Linux Device Driver

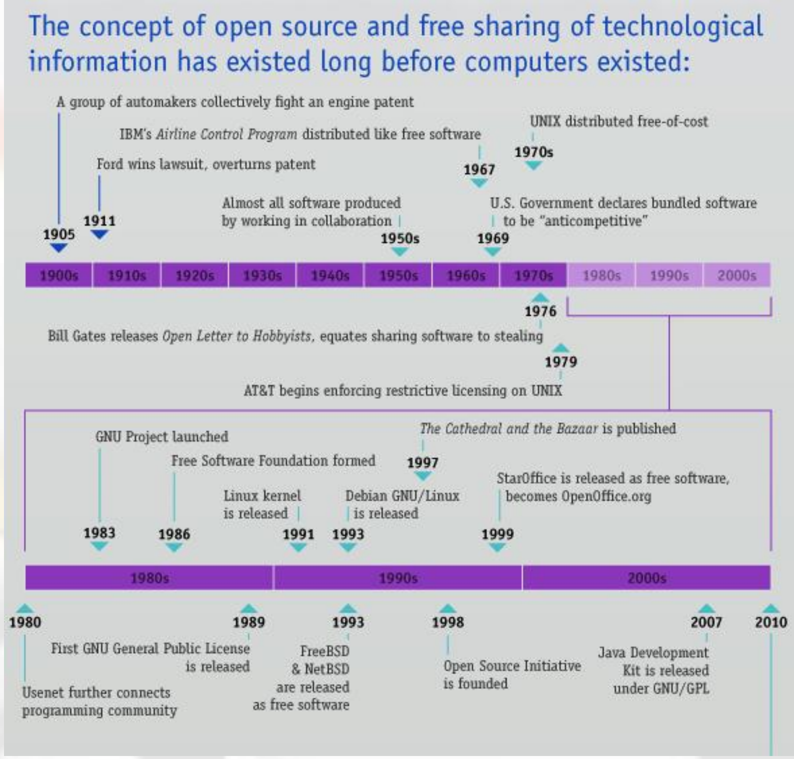
http://lwn.net/Kernel/LDD3/

2012/2013
EnsiCaen
André Lépine

Open Source Software

Open Source Software is computer software whose source code is available under a **license** (or arrangement such as the public domain) that permits users to use, change, and improve the software, and to redistribute it in modified or unmodified form.

(Source: Wikipedia)



The concept of open source and free sharing of technological information has existed long before computers existed:

