LINUX DEVICE DRIVER

PRABHAT RANJAN

(e-mail : prabhat_ranjan@daiict.ac.in)

Device Driver

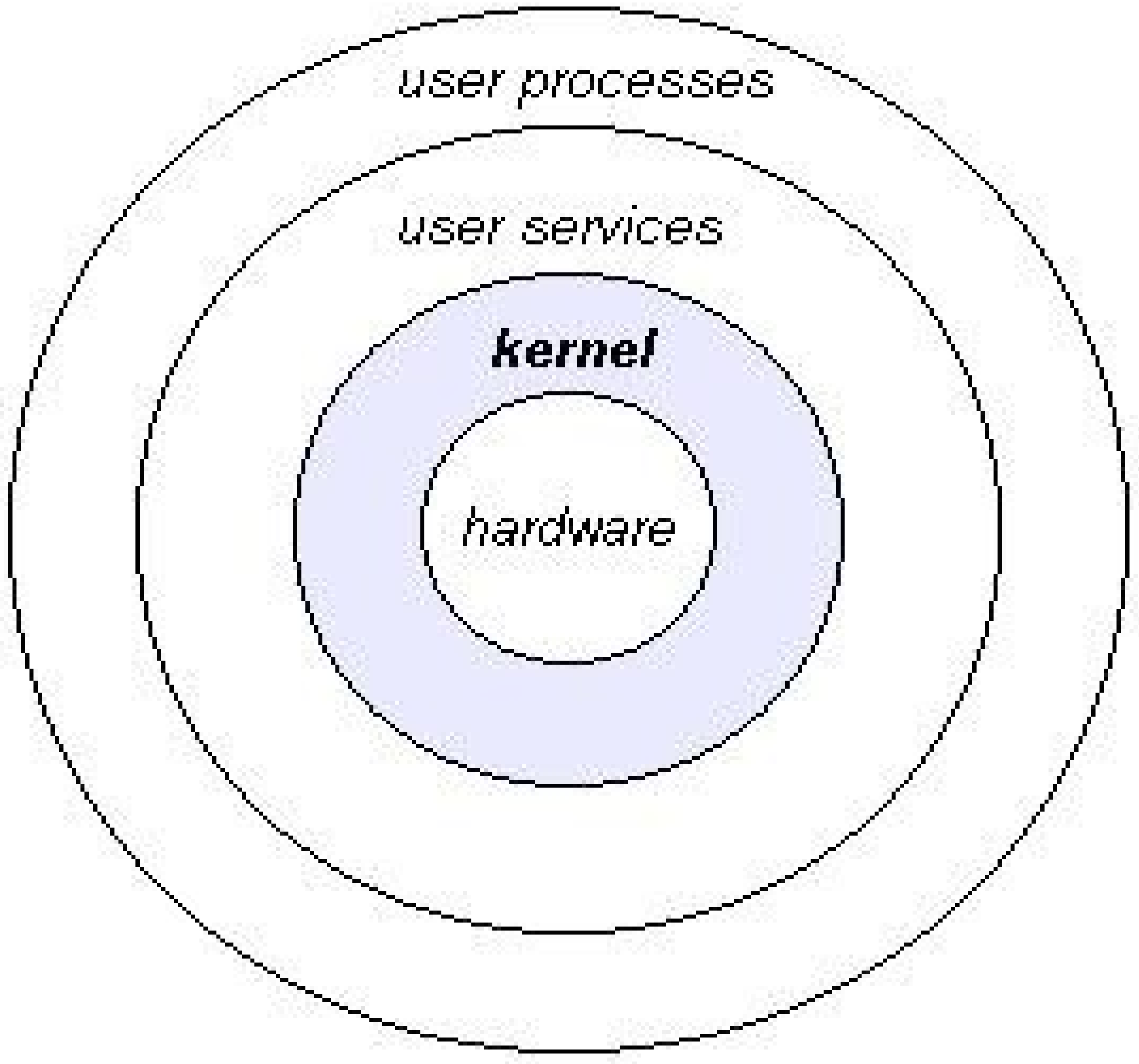
- A device driver consists of a set of routines that control a peripheral device attached to a workstation.
- The operating system normally provides a uniform interface to all peripheral devices.
- Linux and Unix present peripheral devices at a sufficiently high level of abstraction by observing that a large proportion of I/O devices can be represented as a sequence of bytes.

