CENG 425

Linux Device Driver Programming

by Yusuf SOYMAN

What's a 'device-driver'?

- A special kind of computer program
- Intended to control a peripheral device
- Needs to execute 'privileged' instructions
- Must be integrated into the OS kernel
- Interfaces both to kernel and to hardware
- Program-format specific to a particular OS

Linux device-drivers

- A package mainly of ‘service functions’
- The package is conceptually an ‘object’
- But in C this means it’s a ‘struct’
- Specifically: struct file_operations { …; };
- Definition is found in a kernel-header:

  ‘/usr/src/linux/include/linux/fs.h’

Types of Device-Drivers

- Character drivers:
  - the device processes individual bytes
    (e.g., keyboard, printer, modem)

- Block drivers:
  - the device processes groups of bytes
    (e.g., hard disks, CD-ROM drives)

Linux has other driver-types

- Network drivers
- Mouse drivers
- SCSI drivers
- USB drivers
- Video drivers
- ‘Hot-swap’ drivers
- … and others

Developing a device-driver

- Clarify your requirements
- Devise a design to achieve them
- Test your design-concept (‘prototype’)
- ‘Debug’ your prototype (as needed)
- Build your final driver iteratively
- Document your work for future use

‘Open Source’ Hardware

- Some equipment manufactures regard their designs as ‘intellectual property’
- They don’t want to ‘give away’ their info
- They believe ‘secrecy’ is an advantage
- They fear others might copy their designs
- BUT: This hinders systems programmers!

Non-Disclosure Agreements

- Sometimes manufacturers will let ‘trusted’ individuals, or commercial ‘partners’, look at their design-specs and manuals
- College professors often are ‘trusted’
- BUT: Just to be sure, an NDA is required -- which prevents professors from teaching students the design-details that they learn

Some designs are ‘open’

- The IBM-PC designs were published
- Then other companies copied them
- And those companies prospered!
- While IBM lost market-share!
- An unfortunate ‘lesson’ was learned