1905 A group of automakers collectively fight an engine patent

1911 Ford wins lawsuit, overturns patent

1950s Almost all software produced by working in collaboration

1967 IBM's *Airline Control Program* distributed like free software

1969 U.S. Government declares bundled software to be "anticompetitive"

1970s UNIX distributed free-of-cost

1976 Bill Gates releases *Open Letter to Hobbyists*, equates sharing software to stealing

1979 AT&T begins enforcing restrictive licensing on UNIX

1983 GNU Project launched

1986 Free Software Foundation formed

1991 Linux kernel is released

1993 Debian GNU/Linux is released

1997 *The Cathedral and the Bazaar* is published

1999 StarOffice is released as free software, becomes OpenOffice.org

1980 Usenet further connects programming community

1989 First GNU General Public License is released

1993 FreeBSD & NetBSD are released as free software

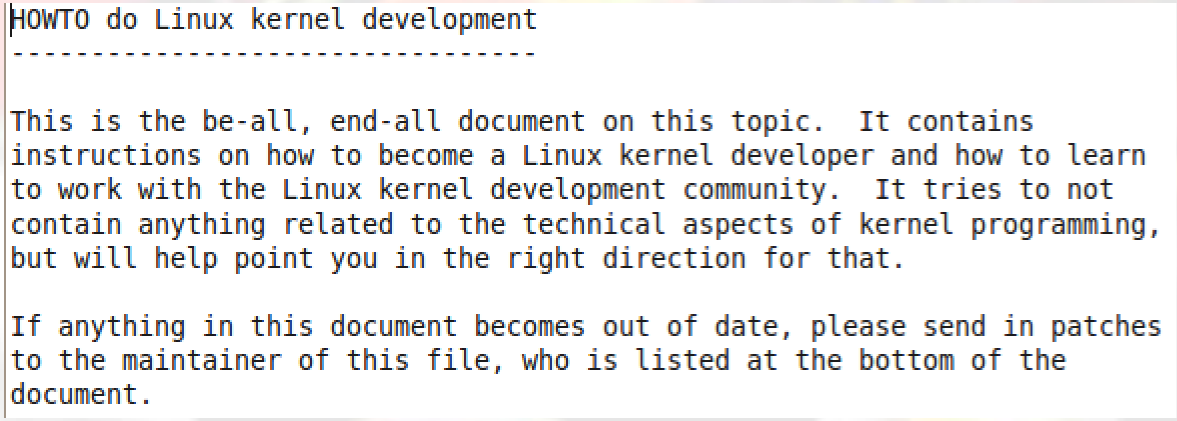
1998 Open Source Initiative is founded

2007 Java Development Kit is released under GNU/GPL

2010

References

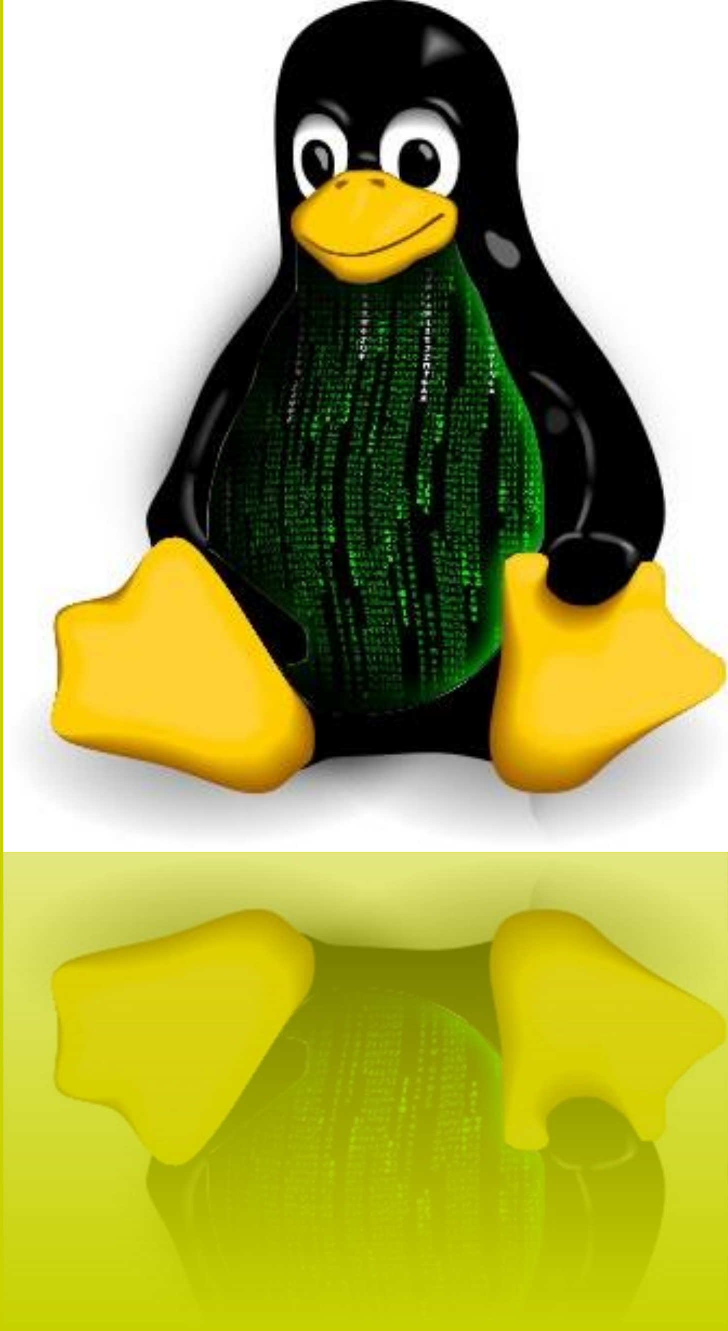
- “*Linux Device Driver, 3rd Edition*”
  - Jonathan Corbet, Alessandro, Rubini, and Greg Kroah-Hartman [O’Reilly]
- The kernel itself
  - /Documentations



```
HOWTO do Linux kernel development
---------------------------------

This is the be-all, end-all document on this topic.  It contains
instructions on how to become a Linux kernel developer and how to learn
to work with the Linux kernel development community.  It tries to not
contain anything related to the technical aspects of kernel programming,
but will help point you in the right direction for that.

If anything in this document becomes out of date, please send in patches
to the maintainer of this file, who is listed at the bottom of the
document.
```