Device Driver

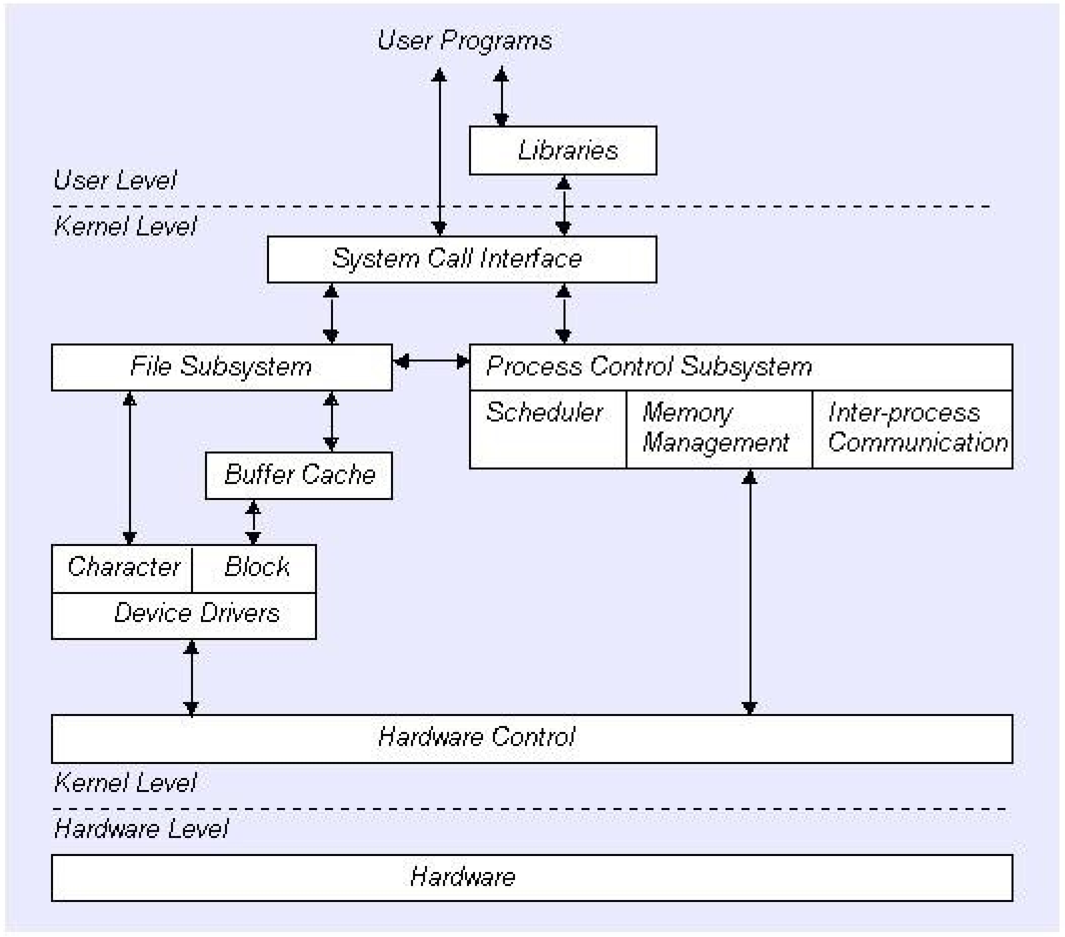
- Linux and Unix use the file--which is a well understood data structure for handling byte sequences--to represent I/O devices

Linux architecture in the most general terms.

