso9660 /dev/hdc /mnt/cdrom
# cd /mnt/cdrom
```

3. Character Devices

Char devices have to register themselves with the kernel using the following syntax

```
int register_chrdev (unsigned int major, const char *name, struct file_operations
*fops)
```

A device is indentified by its major number and its name, *eg.* the first hard drive (IDE0) has major number of **3** and its name is “**hda**”

In Linux, almost *everything is a file*. Device has no exception, so it has its own file operations.

```
struct file_operations {
   struct module *owner;
   int (*open) (struct inode *, struct file *);
   int (*release) (struct inode *, struct file *);
   int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
   ssize_t (*read) (struct file *, char *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
   . . .
}
```

When the character device is registered with the kernel, its file_operations structure and name is added to the global `chrdevs array of devive_struct` structures where the major number indexes it.

3.1. Debug

To control debug message

```
#define DEBUG

#ifdef DEBUG
#define MSG(string, args…) printk(KERN_DEBUG “foo:” string, ##args)
#else
#define MSG(string, args…)
#endif
```

3.2. Init and CleanUp

When load module using `insmod`

```
int __init init_module(void)
{
   int res;

   if (foo_name == NULL)
      foo_name = "foo";

   /* register device with kernel */
   res = register_chrdev(FOO_MAJOR, foo_name, &foo_fops);
   if (res) {
      MSG("can't register device with kernel\n");
   }

   return res;
}
```

When unload module with rmmod

```
void cleanup_module(void)
{
   struct page *page;

   /* unregister device and proc entry */
   unregister_chrdev(FOO_MAJOR, "foo");

   return;
}
```

3.3. Open and Release

When using function open()

```
static int foo_open(struct inode *inode, struct file *file)
{
   /* increment usage count */
   MOD_INC_USE_COUNT; // mod_cnt++


   return 0;
}
```

When using function close()

```
static int foo_release(struct inode *inode, struct file *file)
{
   MOD_DEC_USE_COUNT; // mod_cnt--
   return 0;
}
```

MOD_INC_USE_COUNT and MOD_DEC_USE_COUNT are used to count module usage. The kernel will only allow unloading of modules with usage count of zero.

3.4. Read and Write

Reading data from device

```
copy_to_user(void *to, void *from, unsigned long size)
```

Writing data to device

```
copy_from_user(void *to, void *from, unsigned long size)
```

3.5. IOCTL

IOCTL is used to set or get parameters from a running driver. Every device has a unique IOCTL base number along with a range of command. An IOCTL command consists of an upper 16-bit base and 16-bit command (256 coomands)

Linux distinguishes between 4 types of IOCTL function calls: direct _IO(), read _IOR(), write _IOW() or read-write _IOWR().

```
/* ioctl's for foo. */
#define FOO_IOCTL_BASE     0xbc
#define FOO_CHK            _IO(FOO_IOCTL_BASE, 0)
#define FOO_READ           _IOR(FOO_IOCTL_BASE, 1, unsigned long)
#define FOO_WRITE          _IOW(FOO_IOCTL_BASE, 2, unsigned long)
```

```
static int foo_ioctl(struct inode *inode, struct file *file,
            unsigned int cmd, unsigned long arg)
{

   /* make sure that the command is really one of foo's */
   if (_IOC_TYPE(cmd) != FOO_IOCTL_BASE)
      return -ENOTTY;

   switch (cmd) {
      case FOO_CHK: {
           printk(“Hello, kernel!\n”);
           return 0;
      }
      case FOO_READ: {
           if (put_user(foo_data_read, (unsigned long *)arg))
                return -EFAULT;
           break;
      }

      case FOO_WRITE: {
           if (put_user(foo_data_read, (unsigned long *)arg))
                return -EFAULT;
           break;
      }


      default: {
           MSG("ioctl: no such command\n");
           return -ENOTTY;
      }
   }

   /* to keep gcc happy */
   return 0;
}
```

3.6. Module Usage in User-Space

To use module in **user-space**

```
int main(void)
{
  int fd = open("/dev/foo", O_RDWR);

  /* complain if the open failed */
  if (fd == -1) {
    perror("open");
    return 1;
  }

  /* complain if the ioctl call failed */
  if (ioctl(fd, FOO_CHK) == -1) {
    perror("ioctl");
    return 2;
  }

  printf("dmesg | tail => Hello, kernel!\n");

  return 0;
}
```

4. IO Port

Read : `inb(), inw()`

Write : `outb(), outw()`

```
int __init foo_init(void)
{
   int result;

  result = check_region(foo_base, FOO_SIZE);
  if (result) {
    printk(KERN_INFO "foo: can't get I/O port address 0x%lx\n",
      foo_base);
    return result;
  }
  request_region(foo_base, FOO_SIZE, "foo");

   result = register_chrdev(major, "foo", &foo_fops);
   if (result < 0) {
      printk(KERN_INFO "foo: can't get major number\n");
      release_region(foo_base,FOO_SIZE);
      return result;
   }
   if (major == 0) major = result; /* dynamic */

   return 0;
}
```

```
void foo_cleanup(void)
{

    unregister_chrdev(major, "foo");

   release_region(foo_base,FOO_SIZE);