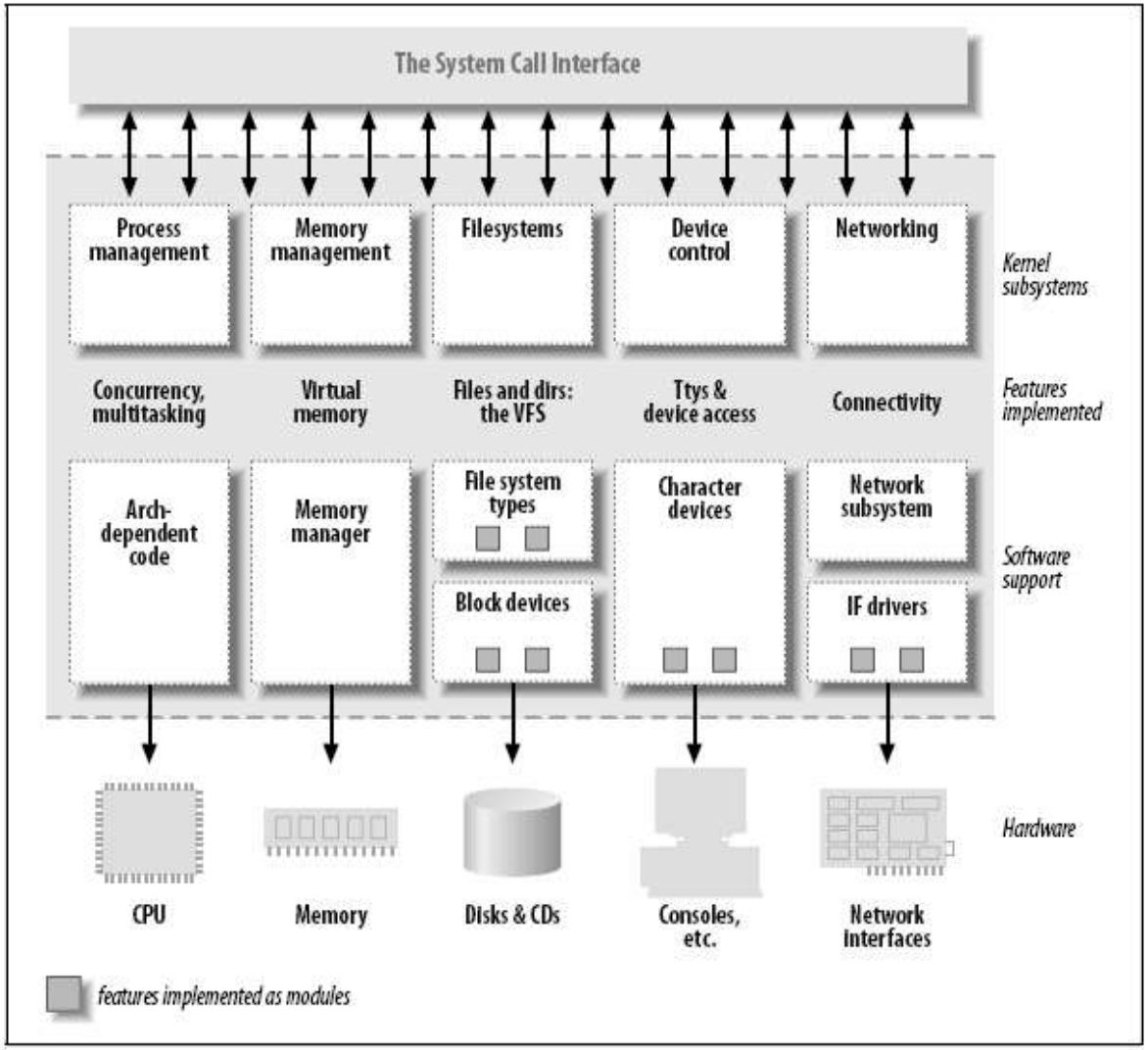
Advantage of ‘open’ designs

- Microsoft and Apple used to provide lots of technical information to programmers
- They wanted to encourage innovations that made their products more valuable
- Imagine hundreds of unpaid ‘volunteers’ creating applications for **your** platform!
- BUT: Were they ‘giving away the store’?

A ‘virtual device’

- To avoid NDA hassles, we can work with a ‘pseudo’ device (i.e., no special hardware)
- We can use a portion of physical memory to hold some data that we ‘read’ or ‘write’

Linux Kernel



Kernel Components

Process management

- Creating and destroying processes
- Handling their connection to the outside world (input and output).
- Communication among different processes (through signals, pipes, or interprocess communication primitives)
- Scheduling

Memory management

- Provides virtual addressing space for any and all processes
- The different parts of the kernel interact with the memory-management subsystem

Filesystems

- Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file.
- The kernel builds a structured filesystem on top of unstructured hardware
- The resulting file abstraction is heavily used throughout the whole system.
- In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium.

Kernel Components

Device control

- Almost every system operation eventually maps to a physical device.
- With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a *device driver*.
- The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive.

Networking

- Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events.
- The packets must be collected, identified, and dispatched before a process takes care of them.
- The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity.
- Routing and address resolution are implemented in the Kernel

Loadable Modules

- Linux allows extending at runtime the set of features offered by the kernel.
  - This means that you can add functionality to the kernel (and remove functionality as well) while the system is up and running.
- Each piece of code that can be added to the kernel at runtime is called a module.
- The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers.
- Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program.

Device Types (1)

Driver Modules can be classified into

- **Character Devices**
  - Accessed as a stream of bytes (like a file)
  - Typically just data channels, which allow only sequential access
    - Some char devices look like data areas and allow moving back and forth in them (example: frame grabbers)
  - A char driver is in charge of implementing this behavior
  - Driver needs to implement at least the *open*, *close*, *read*, and *write* system calls
  - Examples:
    - Text console *(/dev/console*)
    - Serial ports *(/dev/ttyS0*)

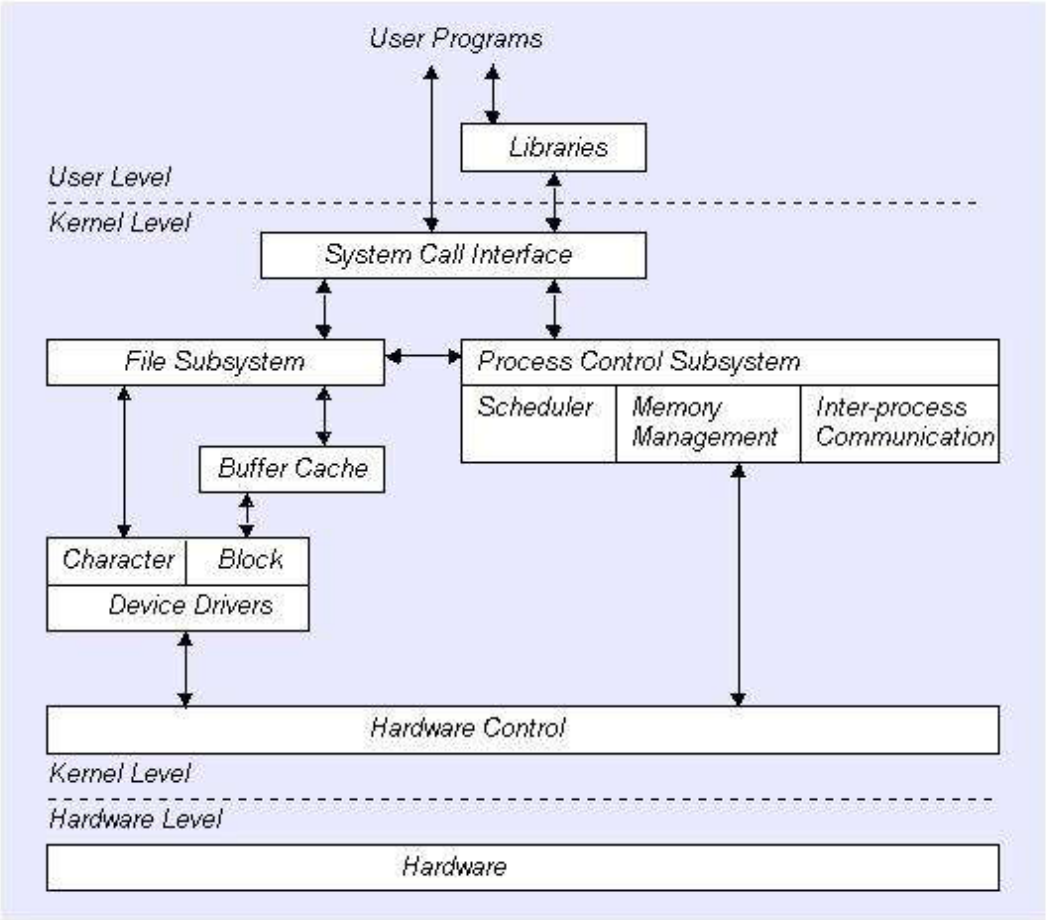
Device Types (2)