kernel is shown wrapped around the hardware to depict that it is the software component that has direct access to--and control over--the system hardware, including the processor, primary memory, and

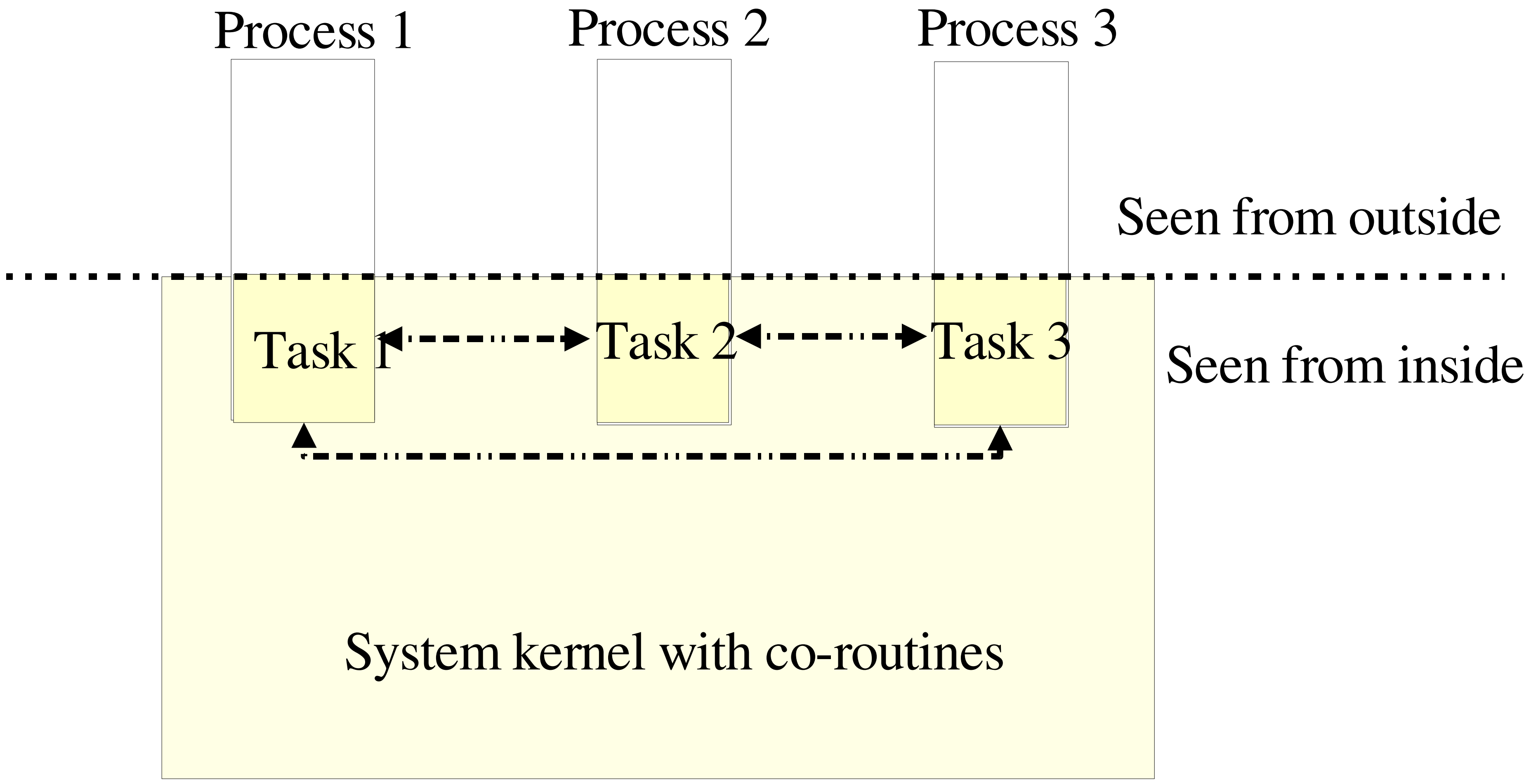


user-level programs communicate with the kernel using system calls, for instance, open(), read(), write(), ioctl(), close(), and the like.