}
```

5. IO Mem

Read : `readb(), readw()`

Write : `writeb(), writew()`

```
int foo_init(void)
{
    int result;

      result = check_mem_region(foo_base, FOO_SIZE);
      if (result) {
           printk(KERN_INFO "foo: can't get I/O mem address 0x%x\n",
              foo_base);
           return result;
      }
      request_mem_region(foo_base, FOO_SIZE, "foo");

      /* also, ioremap it */
      foo_base = (unsigned long)ioremap(phyBase, phySize);

    result = register_chrdev(major, "foo", &foo_fops);
    if (result < 0) {
        printk(KERN_INFO "foo: can't get major number\n");
        release_mem_region(foo_base,FOO_SIZE);
        return result;
    }
    if (major == 0) major = result; /* dynamic */

    return 0;
}
```

```
void foo_cleanup(void)
{

    unregister_chrdev(major, "foo");
      iounmap((void *)foo_base);
      release_mem_region(foo_base, FOO_SIZE);

}
```

Remark 1

IO Port uses: request_region(), release_region(), inb(), outb()

IO Mem uses: request_mem_region(), release_mem_region(), AND ioremap(), readb(), writeb()

6. MMAP

In user-space, mmap() provides user address which is passed to kernel via ioctl().

In kernel space, remap_page_range() maps the user address above to physical address which is provided by pci_resource_start() or by kmalloc() and virt_to_phy(). Note that kmalloc() gives kernel (virtual) address which is required converted to physical address by virt_to_phy().

6.1. User Space

```
void  *  mmap(void *start, size_t length, int prot , int flags, int fd, off_t
offset);
```

```
int munmap(void *start, size_t length);
```

The mmap function asks to map length bytes starting at offset offset from the file (or other object) specified by the file descriptor fd into memory, preferably at address start. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by mmap, and is never 0.

As device is also a file in kernel space, so mmap can be used to map kernel memory into user memory.

```
fd = open("/dev/fpga",O_RDWR);
usrAddr = mmap(0, Size, PROT_READ | PROT_WRITE, MAP_SHARED, (int)fd, 0 );
```

6.2. Kernel Space

```
int remap_page_range(unsigned long virt_add, unsigned long phys_add, unsigned long size,
pgprot_t prot);
```

This function sits at the heart of *mmap*. It maps size bytes of physical addresses, starting at phys_addr, to the virtual address virt_add. The protection bits associated with the virtual space are specified in prot.

virt_add : it's indeed vma->start where VMA is created by the user mmap().

phys_add : it's physiavl address of the device. It's determined either by (1) kernel-space pci_resource_start() or by (2) kmalloc() and converted by virt_to_phys().

size : The dimension, in bytes, of the area being remapped.

prot : The ''protection'' requested for the new VMA. The driver can (and should) use the value found in vma->vm_page_prot.

In the simplest way, whene there's only 1 mmap address space, remap_page_range() is invoked within drv_mmap() method as drv_mmap() method is called only once

```
static int drv_mmap(struct file *file, struct vm_area_struct *vma)
{
   unsigned long size = vma->vm_end - vma->vm_start;

   if (remap_page_range(vma->vm_start, phyAddr, size, vma->vm_page_prot))
      return -EAGAIN;

   return 0;
}
```

If there're more than 1 mmap address, drv_mmap() is used to init global mmap_obj

User	Kernel
ioctl(GET_MM, phyBase, Size) // for Offset	GET_MM : copy_to_user(phYBase, Size)
mmap(Size) // request mmap	VMA created
ioctl(SET_MM)	remap_page_range() if matched

```
int drv_mmap(struct file *filp, struct vm_area_struct *vma) // *vma created by mmap()
{

    printk("VMA created by mmap()\n");

    // Init mmap pMapObj
    pMapObj->vma = *vma;

    // Record the owner for checking before remap_page_range()
    pMapObj->pOwner = (void*)filp;

   return 0;
}
```

and remap_page_range() is invoked within drv_ioctl() method

```
int drv_ioctl (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long
arg)
{

      case (FPGA_IOC_GET_MMAP):     // send phyBase & Size to User
      {

            if (copy_from_user(iocreg, (fpga_ioc_reg *)arg, sizeof (fpga_ioc_reg))) {
                  printk("<<ERR>> GET_MMAP copy_from_user\n");
                  return -EFAULT;
            }

            pMapObj->phyBase = (u32)iocreg->mmPhyBase = phyBase; // pci_resource_start()
            pMapObj->phySize = (u32)iocreg->mmSize = phySize; //pci_resource_len()

            if (copy_to_user((fpga_ioc_reg *)arg, iocreg, sizeof (fpga_ioc_reg))) {
                  printk("<<ERR>> ioctl entry GET_MMAP copy_to_user\n");
                  return -EFAULT;
            }

            break;
      }

      case (FPGA_IOC_SET_MMAP):     // get UsrAddr from User
      {

            if (copy_from_user(iocreg, (fpga_ioc_reg *)arg, sizeof (fpga_ioc_reg))) {
                  printk("<<ERR>> GET_MMAP copy_from_user\n");
                  return -EFAULT;
            }

            pMapObj->phyBase = (u32)iocreg->mmPhyBase = phyBase; //pci_resource_start()
            pMapObj->phySize = (u32)iocreg->mmSize = phySize; //pci_resource_len();

