EASY ARM

Device Drivers Programming

Kernel Modules

Each piece of code that can be added to the kernel at runtime is called a 'module'. The linux kernel offers support for quiet a few different types of modules, including, but not limited to device drivers.

A module is 'object code' (not linked to a complete executable) that can be dynamically linked to the running kernel and can be unlinked.

Kernel Modules

Uses of Modules

- Allow third party commercial development of drivers
- Minimize kernel size
- Allow smaller generic kernels

Kernel Modules

Modutils: insmod; rmmod; ksyms; lsmod; modinfo; modprobe; depmod; kerneld;

insmod →
- Installs a loadable module in the running kernel.
- Load time parameters can be passed to the module, to customize its operation (eg. I/O ports, IRQ Nos etc)

rmmod →tries to unload a (set of) module(s) from the running kernel with the restriction that they are not in use and that they are not referred to by other modules.

lsmod ⟶ Shows information about all loaded modules. (name, size, use count, list of referring modules etc..)

Kernel Modules

modinfo → Displays information about a kernel module (from object file)

modprobe → Uses a "makefile"-like dependency file (created by 'depmod') to automatically load the relevant modules from the set of modules available in predefined directory trees.

- Used to load a module, either a single module, a stack of dependent modules.
- Will try to load a module out of a list and stops loading as soon as one module loads successfully.

depmod → Makes a "Makefile"-like dependency file, which is later used by modprobe.

First Module Writing

hello module – initialization and cleanup

```
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
static int __init hello_init(void)
{
   printk(KERN_ALERT “Hello World\n");
   return 0;
}
static void __exit hello_exit(void)
{
   printk(KERN_ALERT “Good Bye!\n");
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR(“B Vasu Dev”);
```

__init:
removed after initialization (static kernel or module).
__exit: discarded when module compiled statically into the kernel.

Kernel Modules

Compiling kernel modules

Kernel modules need to be compiled with certain gcc options to make them work.

The 2.6 compatible Makefile for the module testmod.ko

```
obj-m := testmod.o
```

A simple command line for building a module

#make –C /usr/src/linux-2.6.1 SUBDIRS=$PWD modules

assumes, your module's source code and Makefile are located at the same directory.

Compiling a module

The below Makefile should be reusable for any Linux 2.6 module.
Just run make to build the hello.ko file
Caution: make sure there is a [Tab] character at the beginning of the $(MAKE) line (make syntax)

either
-Full kernel sourcedirectory (configured and compiled)
-or just kernel headers dir (minimum needed)

```
# Makefile for the hello module
obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
	$(MAKE) –C $(KDIR) SUBDIRS=$(PWD) modules
```

[Tab]!
(no spaces)