- **Block Devices**
  - In some Unix systems, block devices can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of 2).
    - Linux, instead, allows applications to read and write a block device like a char device
  - Like char devices accessed through filesystem nodes in the /dev directory
  - Char and block devices differ in the kernel/driver interface

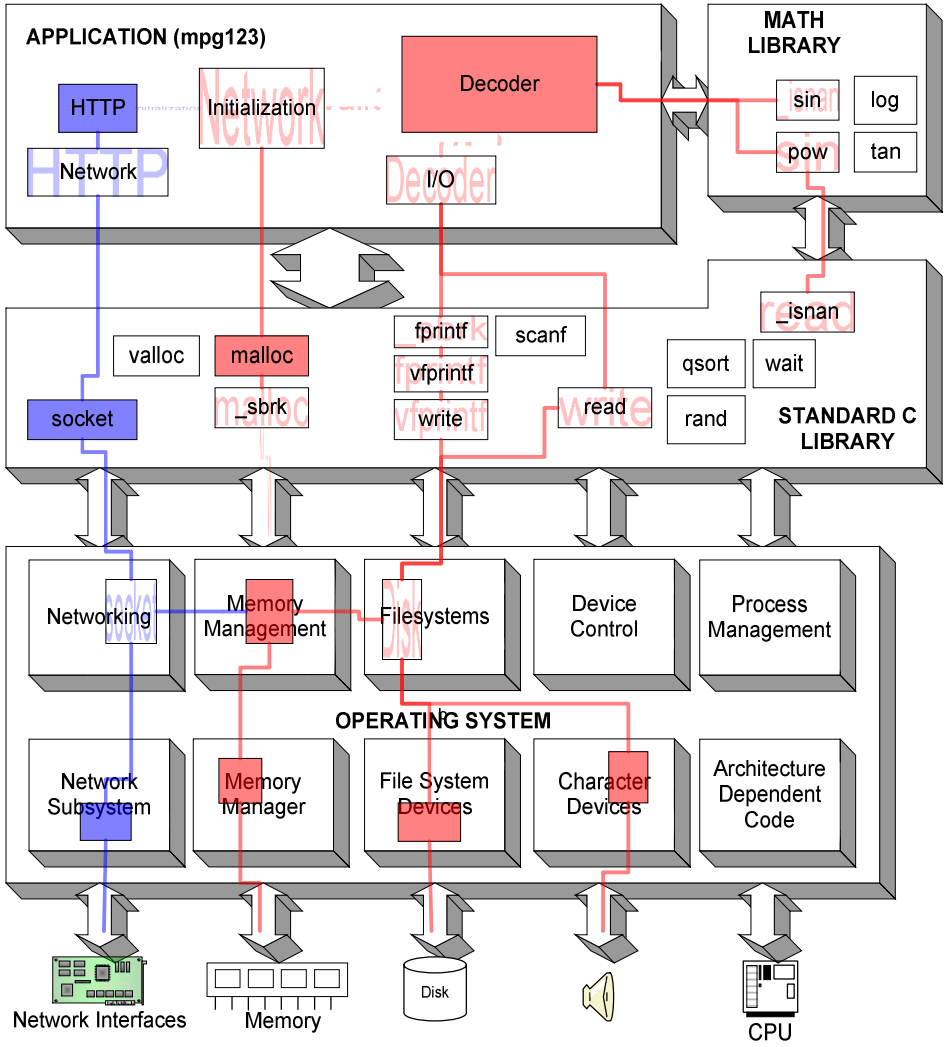
Device Types (2)

- Network Interfaces
  - Network transactions made through an interface
    - Hardware device
    - Pure software device (loopback)
  - Network interfaces usually designed around the transmission and receipt of packets
    - Network driver knows nothing about individual connections; it only handles packets
  - Char device? Block device?
    - not easily mapped to filesystem nodes
    - Network interfaces don’t have entries in the file system
    - Communication between the kernel and network device driver completely different from that used with char and block drivers