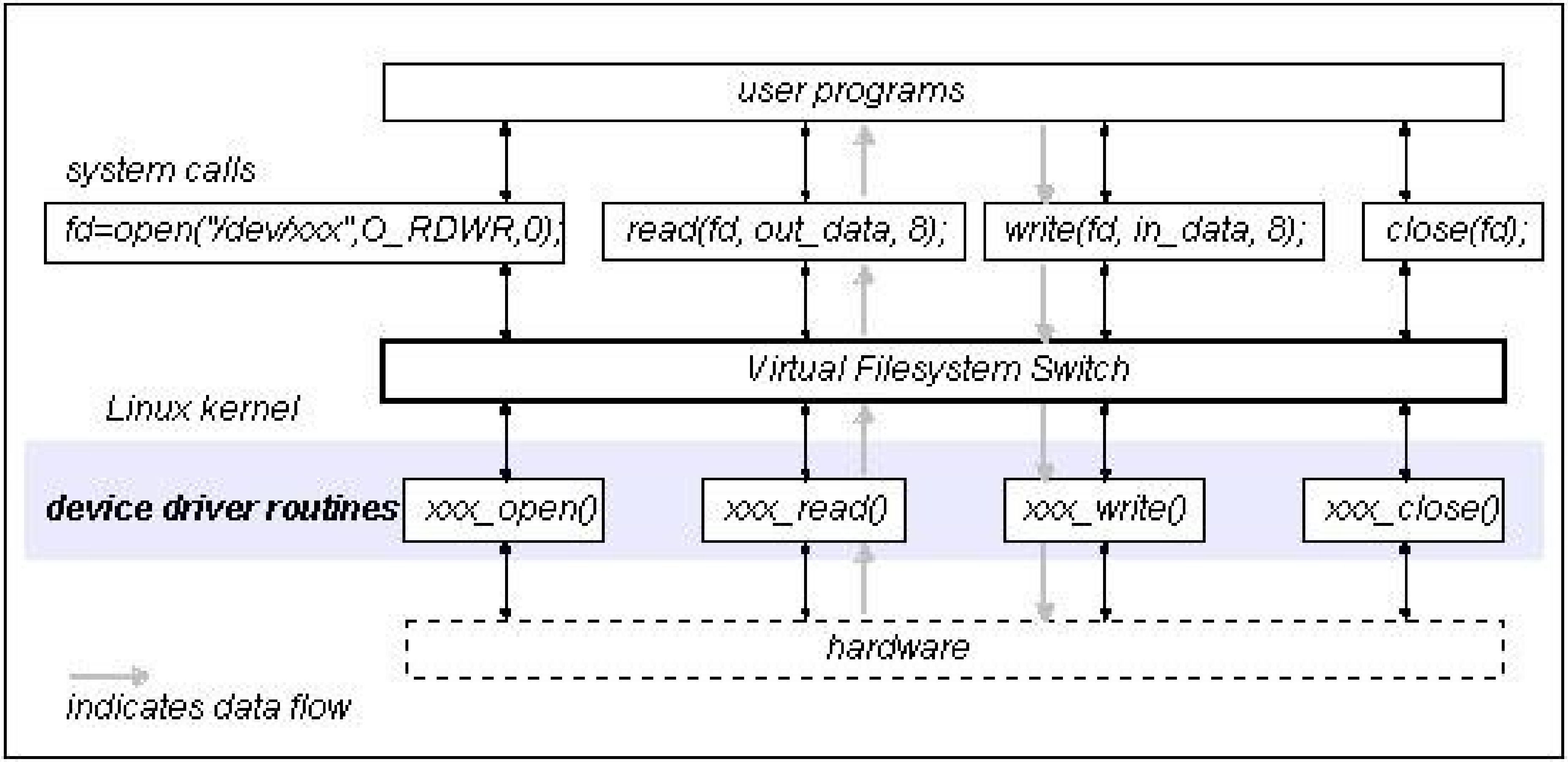


User Process and Kernel

- The kernel is not a separate task under Linux
- It is as if each process has a copy of the kernel
- When a user process executes a system call, it does not transfer control to another process, but changes its execution mode from user to kernel mode
- In kernel mode, while executing the system call, the process has access to the kernel address space, and