            // Verify physical address
            if (pMapObj->phyBase == (u32)NULL) {
                  printk("ERROR - Invalid physical address (0x%08x), cannot map to user
space\n", (u32)pMapObj->phyBase );

                  return -1;
            }

            // and verify VMA before remap_page_range() to map kernel space to user one
            if ((pMapObj->pOwner == filp) && (pMapObj->vma.vm_start == iocreg-
>mmUsrBase)) {
```

```
                printk("vm_start %x, mmUsrBase %x\n", (unsigned int)pMapObj-
>vma.vm_start, (unsigned int)iocreg->mmUsrBase);

                if(remap_page_range(
                    pMapObj->vma.vm_start,                 // start of kernel space
                    pMapObj->phyBase,                      // start of Device space
                    pMapObj->vma.vm_end - pMapObj->vma.vm_start, // size
                    pMapObj->vma.vm_page_prot))      // access mode
                {

                    printk("<<ERR>> remap_page_range\n");

                    return -1;
                  }
            }

            break;
      }
```

7. Interrupt

```
      result = request_irq (fpga_IRQ /* IRQ number */,
                             fpga_ISR /* ISR function */,
                             SA_INTERRUPT|SA_SHIRQ,
                             "fpga" /* device name */,
                             dev);
```

When the driver is initialized by `insmod`, the device is idle until `open`() and idle again after `close`(). The drive is removed by `rmmod`. Therefore `request_irq`() should be invoked in `open`() and free in `close`(), so it can be used by others.

8. PCI

To init PCI device

```
   // Detect PCI device
   if (!pci_present())
      return -ENODEV;

   // The 1st word for vendor, 2nd for device
   dev = pci_find_device (SYMM_VENDOR_ID, FPGA_DEVICE_ID, dev);
   if (!dev) {
      printk(KERN_WARNING"No fpga found.\n");
      return -ENODEV;
   }

   if (pci_enable_device (dev) == 0)
      printk(KERN_INFO"fpga enabled\n");
   else
      printk(KERN_WARNING"Failed to enable fpga\n");
```

PCI has several address spaces,

```
11.1.1.44:~/_NFS_DEV/FPGA/MM_05 % lspci -v -s 0:a.0
00:0a.0 Bridge: PLX Technology, Inc.: Unknown device 9030 (rev 0a)
        Flags: medium devsel, IRQ 10
        Memory at fbf8ff80 (32-bit, non-prefetchable) [size=128]
        I/O ports at ed80 [size=128]
        Memory at fbe00000 (32-bit, non-prefetchable) [size=1M]
        Expansion ROM at fbdf0000 [disabled] [size=64K]
        Capabilities: [40] Power Management version 1
        Capabilities: [48] #06 [0080]
        Capabilities: [4c] Vital Product Data
```

For space N (0 ~ 5)

```
   phyAddr_N = pci_resource_start(dev, N);
   size_N = pci_resource_len(dev, N);
```

To get IRQ

```
   result = pci_read_config_byte (dev, PCI_INTERRUPT_LINE, &fpga_IRQ);
   if (result) {
      printk(KERN_WARNING"fpga: can't get interrupt info(%d)\n", result);
      return -ENODEV;
   }
```

9. Summary

IO Port	IO Mem	MMAP

kmalloc(ioc_data)
kfree(ioc_data)
	kmalloc(ioc_data)
	kfree(ioc_data)
		kmalloc(ioc_data)
		kfree(ioc_data)
		kmalloc(mmap_data)
		kfree(mmap_data)

ioctl()	ioctl()	ioctl()
		mmap(Size)

// for multi-user checking
check_region(phyBase)
request_region(phyBase)
release_region(phyBase)
	// for multi-user checking
	check_mem_region(phyBase)
	request_mem_region(phyBase)
	release_mem_region(phyBase)
		// for multi-user checking
		check_mem_region(phyBase)
		request_mem_region(phyBase)
		release_mem_region(phyBase)

// no remap for IO port	// for kernel access
	virtBase = ioremap(phyBase)
	iounmap(virtBase)
		// for kernel access
		virtBase = ioremap(phyBase)
		iounmap(virtBase)

		drv_mmap(VMA)
		remap_page_range(VMA, phyBase)

In kernel space
inb() outb()
inw() outw()
In user space
ioctl()
	In kernel space
	readb(), writeb()
	readw(), writew()
	In user psace
	ioctl()
		In user space
		datab = (char*)addr;
		(char*)addr = data;