Kernel Modules

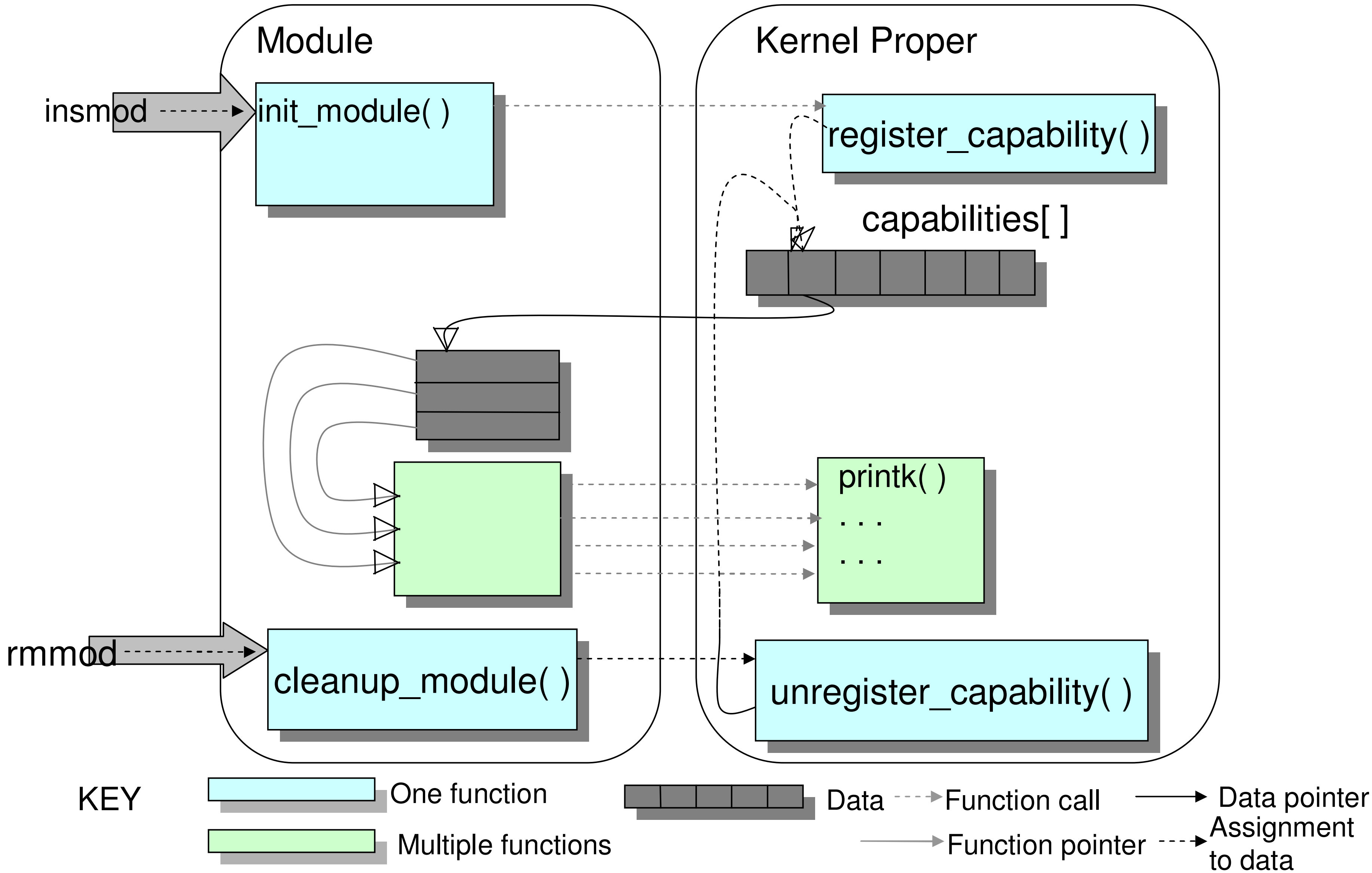
Running module

You can test the module by calling insmod and rmmod (Note that only superuser can load and unload a module).

root# insmod ./hello.o
Hello, World
root# rmmod hello
Goodbye
#

- You can verify the loading of modules using lsmod.
- The messages appear only on the console, not on xconsole.
- Take a look at /var/log/messages just to see that it got logged to your system logfile.

Linking a module to the kernel



Application vs Module

- An application performs a single task from beginning to end.
- A module registers itself in order to serve future requests and its "main" function i.e., init_module terminates immediately.
- The second entry point of a module, "cleanup_module" gets invoked just before the module is unloaded by rmmod etc.
- A module is linked only to the kernel and the only functions it can call are the ones exported by the kernel. A module is not linked to any standard library like libc etc.

Usage Count

The system keeps a usage count for every module in order to determine whether the module can be safely removed. For example you can remove a filesystem type while the filesystem is mounted.

To work with the usage count, use these three macros:

MOD_INC_USE_COUNT
Increments the count for the current module
MOD_DEC_USE_COUNT
Decrements the count
MOD_IN_USE
Evaluates to true if the count is not zero

Module Configuration Parameters

Several parameters that a driver needs to know can change from system to system.

Parameter values can be assigned at load time by insmod. The command accepts the specification of integer and string values on the command line. Thus, if your module were to provide an integer parameter called ival and a string parameter called sval, the parameters could be set at module load time with an insmod command like:

```
insmod  hello.o ival=200 sval=”Welcome”
```

Kernel Modules

However, before insmod can change module parameters, the module must make them available. Parameters are declared with the MODULE_PARM macro, which is defined in module.h. MODULE_PARM takes two parameters: the name of the variable and a string describing its type. The macro should be placed outside of any function and is typically found near the head of the source file . The two parameters mentioned earlier could be declared with the following lines:

```
int ival=0;
    char *sval;

MODULE_PARM(ival, "i");
MODULE_PARM(sval, "s");
```

```c
/* module2.c */
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>

int ival=0;
char *sval;
module_param(ival, int, S_IRUGO);
module_param(sval, charp, S_IRUGO);

int my_init(void)                              //module entry point
{
  printk("#=== Module Initialized ===#\n");      //kernel print function
  printk("ival = %d \nval = %s\n", ival, sval);    //kernel print function
  return 0;
}
void my_exit(void)                             //cleanup function
{
  printk("#=== Module removed ===#\n");
}
module_init(my_init);
module_exit(my_exit);
MODULE_LICENSE("GPL");
```