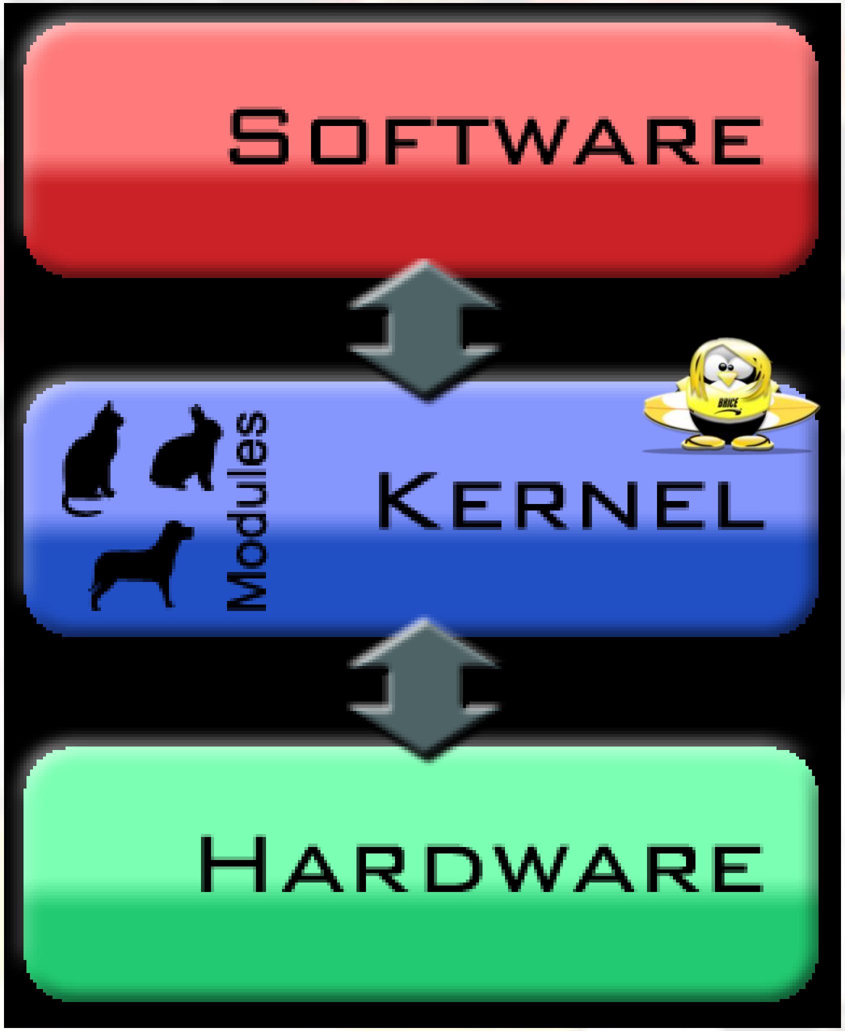
- “*The C programming language*”
  - Kernighan and Ritchie [Prentice Hall]