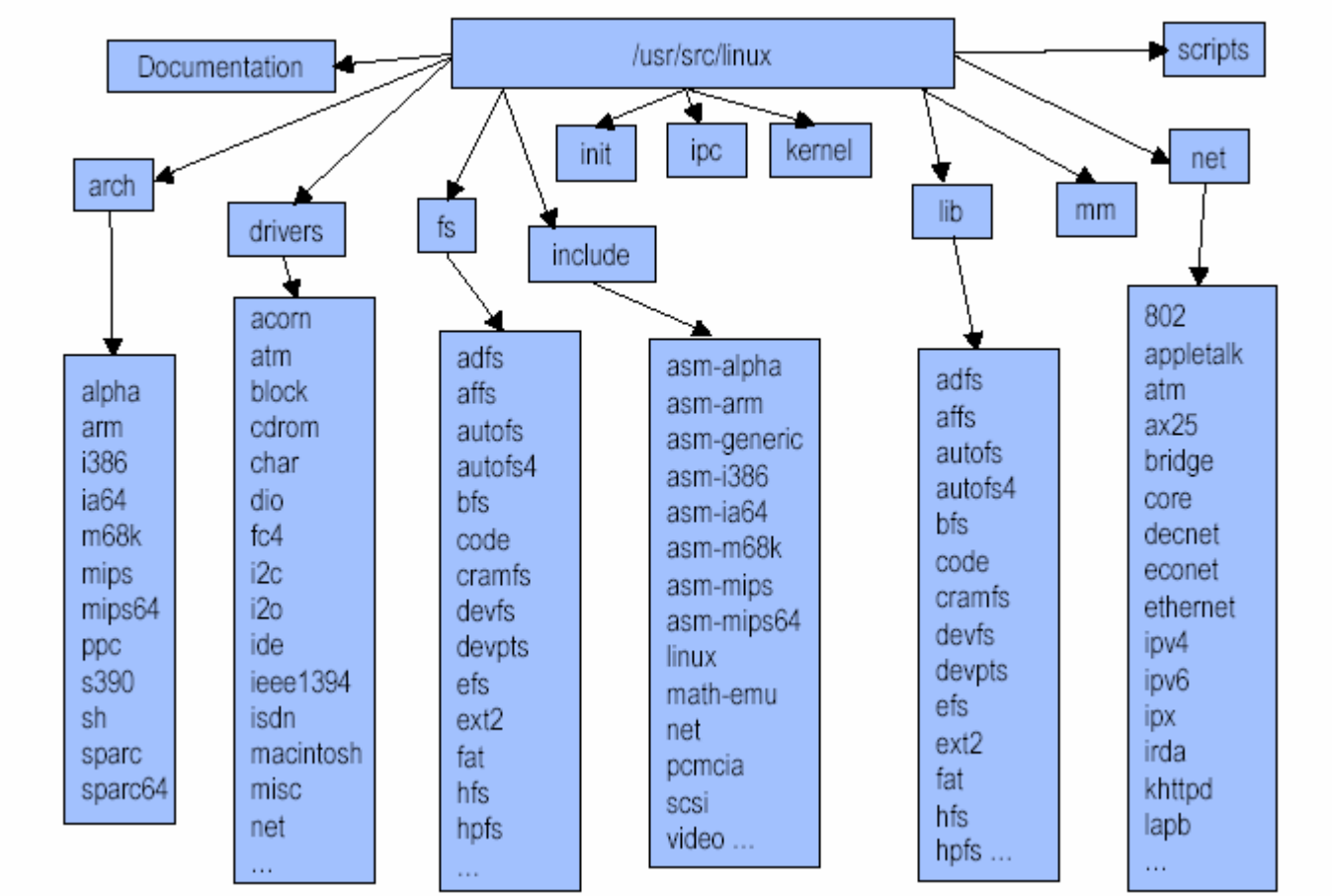
User program & Kernel Interface(1)



User program & Kernel Interface(2)



Linux Kernel Source Tree



Linux Code Layout (1)

- **Linux/arch**
  - Architecture dependent code.
  - Highly-optimized common utility routines such as memcpy
- **Linux/drivers**
  - Largest amount of code
  - Device, bus, platform and general directories
  - Character and block devices , network, video
  - Buses – pci, agp, usb, pcmcia, scsi, etc
- **Linux/fs**
  - Virtual file system (VFS) framework.
  - Actual file systems:
    - Disk format: ext2, ext3, fat, RAID, journaling, etc
    - But also in-memory file systems: RAM, Flash, ROM

Linux Code Layout (2)

- **Linux/include**
  - Architecture-dependent include subdirectories.
  - Need to be included to compile driver code:
    - gcc … -I/<kernel-source-tree>/include …
  - Kernel-only portions are guarded by #ifdefs

```
#ifdef __KERNEL__
  /* kernel stuff */
#endif
```

  - Specific directories: asm, math-emu, net, pcmcia, scsi, video.