Exporting Symbols

A module implements its own functionality without the need to export any symbols. You will need to export symbols, whenever other modules may benefit from using them. You may also need to include specific instructions to avoid exporting all non-static symbols, as most versions of modutils export all of them by default.
The Linux kernel header files provide a convenient way to manage the visibility of your symbols, thus reducing namespace pollution and promoting proper information hiding.
If your module exports no symbols at all, you might want to make that explicit by placing a line with this macro call in source file:

```
EXPORT_NO_SYMBOLS;
```

Module Implementation

Modules are stored in the filesystem as ELF object files. Module is loaded into RAM by /sbin/insmod. The kernel allocates memory area containing the following data:

- A module object
- A null-terminated string that represents the name of the module.
- The code that implements the functions of the module.

The module object describes a module. A simple linked list collects all module objects. The first element of the list is addressed by the module_list variable.

Module Object

size_of_struct	size of module object
next	next list element
name	pointer to module name
size	module size
uc.usecount	module usage counter
flags	module flags
nsyms	number of exported symbols
ndeps	number of referenced modules
syms	table of exported symbols
deps	list of referenced modules
refs	list of referencing modules
init	initialization method
cleanup	cleanup method
ex_table_start	start of exception table
ex_table_end	end of exception table

Linking and unlinking modules

A user can link a module into running kernel by executing /sbin/insmod:

1. Reads the name of module from the command line
2. Locates the file containing module's object code
3. Computes the size of memory area needed
4. Invokes the create_module() system call
5. Invokes the query_module() system call
6. Using the kernel symbol table, relocates object code included in the module's file
7. Allocates a memory area in the user mode address space, loads it with a copy of the module object. Sets init and cleanup fields.

1. Invokes the init_module()
2. Releases the user mode memory area and terminates.

Linking and unlinking modules…

A user can unlink a module from running kernel by executing /sbin/rmmod:

1. Reads the name of module from the command line
2. Invokes the query_module() system call to get list of linked modules.
3. Invokes the query_module() system call to retrieve dependency information.
4. Invokes the delete_module() system call, paasing the module’s name.

Device Drivers

Device Drivers

Need for Device Driver

For the new hardware device, for which driver is not available, you are required to write device driver.