Zoom into the Linux Kernel

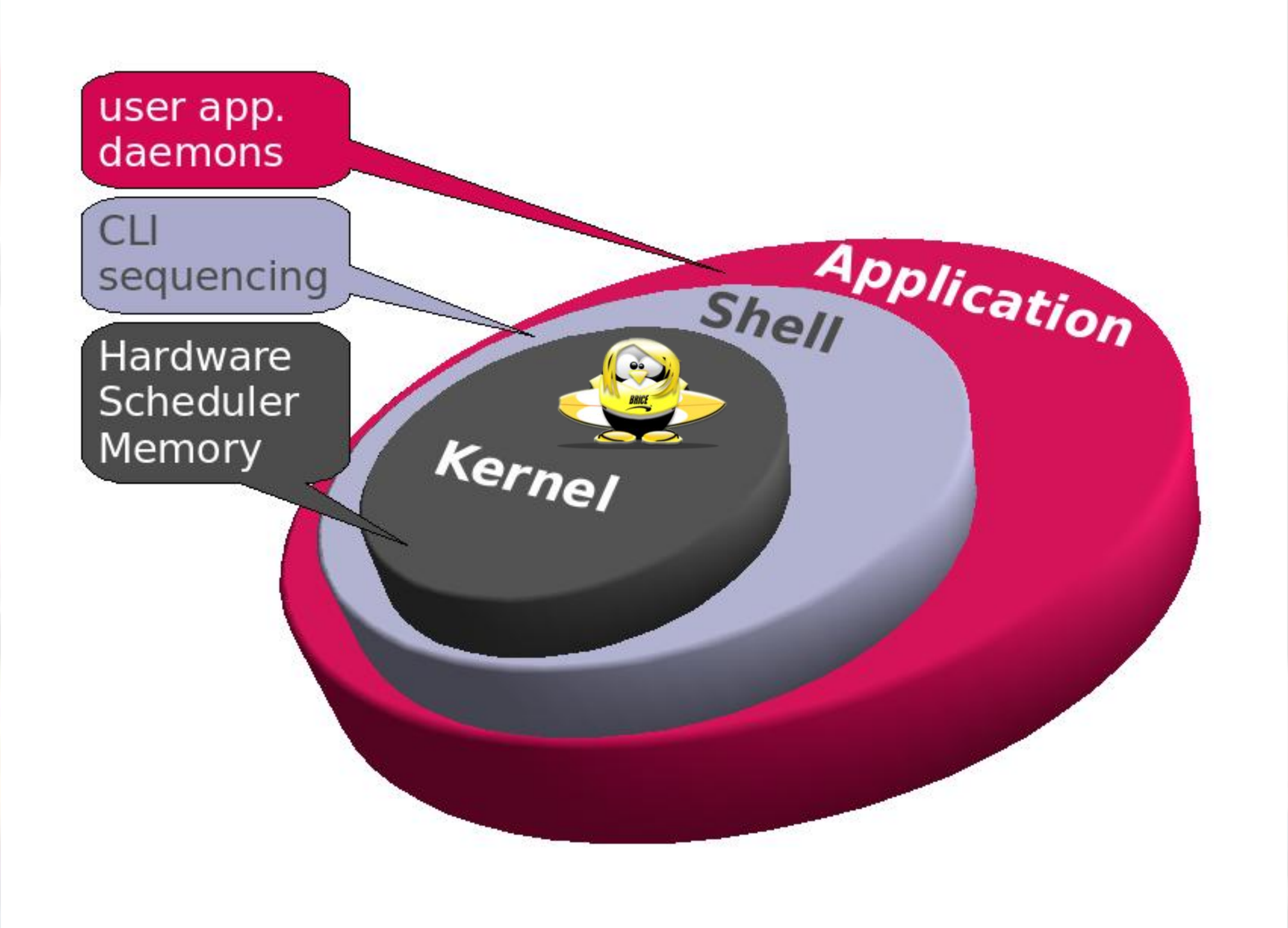


Welcome in the real word !