Kernel Modules (1)

- So what are modules? A Linux module is nothing but an object file, usually created with the -c flag argument to gcc. The module itself is created by compiling an ordinary C language file without the main() function. Instead there will be a pair of init_module/cleanup_module functions:
  - The init_module() which is called when the module is inserted into the kernel. It should return 0 on success and a negative value on failure.
  - The cleanup_module() which is called just before the module is removed.
- Typically, init_module() either registers a handler for something with the kernel, or it replaces one of the kernel function with its own code (usually code to do something and then call the original function). The cleanup_module() function is supposed to undo whatever init_module() did, so the module can be unloaded safely.
- Similarly, for removing the module, you can use the 'rmmod' command :
- $ rmmod module

Kernel Modules (2)

- Kernel modules are inserted and unloaded dynamically
  - Kernel code extensibility at run time
  - insmod / lsmod/ rmmod commands. Look at /proc/modules
  - Kernel and servers can detect and install them automatically, for example, cardmgr (pc card services manager)
- Modules execute in kernel space
  - Access to kernel resources (memory, I/O ports) and global variables (look at /proc/ksyms)
  - May export their own visible variables
  - Can implement new kernel services (new system calls, policies) or low level drivers (new devices, mechanisms)
  - Use internal kernel basic interface and can interact with other modules (pcmcia memory_cs uses generic card services module)
  - Need to implement init_module and cleanup_module entry points, and specific subsystem functions (open, read, write, close, ioctl …)

Hello World Module (1)

- Modules define two special functions
  - One to be invoked when the module is loaded (`hello_init`)
  - And one for when the module is removed (`hello_exit`)
- Kernel macros (`module_init` and `module_exit`) used to indicate the role of these two functions
- The `printk` function is defined in the Linux kernel and behaves similarly to the standard C library function `printf`
- The module can call `printk` because, after insomd has loaded it, the module is linked to the kernel and can access the kernel's public symbols (functions and variables)

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");

    return 0;
}
static int hello_exit(void)
{
  printk("<1>Goodbye cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Compiling and Loading Modules

- Create a makefile
  - Must be invoked within the context of the larger kernel build system
  - Example:

```
obj-m := hello.o

$(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
```

- Load the module

```
% make
make[1]: Entering directory `/usr/src/linux-2.6.10'
  CC [M]  /home/ldd3/src/misc-modules/hello.o
  Building modules, stage 2.
  MODPOST
  CC      /home/ldd3/src/misc-modules/hello.mod.o
  LD [M]  /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

Kernel Modules Versus Applications

- Modules are event-driven
  - Every kernel module registers itself in order to serve future requests
  - It's initialization function terminates immediately
  - Exit function of a module must carefully undo everything the init function built up
- User-level applications can call functions they don't define
  - Linking stage resolves external references using libraries
- Module is linked only to the kernel, the only functions it can call are the ones exported by the kernel
  - No libraries to link to
  - Example: printk is the version of printf defined within the kernel and exported to the modules
- Don't include typical user-level header files (like <stdio.h>, etc.)
  - Only functions that are actually part of the kernel may be used in kernel modules
  - Anything related to the kernel is declared in headers found in the kernel source tree

User Space Versus Kernel Space

- Module runs in *kernel space*, whereas applications run in *user space*
- Unix transfers execution from user space to kernel space whenever an application issues a system call or is suspended by a hardware interrupt
- Role of a module is to extend kernel functionality
  - Some functions in the module are executed as part of system calls
  - Some are in charge of interrupt handling
- Linux driver code must be reentrant
  - Must be capable of running in more than one context at the same time
  - Must avoid race conditions

    => Linux 2.6 is a preemptive kernel!

Character Drivers

- Character devices are accessed through names in the filesystem
  - Those names are called *special files* or *device files*
    - *conventionally located in the `/dev` directory*
    - *Identified by a "`c`" in the first column of the output of "`ls -l`"*
    - *Block devices are identified by "`b`"*
- User applications access a char device by
  1. Opening it
  2. Reading from/writing to it
  3. Closing it

Example

- For example, a user application may use a terminal char device in the following way

```
int fd;
char cbuf;
fd=open(“/dev/tty”, R_ONLY, 0);
read(fd, &cbuf, 1);
close(fd);
```

Major and Minor Numbers

- If you issue “ls –l” you’ll see two numbers in the device file entries before the date of the last modification
  - These are the major and minor device number for the particular device
- Major number:
  - Identifies the driver associated with the device
    - For example, `/dev/null` and `/dev/zero` are both managed by driver 1
- Minor number:
  - Used by the kernel to determine exactly which device is being referred to
    - Example: differentiate between different partitions on an ide disk /dev/hda (i.e., /dev/hda1, /dev/hda2 all have the same major, but different minor numbers)
    - Could be used internally by the driver as an index into a local array of devices