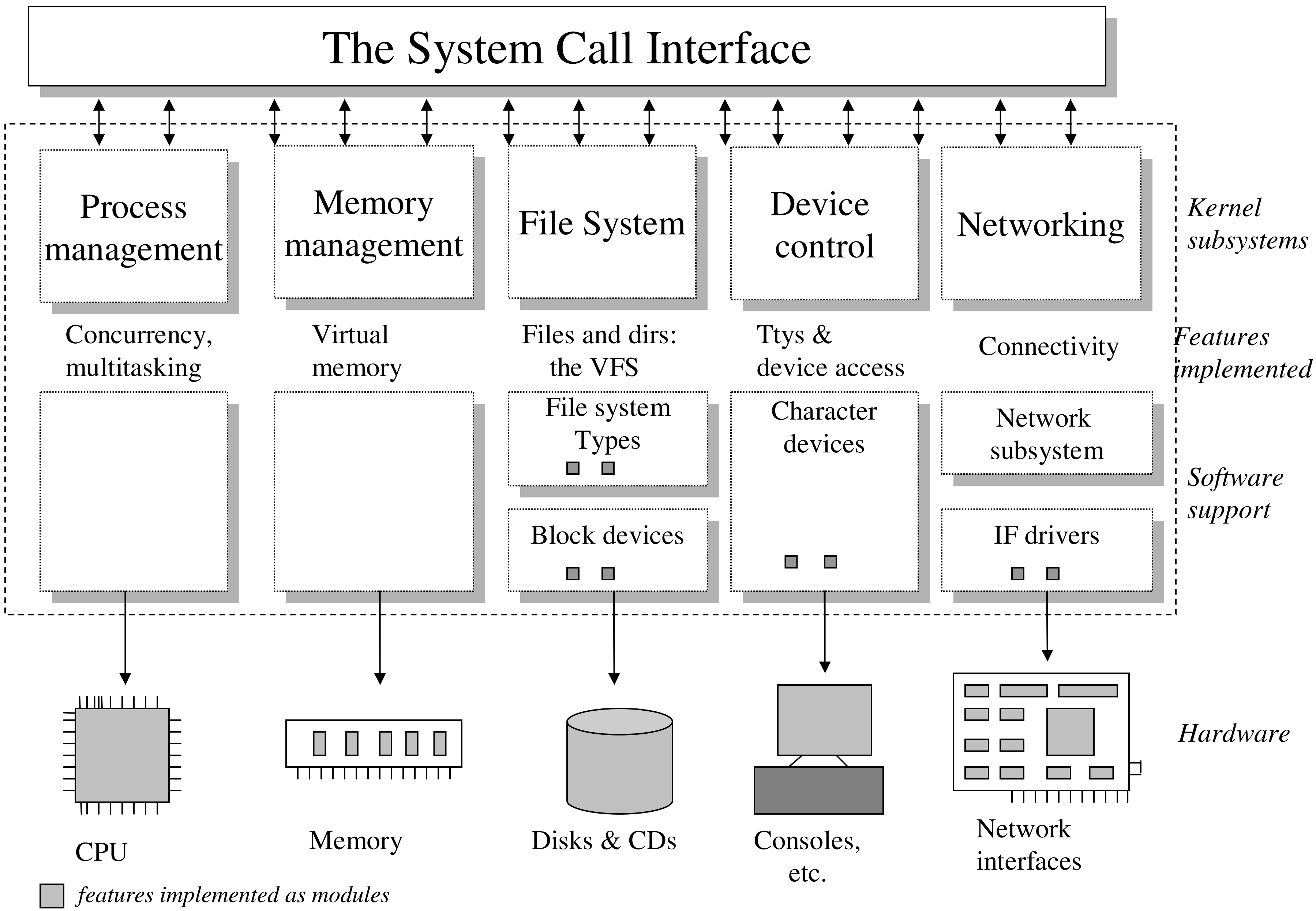
You may like to add additional capability to existing device driver. (Filter Driver)

Device Drivers

They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface ; they hide completely the details of how the device works.

Linux kernel's role is split into various tasks. The following slide shows the position of device driver. For each type of device, the device driver directly interacts with the kernel. A driver usually implements at least the open, close, read and write systems calls.

A split view of the kernel



Classes of Devices

There are mainly three classes of devices:

- Character Devices (Console, serial ports, parallel port etc.)
- Block Devices (HDD, FDD, CD-ROM etc..)
- Network Devices (NIC etc)