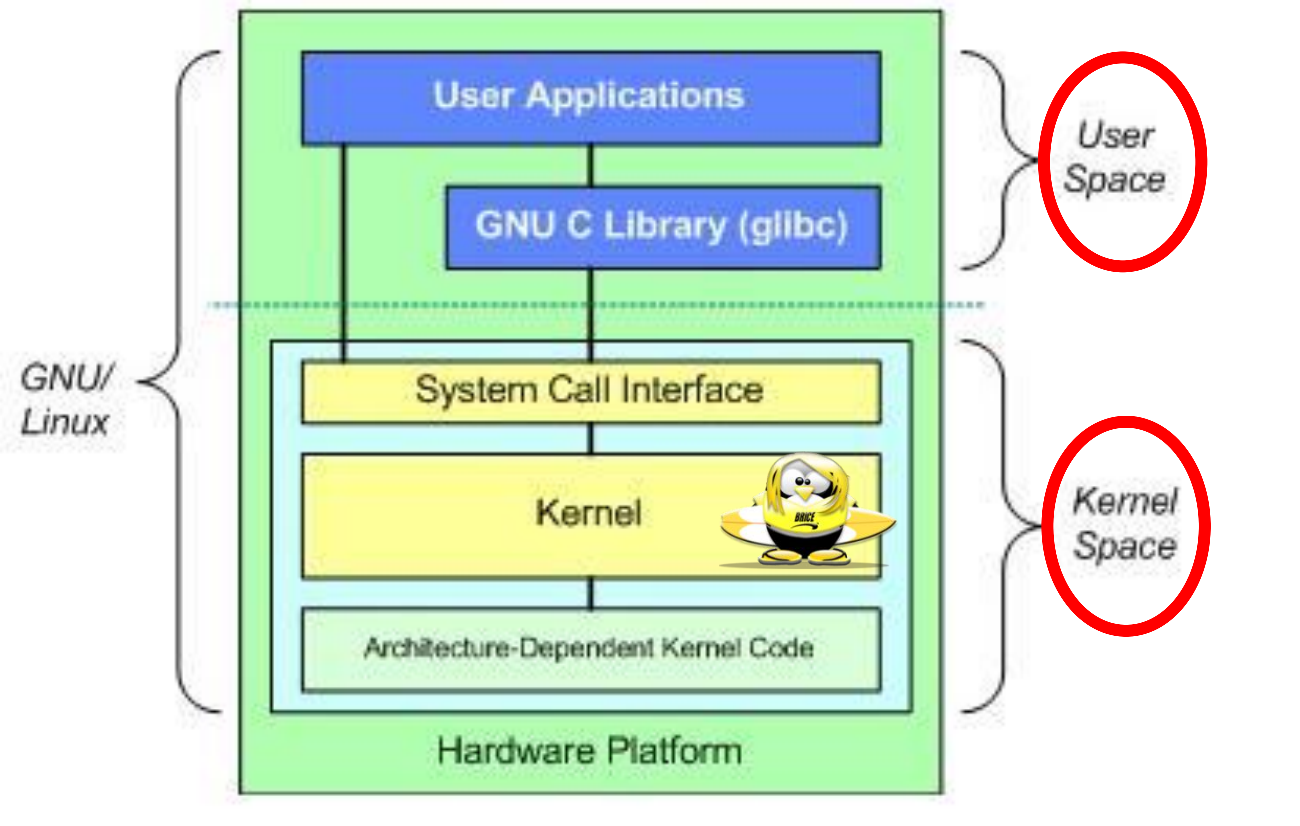
Between SW and HW



Kernel of what ?