```
crw-rw-rw-    1 root     root       1,   3 Apr 11  2002 null
crw-------    1 root     root      10,   1 Apr 11  2002 psaux
crw-------    1 root     root       4,   1 Oct 28 03:04 tty1
crw-rw-rw-    1 root     tty        4,  64 Apr 11  2002 ttyS0
crw-rw----    1 root     uucp       4,  65 Apr 11  2002 ttyS1
crw--w----    1 vcsa     tty        7,   1 Apr 11  2002 vcs1
crw--w----    1 vcsa     tty        7, 129 Apr 11  2002 vcsa1
crw-rw-rw-    1 root     root       1,   5 Apr 11  2002 zero
```

Internal Representation of Device Numbers

- Within the kernel the data type dev_t (defined in <linux/types.h>) is used to hold device numbers
  - It's a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number
  - Macros are used to translate between minor/major numbers and the internal dev_t data type

`MKDEV(int major, int minor);` // gives the internal representation
`MAJOR(dev_t dev);` // gives the major number as an `int`
`MINOR(dev_t dev);` // gives the minor number as an `int`

- `cat /proc/devices`
  - gives the major/minor numbers for the existing devices
  - If you need one, you could pick one that is not currently in use
- `mknod <device_name> [b][c] major minor`
  - Creates a new device file
  - Use b for creating a block device and c for creating a character device

Allocating and Freeing Device Numbers

- In order to set up a char device, you will need to obtain one or more device numbers to work with
- The kernel function for this task is

```
int register_chrdev_region(dev_t first, unsigned int
   count, char *name);
```

  - `first:` beginning device number of the range you would like to allocate
  - `count:` total number of contiguous device numbers that you need
  - `name:` name of the device (will appear in `/proc/devices`)
  - Return value:
    - 0 if successful
    - In case of error, a negative error code will be returned
- You should free device numbers when no longer in use:

```
void unregister_chardev_region(dev_t first, unsigned int
   count)
```

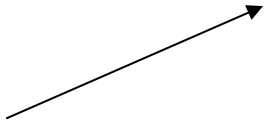
Some Important Data Structures

- Some data structures that are of interest to device driver programmers
- File Operations Structure
  - In order for a driver to turn a module into a device driver module, it needs to implement the `file_operations` structure (defined in `<linux/fs.h>`)
  - It defines a set of functions that the VFS calls when an application accesses a device node (device file)
    - Each VFS file object representing a char device file contains a pointer to this structure
  - These operations are in charge of implementing the system calls and are therefore named *open, read*, and so on.
  - Each member of the structure must point to the function in the driver that implements a specific operation (or be left NULL for unsupported operations)
  - See chapter 3 in “Linux Device Drivers, J. Corbet, A. Rubini, G. Kroah-Hartman, Third Edition, February 2005” for a complete list of members

File Operations Structure
`struct file_operations`

- Example:

```
struct file_operations my_fops = {
   .owner    = THIS_MODULE,
   .llseek   = my_llseek,
   .read     = my_read,
   .write    = my_write,
   .ioctl    = my_ioctl,
   .open     = my_open,
   .release  = my_release,
};
```



These are pointers to the functions that a driver needs to implement; if an application uses the `read()` system call on a char device node, the OS will eventually call `my_read()`

- Uses standard C tagged structure initialization syntax
  - (`.struct_member = my_elem`)
  - Allows reordering of structure members
  - makes it more portable

File Structure (1)
`struct file`

- You already know if from the VFS (file objects)!
- Defined in `<linux/fs.h>`
- Represents a file opened by an application
  - Created by the kernel on open
  - Removed on the last close

File Structure (2)
`struct file`

- The most important members of `struct file` for device driver programmers are:

`loff_t f_ops;`	Current reading/writing position
`unsigned int flags;`	File flags, such as O_RDONLY, O_NONBLOCK, etc.
`struct file_operations *fops`	Operations associated with the file
`Void *private_data;`	This is where drivers can store all kinds of internal device data

The Inode Structure (`struct inode`)

- VFS inode object internally represents a file
  - 1 to 1 relationship between files and VFS inode objects
  - Multiple `file` structures may point to a single inode structure
- Members of interest for drivers:

`dev_t i_rdev;`	For inodes that represent device files, this field contains the actual device number
`struct cdev *i_cdev;`	`struct cdev` is the kernals internal structure to represent char devices

Char Device Registration (1)

- `struct cdev` represents the char device internally
- Before kernel can invoke your device's operations, you must allocate and register one or more of these structures
- Include `<linux/cdev.h>`
- Allocating a cdev structure:

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
```

Char Device Registration (2)

- Registration with the kernel

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

  - `dev` is the cdev structure
  - `num` is the first device number to which the device responds
  - `count` is the number of device numbers that should be associated with the device
- Remove a char device from the kernel

```
int cdev_del(struct cdev *dev);
```

Char Device Registration (3)

- Example (from the Rubini Book):

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
    int err, devno = MKDEV(scull_major, scull_minor + index);
    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Fail gracefully if need be */
    if (err)
      printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