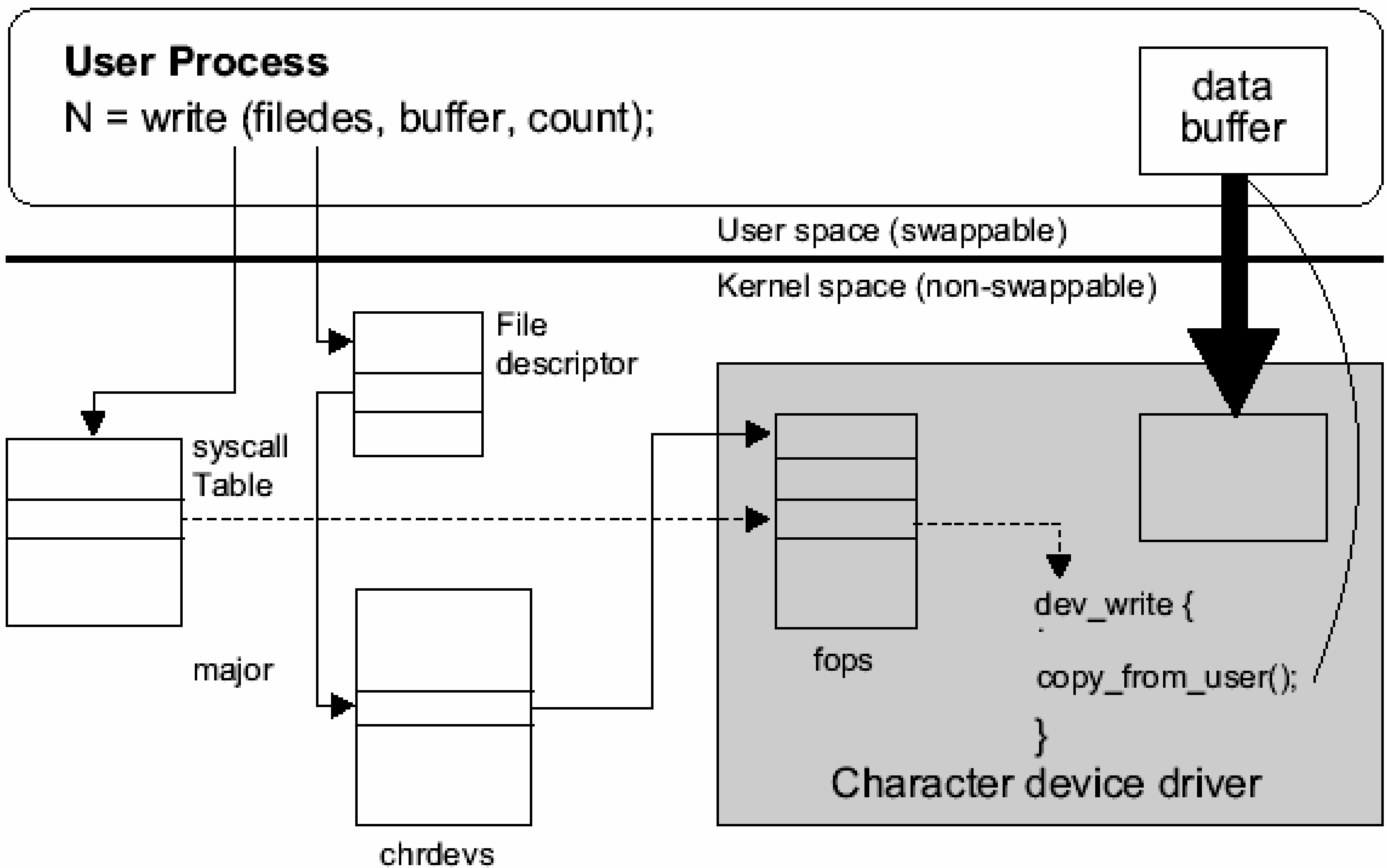
The Driver and the filesystem

From the user's side of view the driver looks like an ordinary file. If you operate on this file via open, close, read or write requests the kernel looks up the appropriate functions in your driver code. All drivers provide a set of routines. Each device has a struct char_fops that holds the pointers to this routine. At init time of the driver this struct is hooked into another table where the kernel can find it. The index that is used to dereference this set of routines is called MAJOR number and is unique so that a definite distinction is possible. The special inode file for the driver gets its MAJOR number at creation time via the mknod command. The driver code gets its MAJOR by the register_chrdev kernel routine.

Driver Processing – Write

The user process calls write(). This invokes the kernel through an INT instruction where the write operation is conveyed as an index into the syscall[] table. The filedes argument contains, among other things, a major device number so that the kernel's write function can access the driver's file_operations structure. The kernel calls the driver's write function, which copies the User Space buffer into Kernel Space.

Driver Processing – Write



The general look of a Driver

The structure of a driver is similar for each peripheral device:

- You have an init routine that is for initializing your hardware, perhaps setting memory from the kernel and hooking your driver-routines into the kernel.
- You have a char_fops struct that is initialized with those routines that you will provide for your device. This struct is the key to the kernel it is 'registered' by the register_chrdev routine.
- Mostly you have open and release routines that are called whenever you perform a open or close on your special inode.
- You can have routines for reading and writing data from or to your driver, a ioctl routine that can perform special commands to your driver like config requests or options.
- You have the possibility to readout the kernel environment string to configure your driver via lilo.
- An interrupt routine can be registered if your hardware support this.

Compile your Driver into kernel code

The most drivers in Linux are linked to the kernel at compile time. That means if you want to add a driver you have to put your .c and .h files directly somewhere in the kernel source path and rebuild the kernel.

Dynamically loaded drivers

The kernel module can be loaded into the kernel at runtime. That means loaded and removed at any time after the boot process. The only differences between a loadable Module and a kernel linked driver are a special init() routine that is called when the module is loaded into the kernel and a cleanup routine that is called when the module is removed.