THE PROCESS AS SEEN FROM OUTSIDE AND FROM INSIDE


user programs
system calls
fd=open("/dev/xxx",O_RDWR,0);
read(fd, out_data, 8);
write(fd, in_data, 8);
close(fd);
Virtual Filesystem Switch
Linux kernel
device driver routines
xxx_open()
xxx_read()
xxx_write()
xxx_close()
hardware
indicates data flow

llseek
read
write
readdir
poll
ioctl
mmap
open
flush
release
fsync
fasync
lock
read_v
write_v

Kernel Modules

Linux Kernel modules are pieces of code (examples: fs, net, and hw driver) running in kernel mode that you can add at runtime.

The Linux core cannot be modularized: scheduling and interrupt management or core network, and so on.

Under "/lib/modules/KERNEL_VERSION/" - all the modules

Module loading and unloading

To load a module, type the following:

```
insmod MODULE_NAME parameters
```

example: `insmod ne io=0x300 irq=9`

modprobe in place of insmod

To unload a module, type the following:

```
rmmod MODULE_NAME
```

insmod	(8)	- install loadable kernel module
lsmod	(8)	- list loaded modules
rmmod	(8)	- unload loadable modules
depmod	(8)	- handle dependency descriptions for loadable kernel modules
modprobe	(8)	- high level handling of loadable modules
mknod	(1)	- make block or character special files
modules.conf [modules]	(5)	- configuration file for loading kernel modules
ksyms	(8)	- display exported kernel symbols
genksyms	(8)	- generate symbol version information
MAKEDEV	(8)	- create devices

```
#define MODULE
#include <linux/module.h>




int init_module(void)      { printk("<1>Hello,
world\n"); return 0; }

void cleanup_module(void)  { printk("<1>Goodbye
cruel world\n"); }

MODULE_LICENSE("GPL");
```

```
gcc -D__KERNEL__ -I/usr/src/linux/include -c
hello.c -o hello.o
```

DEVICE DRIVER FOR KEYBAORD LED CONTROL

```
# cat /proc/ioports | grep keyboard
0060-006f : keyboard
```

- We can use 0x60 to send and receive commands from KeyBoard
- To set the KeyBoard leds, the command to be sent is 0xED
- keyboard sends an acknowledgement 0xFA
- expects a new byte where

    bit 0 : scroll lock

    bit 1 : num lock

    bit 2 : caps lock

(Bits 3 to 7 are ignored.)

```
/********** keybmod.h  ************/
/* LEDs */
#define KEYBMOD_SCR 0x01
#define KEYBMOD_NUM 0x02
#define KEYBMOD_CAP 0x04

/* action */
#define KEYBMOD_ON  0x01
#define KEYBMOD_OFF 0x00
```

```
static struct file_operations keybmod_fops = {
        NULL,           /* lseek   */
        NULL,           /* read    */
        NULL,           /* write   */
        NULL,           /* readdir */
        NULL,           /* poll    */
        keybmod_ioctl,  /* ioctl   */
        NULL,           /* mmap    */
        keybmod_open,   /* open    */
        NULL,           /* flush   */
        keybmod_close,  /* close   */
};
```

```
static struct file_operations keybmod_fops = {
 ioctl : keybmod_ioctl, /* ioctl   */
 open : keybmod_open,   /* open    */
 release : keybmod_close,       /* close   */
};
```

```
int init_module (void)
{
        int ret;

        keybmod_dev = 0;

        ret = register_chrdev (keybmod_major, "keybmod", &keybmod_fops);

       return 0;

}
```