Open and Release
(from struct file_operations)

- Prototype for the `open` method:

```
int (*open)(struct inode *inode, struct file *filp);
```

- In most drivers, `open` should perform the following tasks:
  - Check for device-specific errors
  - Initialize the device if it is being opened for the first time
  - Update the `f_op` pointer, if necessary
  - Allocate and fill any data structure to be put in `file`→`private_data`
- `Release method:`
  - Reverse of `open`
  - Deallocate everything that open allocated in `file->private_data`
  - Shut down the device on last close
- Example (minimal release method):

```
int scull_release(struct inode *inode, struct file *filp)
{
  return 0;
}
```

Kernel Memory Allocation

- How can we allocate/deallocate kernel memory?
  - void *kmalloc(size_t size, int flags);
  - void kfree(*ptr);
- A call to `kmalloc` attempts to allocate `size` bytes of memory
- Return value is pointer to this memory or `NULL`
- Flags argument used to describe how the memory should be allocated
  - You should use `GFP_KERNEL` for now

Read and Write Methods (1)

- Read() and write() copy data from and to application code
- Prototypes:

```
ssize_t read(struct file *filp, char __user *buff,
  size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char __user
  *buff, size_t count, loff_t *offp);
```

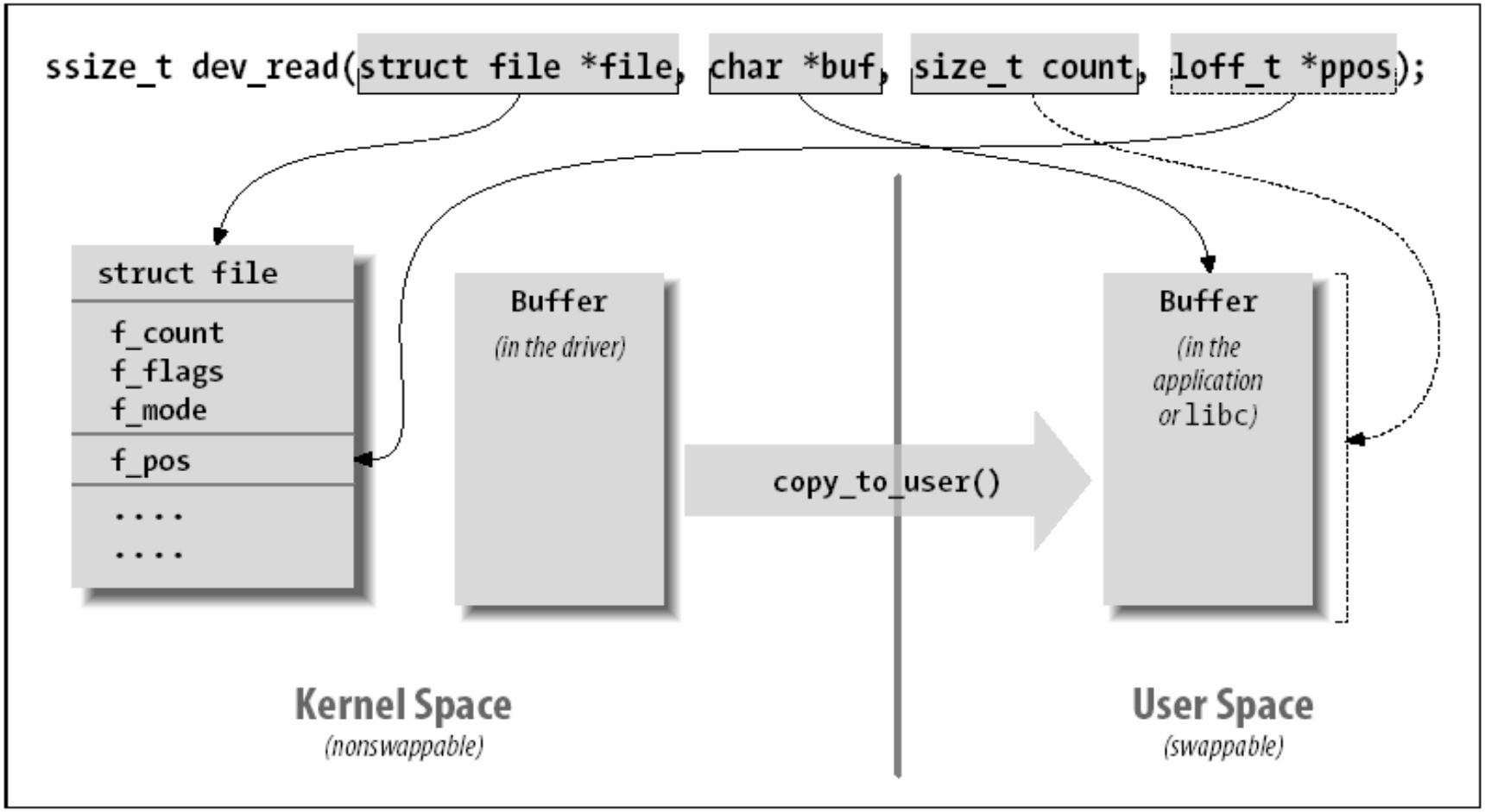
- `Filp:` File pointer
- `Count:` Size of the requested data transfer
- `Buff:` Points to the user buffer holding the data to be writtenor the empty buffer where the newly read data should be placed
- `Offp`: offset into the file