Character Device Drivers

Character Drivers

Compile your Driver into kernel code

The most drivers in Linux are linked to the kernel at compile time. That means if you want to add a driver you have to put your .c and .h files directly somewhere in the kernel source path and rebuild the kernel.

Dynamically loaded drivers

The kernel module can be loaded into the kernel at runtime. That means loaded and removed at any time after the boot process. The only differences between a loadable Module and a kernel linked driver are a special init() routine that is called when the module is loaded into the kernel and a cleanup routine that is called when the module is removed.

DRIVER REGISTRATION :

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

int unregister_chrdev(unsigned int major, const char* name);
```

Important File : /proc/devices

Char Device Registration (new way)

To allocate the devices

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

Two ways to define cdev structure (represents the char device)

```
- struct cdev *my_cdev = cdev_alloc();
   my_cdev->ops = &my_fops;
- void cdev_init(struct cdev *dev, struct file_operations *fops);
```

To add a char device

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

To free the devices

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

To remove a char device from the system:

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

Character Drivers

Major & Minor Numbers:

- Char devices are accessed through names in the file system called special devices or device files or nodes.
- Conventionally located in /dev directory .
- Identified by ‘c’ in the first column of ls -l.
- Two comma separated numbers in file size field denote major and minor.
- major number identifies the driver associated with the device.
- minor number is only used by the driver to differentiate several devices controlled / managed /driven by it.

Excerpt from the LINUX major numbers list

Major	Character devices	Block devices
0	*unnamed* for NFS,network and so on	
1	Memory devices (mem)	RAM disk
2		Floppy disks (fd*)
3		IDE hard disks
4	Terminals	
5	Terminals & AUX	
6	Parallel interfaces	
7	Virtual consoles (vcs*)	
8		SCSI hard disks (sd*)
9	SCSI tapes	
10	Bus mice (bm, psaux)	
11		SCSI CD-ROM
12	QIC02 tape	
13	PC speaker driver	XT 8-bit hard disks(xd*)
14	Sound cards	BIOS hard disk support
15	Joystick	Cdu31a/33a CD-ROM
16, 17, 18	*not used*	
19	Cyclades drivers	Double compressing driver
20	Cyclades drivers	
21	SCSI generic	
22		2nd IDE interface driver
23		Mitsumi CD-ROM (mcd*)
24		Sony 535 CD-ROM
25		Matsushita CD-ROM 1
26		Matsushita CD-ROM 2
27	QIC117 tape	Matsushita CD-ROM 3
28		Matsushita CD-ROM 4
29	Frame buffer drivers	Other CD-ROMs
30	iCBS2	Philips LMS-205 CD-ROM

mknod

mknod creates device nodes in file system tree.

- Syntax: mknod PATH/DEVICE   TYPE MAJOR MINOR

- Major numbers can be allocated dynamically or statically.

File_operations_structure

- Capabilities of a device (supported by a driver) are made known by filling a 'file-operations' struct and passing it to 'register_chrdev'.
- This is an array of function pointers.
- Kernel uses this 'struct' to access driver's functions.

'struct file-operations' members are :

```
loff_t  (*llseek) ( struct file*, loff_t, int );
ssize_t  (*read) ( struct  file*, char*, size_t, loff_t*);
ssize_t (*write) ( struct file*, const char *, size_t, loff_t*);
int       (*ioctl) ( struct inode*, struct file*,unsigned int,unsigned long);
                    - to handle device specific commands .

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

- after fork or dup, release will be invoked after all copies of fd are closed.

```
struct  module  *owner ;
```

… and more

Character Drivers

- Use tagged structure initialization

eg: struct file_operations my_fops = {

```
llseek   :   my_llseek,
read     :   my_read,
write    :   my_write,
ioctl    :   my_control,
open     :   my_open,
release  :   my_close,

owner    :   THIS_MODULE,
};
```

The File Structure

- C Library FILE appears in user space programs.

- Struct file is a kernel file structure that never appears in user space.
- Every open file in the system has an associated 'struct file' in kernel space, created by the kernel on 'open' and is passed to any function or method that operates on the file.

most important fields are :

```
mode_t        f_mode;
loof_t        f_pos;
unsigned int  f_flags;
struct  file_operations  *f_op;
void          *private_data;
struct        dentry  *f_dentry;
```

The inode structure

inode structure is used by the kernel to represent files. Two fields of this structure are:

```
dev_t i_rdev;
struct cdev *i_cdev;    // represents char devices.
```

Two macros to obtain the major and minor number from an inode

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

HOW TO GET DEVICE NUMBER

The combined device number resides in the field i_rdev of the inode structure.