Major Device No.   Device Name   file_operation structure

```
/* unload the module */
void cleanup_module (void)
{
    if (keybmod_dev) {
        printk (KERN_ERR "keybmod: already being used\n");
        return;
    }

    unregister_chrdev (keybmod_major, "keybmod");

    return;
}
```

```
/* open */
static int keybmod_open  (inode, filp)
        struct inode *inode;
        struct file  *filp;
{
        /* check if keybmod isn't already being used */
        if (keybmod_dev) return -EBUSY;

        keybmod_dev++;
        MOD_INC_USE_COUNT;

        return 0;
}
```

```
/* close */
static int keybmod_close (inode, filp)
        struct inode *inode;
        struct file  *filp;
{
        keybmod_dev--;
        MOD_DEC_USE_COUNT;

        return 0;
}
```

```
ioctl (fd, KEYBMOD_NUM, KEYBMOD_ON);
ioctl (fd, KEYBMOD_NUM, KEYBMOD_OFF);

static int keybmod_ioctl (inode, filp, cmd, arg)

unsigned char real_cmd = arg ? cmd : 0x00;
/*if (arg == 1) real_cmd = cmd
   else real_cmd = 0x00; */

outb_p (0xed, 0x60); udelay (1000);
/* Check Keyboard acknowledgement */

outb_p (real_cmd, 0x60); udelay (1000);
/* Check Keyboard acknowledgement */

udelay (1000);
```

```
if (!keybmod_ack ()) printk (KERN_ERR "keybmod: keyboard
timeout!\n");
```

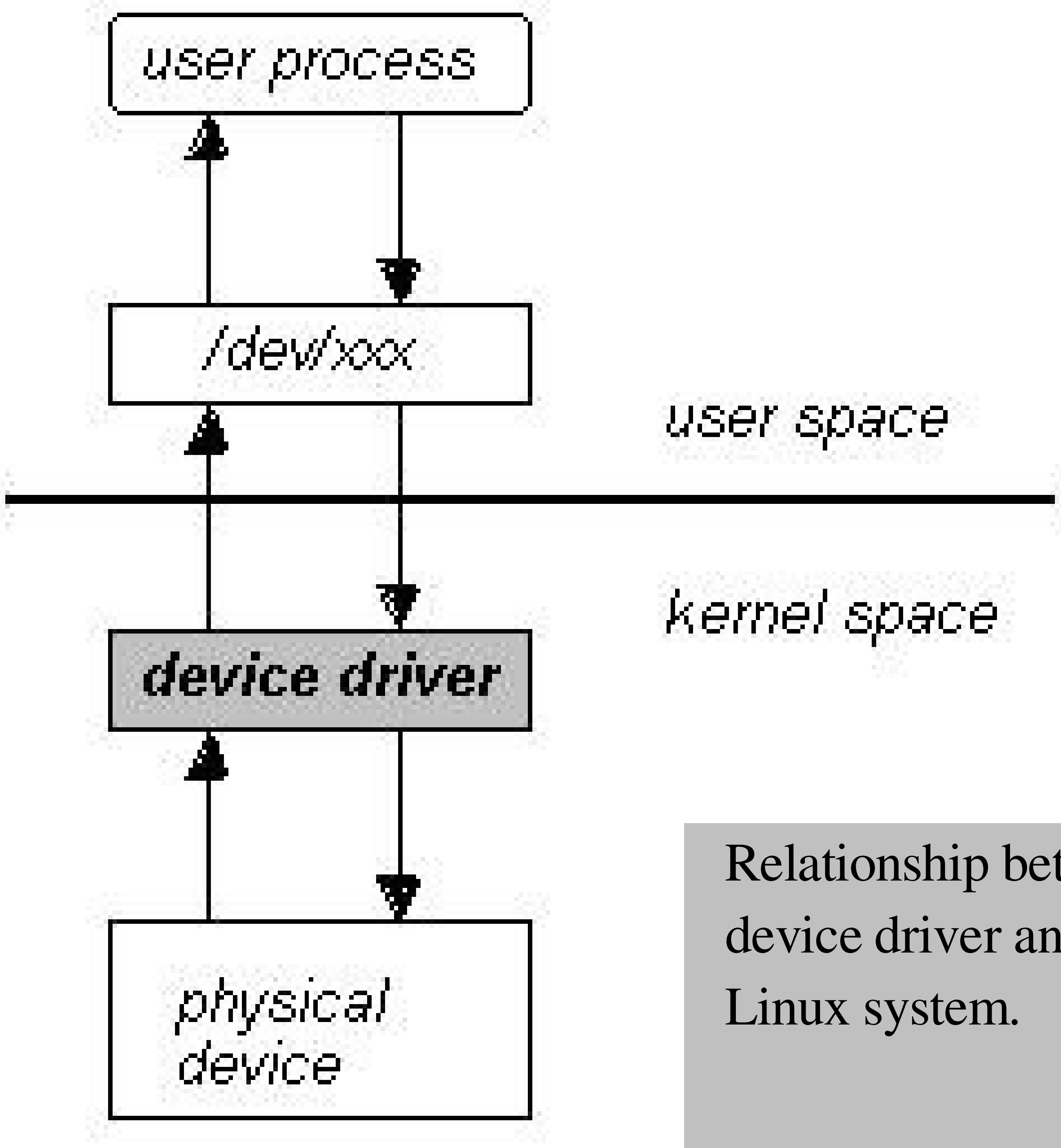
```
static int keybmod_ack (void)
{
        int retry = 5, timeout = 1000;

        while (inb_p (0x60) != 0xfa && retry--) udelay (timeout);
        return retry <= 0 ? 0 : 1;
}
```