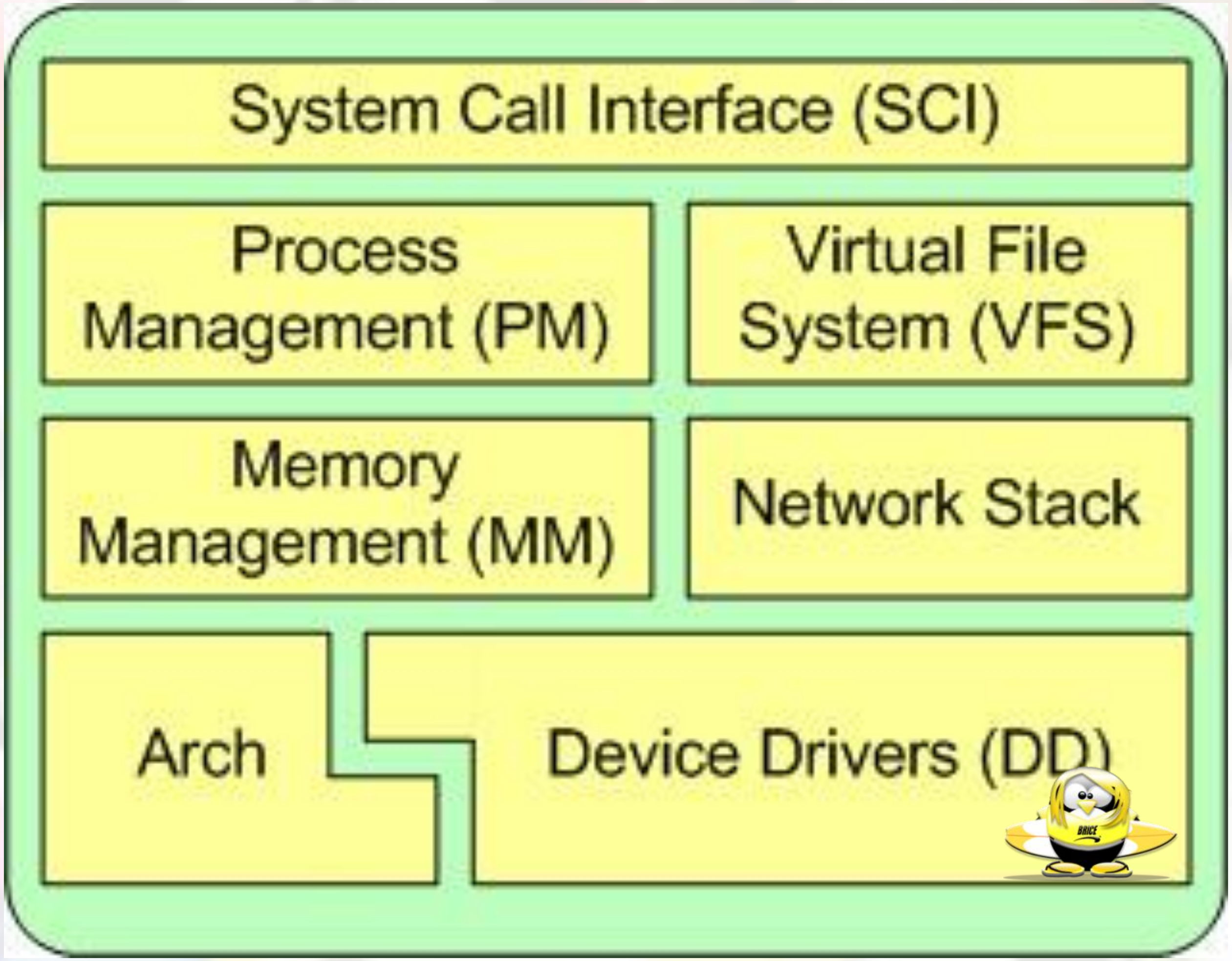


Architecture of the GNU/Linux operating system



- Where is Linux ? Linux kernel ? Android ?
  What are Ubuntu, Debian… ? Where is it ?

Device driver surrounding blocs



System Call Interface (SCI)

Process Management (PM)

Virtual File System (VFS)

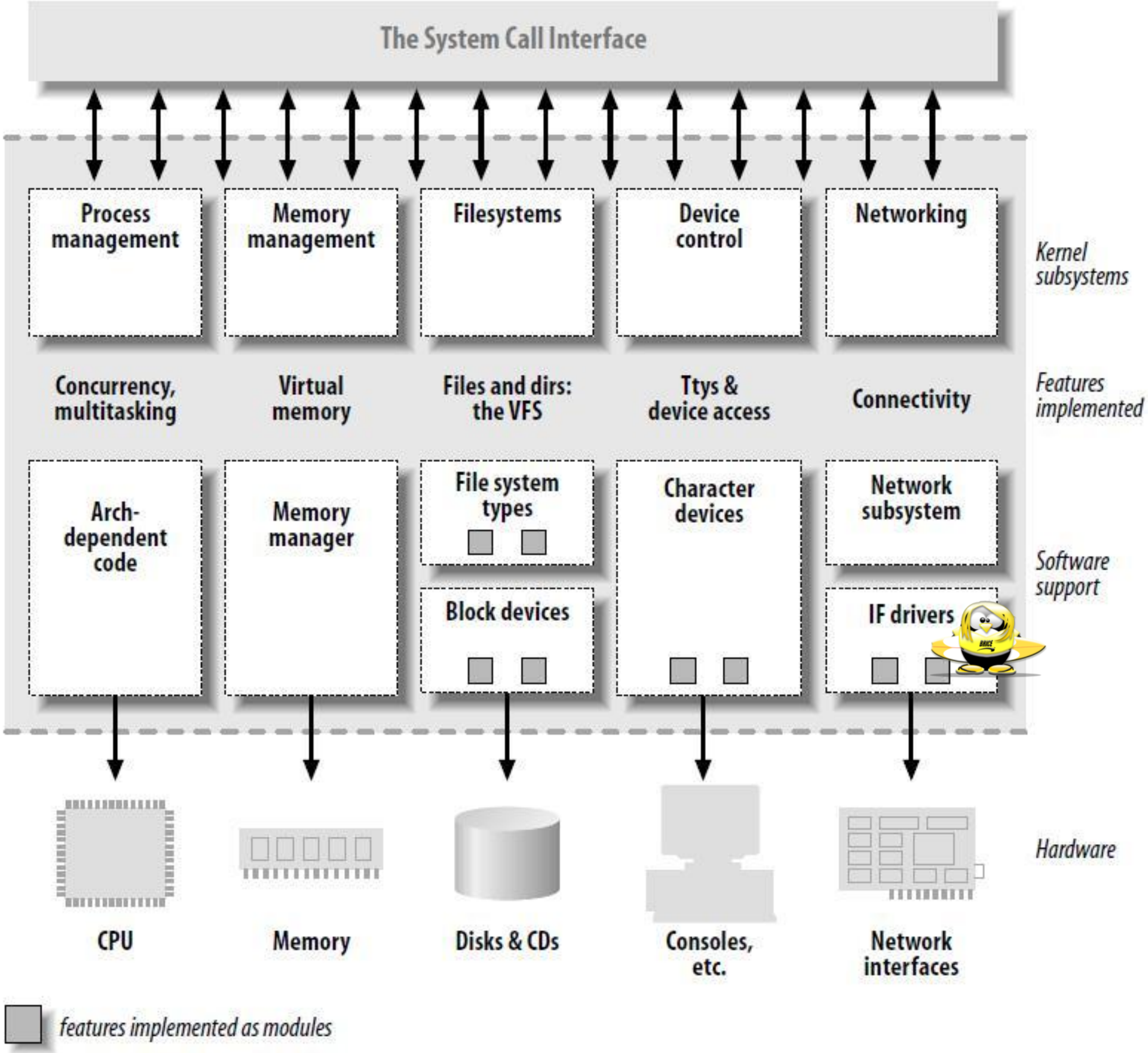
Memory Management (MM)