MAJOR (kdev_t dev); Extract the major number from a kdev_t structure.

MINOR (kdev_t dev) ; Extract the minor number

MKDEV(int ma, int mi) ; create a kdev_t type from major & minor number

Data exchange between Application & DRIVER

- Application runs in user space where as driver in kernel space.
- User-space addresses cannot be used directly in kernel space.
- when the kernel accesses a user space pointer , the associated page may not be present in memory (swapped out).
- To deal with like conditions , following functions are provided:

```
unsigned long copy_from_user( void *to, const void *from,
                              unsigned long count);
unsigned long copy_to_user( void *to, const void *from,
                            unsigned long count);
#include <asm/uaccess.h>
```

access_ok

Address verification (without transferring data) is implemented by the function access_ok, which is declared in <asm/uaccess.h>

int access_ok(int type, const void *addr, unsigned long size);

type - VERIFY_READ or VERIFY_WRITE depending on the action to be performed.

addr - user address space

size - byte count

access_ok returns a boolean value: 1 for success (access is OK) and 0 for failure (access is not OK). If it returns false, the driver should usually return –EFAULT to the caller.

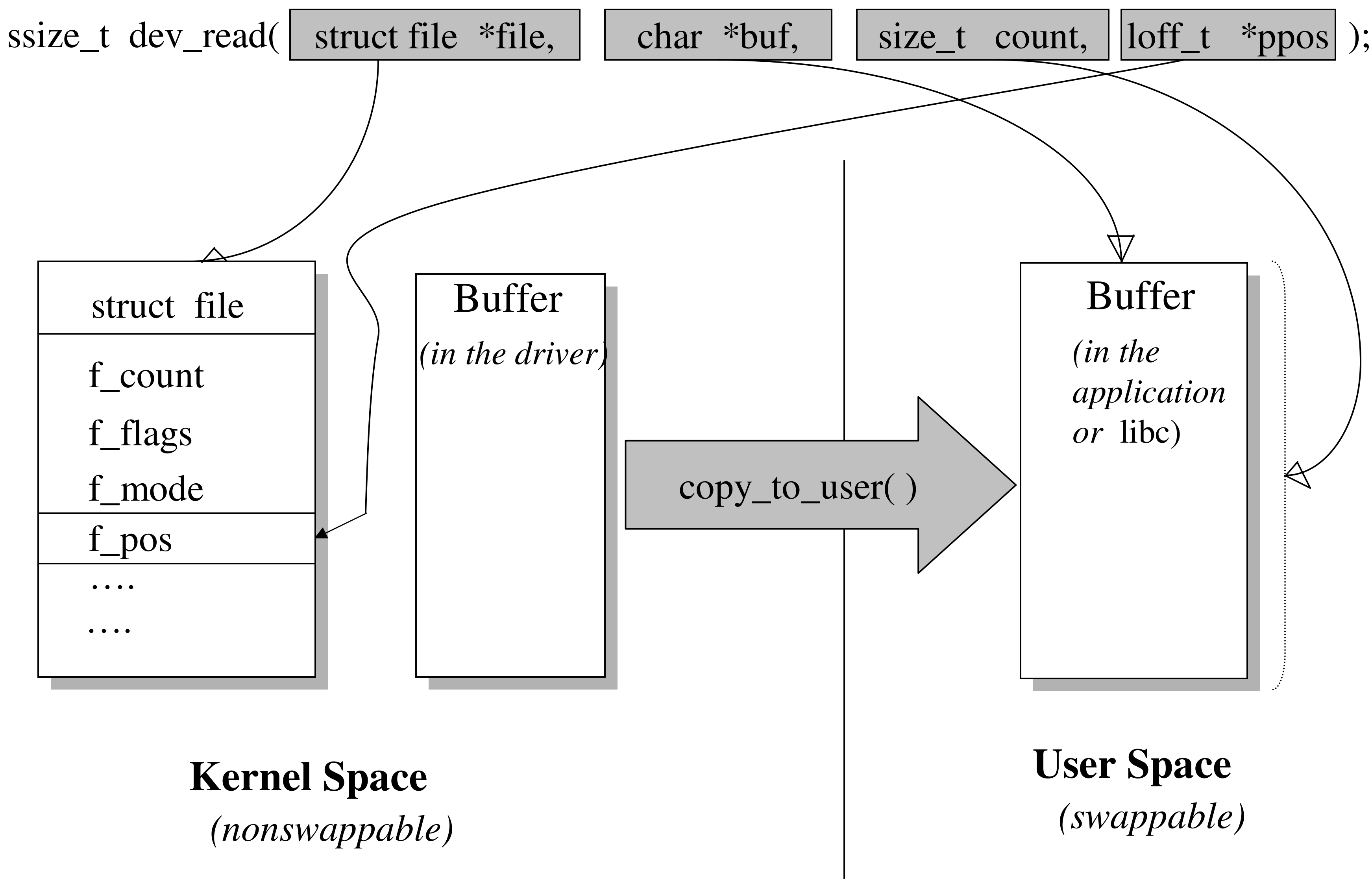
Data transfer functions

A set of functions that are optimized for the most used data sizes (one, two , four and eight bytes).

put_user(dataum, ptr);
__put_user(dataum, ptr);
These macros write the datum to user space, relatively fast, single values are being transferred.

get_user(local, ptr);
__get_user(local,ptr);
These macros are used to retrieve a single datum from user space.
Returns 0 on success and –EFAULT on error.

The arguments to read



The Open Method

In most drivers, open should perform the following tasks:

- Increment usage count
- Check for device specific errors
- Initialize the device, if it is being opened for first time
- Identify the minor number and update the f_op pointer
- Allocate and fill any data structure to be put in filp->private_data

The Release Method

The release method should perform the following tasks:

- Deallocate anything that open allocated in filp->private_data
- Shut down the device on last close
- Decrement the usage count

The Read Method

- If the value equals the count argument passed to the read system call, the requested number of bytes has been transferred.