Read and Write Methods (2)

- Use caution when dealing with user/kernel data transfers
  - User space pointers may be invalid
  - May cause page fault in kernel mode (-> not allowed)
  - Pointer may be buggy or malicious
- Special functions provided to securely transfer data between user and kernel space:

```
unsigned long copy_to_user(void __user *to,
  const void *from, unsigned long count);
unsigned long copy_from_user(void *to, const
  void __user *from, unsigned long count);
```

Read and Write Methods (3)



Read Method

- The return value for *read* is interpreted by the calling application program:
  - If the value equals the count argument passed to the *read* system call, the requested number of bytes has been transferred. This is the optimal case.
  - If the value is positive, but smaller than count, only part of the data has been
  - transferred. Most often, the application program retries the read. For instance, if you read using the *fread* function, the library function reissues the system call until completion of the requested data transfer.
  - If the value is 0, end-of-file was reached (and no data was read).
  - A negative value means there was an error. The value specifies what the error was, according to <*linux/errno.h*>. Typical values returned on error include –EINTR (interrupted system call) or -EFAULT (bad address).

Writing Linux Device Driver

Header files

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/config.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/fs.h>
```

Writing Linux Device Driver
Initialize register

```
static int test_open(struct inode *inodePtr, struct file *fp)
{
    printk( "Enter open routine.\n" );
    return 0;
}

static int test_release(struct inode *inodePtr, struct file *fp)
{
    MOD_DEC_USE_COUNT;
    printk( "Enter test_release\n" );
    return 0;
}
```

Writing Linux Device Driver
Operation

```
static ssize_t test_read(struct file *fp, char *buf, size_t len, loff_t *offset)
{
    printk( "Enter test_read\n" );
    return len;
}

static ssize_t test_write(struct file *fp, char *buf, size_t len, loff_t *offset)
{
    printk( "Enter test_write\n" );
    return len;
}
```

Writing Linux Device Driver

Mapping

```
struct file_operations test_fops =
{
    read:           test_read,
    write:          test_write,
    open:           test_open,
    release:        test_release,
    ioctl:          test_ioctl,
};
```

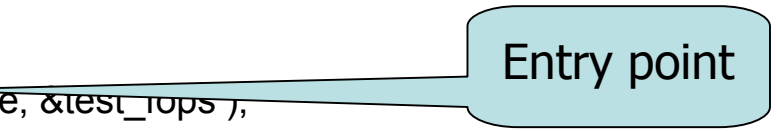
Writing Linux Device Driver

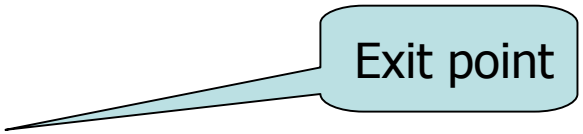
```
static int test_ioctl(struct inode *inodePtr, struct file
   *fp, unsigned int cmd, unsigned long arg)
{
   switch( cmd ){
        default:
                break;
   }
   return 0;
}
```

Writing Linux Device Driver

Entry point V.S. Exit point

```
int test_init_module(void)
{
    register_chrdev( 100, testName, &test_fops );

    printk( "Init test module\n" );
    return 0;
}

void test_cleanup_module(void)
{
    unregister_chrdev( 100, testName );

    printk( "UnInit test\n" );
}

module_init ( test_init_module );
module_exit ( test_cleanup_module );
```





Create Environment

- Compiler
  - gcc -D__KERNEL__ -DMODULE -I/usr/src/linux-2.4.22/include -c -O -Wall drv.c -o drv.o
- Make node
  - mknod /dev/test c ***100*** 0
- Check node

crw-rw-rw- 1 root dialout 4, 64 Jun 30 11:19 test

crw-rw-rw- 1 root dialout 4, 65 Aug 16 00:00 ttyS1

Writing Application

```
int main(int argc, char *argv[])
{
    int fd;
    char buf[256] = "Hello World!!";

    fd = open( "/dev/test", O_RDWR );
    if ( fd == -1 )
    {
        printf( "Open drv driver fail.[Application]\n" );
        return -1;
    }
    write( fd, buf, 6 );
    close( fd );
    return 0;
}
```

References

- The Linux Document Project
  - http://www.tldp.org
- Advanced linux programming
  - http://www.advancedlinuxprogramming.com/alp-folder
- Linux device drivers
  - http://www.xml.com/ldd/chapter/book/
- Linux Kernel Programming
  - by Michael Beck (Author), Harald Bohme (Author)