Kernel Symbol Versioning

- Unresolved symbols while loading – dependency – modprobe
- Kernel version number change – insmod -f
- Incompatible changes across version – Module versioning – Kernel option

```
#define register_chrdev register_chrdev_Rc8dc8350
```

- Genksyms – generate symbol version number



Relationship between device driver and the Linux system.

User Space Program

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "keybmod.h"

int main (int argc, char **argv)
{
        int i, fd;

        if (argc != 2) {
                fprintf (stderr, "usage: %s device\n", *argv);
                return 0;
        }
        if ((fd = open (argv[1], O_RDWR)) < 0)
                perror ("keybctrl: open"), exit (1);

        for (i = 0; i < 5; i++) {
                fprintf (stderr, "Turn NumLock ON!\n");
                ioctl (fd, KEYBMOD_NUM, KEYBMOD_ON);
//              ioctl (fd, KEYBMOD_CAP, KEYBMOD_ON);
                usleep (500000);

                fprintf (stderr, "Turn NumLock OFF!\n");
                ioctl (fd, KEYBMOD_NUM, KEYBMOD_OFF);
//              ioctl (fd, KEYBMOD_CAP, KEYBMOD_OFF);
                usleep (500000);
        }

        close (fd);
        return 0;
}
```

A device driver provides the following features:

- A set of routines that communicate with a hardware device and provide a uniform interface to the operating system kernel.
- A self-contained component that can be added to, or removed from, the operating system dynamically.
- Management of data flow and control between user programs and a peripheral device.
- A user-defined section of the kernel that allows a program or a peripheral device to appear as a "/dev " device to the rest of the system's software.

Name space

• The name of the driver should be a short string.

• For instance, the parallel (printer) device is the "lp" device, the floppies are the "fd" devices, and the SCSI disks are the "sd" devices.

• To avoid name space confusion, the entry point names are formed by concatenating this unique driver prefix with a generic name that describes the routine.

• For instance, xxx_open() is the "open" routine for the "xxx" driver.

Changes in 2.6 Kernel

Standard template for a device driver under the Linux 2.4 kernel:

```
#define MODULE
#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>

static int __init name_of_initialization_routine(void) {
    /*
     * code here
     */
}
static void __exit name_of_cleanup_routine(void) {
    /*
     * code here
     */
}
module_init(name_of_initialization_routine);
module_exit(name_of_cleanup_routine);
```

- One common problem with many 2.4 and earlier device drivers is that they made assumptions about the names of the module initialization and cleanup functions

- When writing device drivers for 2.4 and earlier Linux kernels, one did not have to register the names of the module initialization and cleanup/exit functions if you used the default names init_module() and cleanup_module() for these functions

- This was always "wrong", but it is no longer acceptable

- Under the 2.6 kernel, you must use the module_init() and module_exit() macros to register the names of your initialization and exit routines

- These declarations are only strictly necessary if you ever intend to compile the specified module into the kernel, but it's always best to prepare for that case

- Next, under the 2.6 kernel, the #define MODULE statement is no longer necessary, either in your code or in your Makefile
- This symbol definition and others are now automatically defined (if necessary) and verified by the kernel build system, which you must now use when writing device drivers for the 2.6 kernel
- These are the basic structural changes that you need to make to an existing module to get it to compile and load into a 2.6 kernel
- However, once you load a module with this sort of structure, you will notice that an error message about a tainted module is displayed on standard output and in the system log (/var/log/messages). To eliminate this message, you need to add the following:

```
    MODULE_LICENSE("GPL");
```

- This macro, introduced in later versions of the 2.4 kernel, defines the module as being licensed under the terms of GPL Version 2 or later. Other valid values are "GPL v2", "GPL and additional rights", "Dual BSD/GPL" (your choice of BSD or GPL licensing), "Dual MPL/GPL" (your choice of Mozilla or GPL licensing), and "Proprietary", which is self-explanatory.

A generic template for a minimal 2.6 device driver would therefore be the following:

```
#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>
MODULE_LICENSE("GPL");
static int __init name_of_initialization_routine(void) {
    /* code goes here */
    return 0;
}
static void __exit name_of_cleanup_routine(void) {
    /* code goes here */
}
module_init(name_of_initialization_routine);
module_exit(name_of_cleanup_routine);
```

Aside from changes required to device drivers themselves, the most significant change to working with device drivers for the 2.6 Linux kernel are the changes in the build process

Accessing Procfs from Kernel

```
# cat /proc/arith/sum
0
# echo 7 > /proc/arith/sum
# echo 5 > /proc/arith/sum
# echo 13 > /proc/arith/sum
# cat /proc/arith/sum
25
```

```
static int __init sum_init(void) {
        struct proc_dir_entry *proc_arith;
        struct proc_dir_entry *proc_arith_sum;

        proc_arith = proc_mkdir("arith", 0);
        if (!proc_arith) {
                printk (KERN_ERR "cannot create /proc/arith\n");
                return -ENOMEM;
        }
        proc_arith_sum = create_proc_info_entry("arith/sum", 0, 0, show_sum);
        if (!proc_arith_sum) {
                printk (KERN_ERR "cannot create /proc/arith/sum\n");
                remove_proc_entry("arith", 0);
                return -ENOMEM;
        }
        proc_arith_sum->write_proc = add_to_sum;
        return 0;
}
```

```
/* Expect decimal number of at most 9 digits followed by '\n' */
static int add_to_sum(struct file *file, const char *buffer,
                      unsigned long count, void *data) {
        unsigned long val = 0;
        char buf[10];
        char *endp;
        if (count > sizeof(buf))
                return -EINVAL;
        if (copy_from_user(buf, buffer, count))
                return -EFAULT;
        val = simple_strtoul(buf, &endp, 10);
        if (*endp != '\n')
                return -EINVAL;
        sum += val;     /* mod 2^64 */
        return count;
```

```
static int show_sum(char *buffer, char **start, off_t offset, int length) {
        int size;

        size = sprintf(buffer, "%lld\n", sum);
        *start = buffer + offset;
        size -= offset;
        return (size > length) ? length : (size > 0) ? size : 0;
}
```