Network Stack

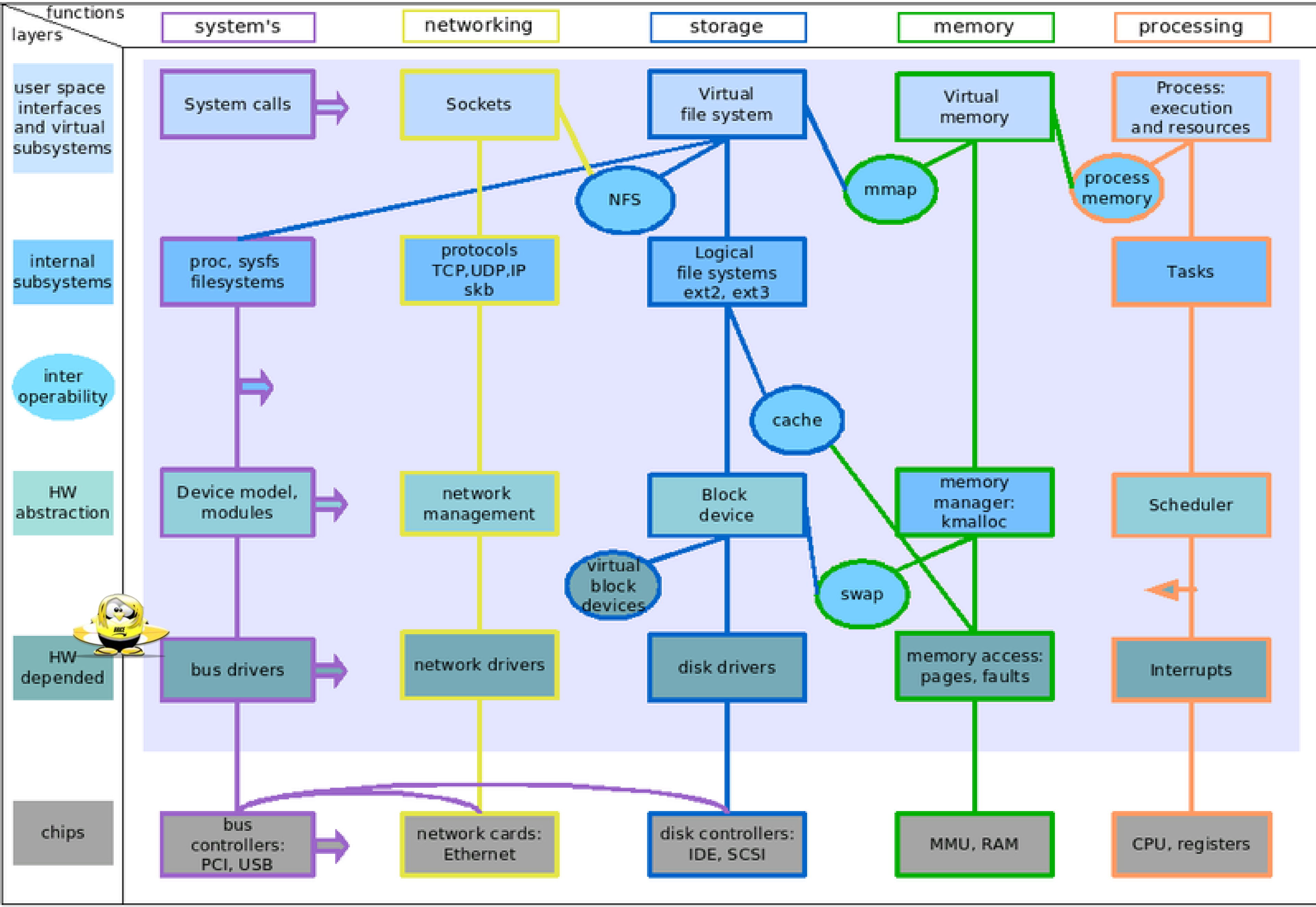
Arch

Device Drivers (DD)