- If the value is positive, but smaller than count, only part of the data has been transferred.
- If the value is 0 end of file is reached
- A negative value means there was error. These errors look like -EINTR (interrupted system call) or -EFAULT (bad address)

The Write Method

- If the value equals the count argument passed to the write system call, the requested number of bytes has been transferred.
- If the value is positive, but smaller than count, only part of the data has been transferred.
- If the value is 0, nothing was written.
- A negative value means there was error. These errors are defined in <linux/errno.h>.

Ioctl method

- Perform various types of hardware control via device driver (ioctl method)
- Device specific entry point for the driver to handle the “commands”
- Allows to access features unique to hardware : configuring the device or Enter / Exit operating modes
- ioctl system call : ioctl (int fd, int cmd, char *argp)

Driver method :

```
int (*ioctl) (struct inode  *inode,  struct file *filep,
              unsigned int cmd, unsigned long arg );
```

cmd is a unique number, represents the request
arg contains the input/output buffer

Choosing ioctl commands

To choose *ioctl* numbers for driver according to the Linux kernel 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.

```
#define _IOC_NRBITS    8
#define _IOC_TYPEBITS  8
#define _IOC_SIZEBITS  14
#define _IOC_DIRBITS   2
```

```
/* Direction bits. */
#define _IOC_NONE  0U
#define _IOC_WRITE 1U
#define _IOC_READ  2U
```

```
/* used to create numbers */
#define _IO(type,nr)
#define _IOR(type,nr,size)
#define _IOW(type,nr,size)
#define _IOWR(type,nr,size)
```

LED Programming

1. Identify the Hardware connection
2. Set the direct of Port Pin

   Set appropriate bit in PIOx_OER

3. Pull the Pins high to turn on the LED's

   Set bits in PIOx_SODR

4. Pull the pins low to turn off the LED's

   Set the bit in PIOA_CODR

EASY ARM

USART Programming

- Select peripheral multiplexed lines for Tx0 and Rx0
  - Set bits 4 and 5 of PIOB_PDR (PIO disable)
  - Set bits 4 and 5 of PIOB_ASR (Peripheral A select)
- Reset Tx, Rx and Status register by Setting bits Control register US0_CR
- Set Mode register US0_MR
  - Normal USART_MODE
  - MCK USCLKS
  - 8 bit character length, SYNC mode 0
  - No parity, 1 Stop bit
  - 16x Sampling
- Set baud rate by writing in to US0_BRGR
  - Value = MCK/(16*baud)
- Enable Tx and Rx in US0_CR
- Transmit by writing US0_THR
  - Check the status of TXRDY in US0_CSR

?