Kernel


The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

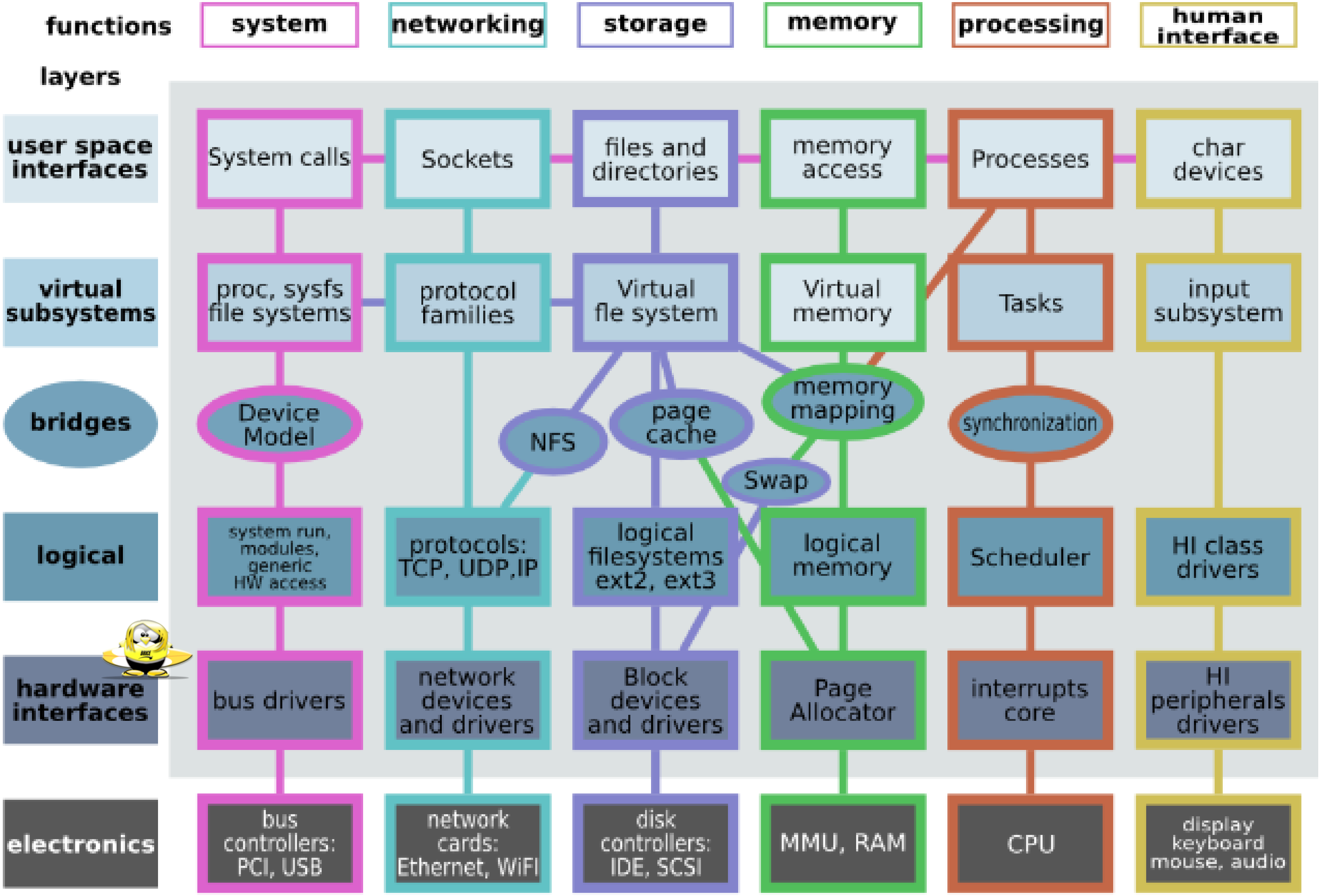
Simplified Linux kernel diagram in form of a matrix map



layers \ functions	system's	networking	storage	memory	processing
user space interfaces and virtual subsystems	System calls	Sockets	Virtual file system	Virtual memory	Process: execution and resources
			NFS	mmap	process memory
internal subsystems	proc, sysfs filesystems	protocols TCP,UDP,IP skb	Logical file systems ext2, ext3		Tasks
inter operability			cache		
HW abstraction	Device model, modules	network management	Block device	memory manager: kmalloc	Scheduler
			virtual block devices	swap	
HW depended	bus drivers	network drivers	disk drivers	memory access: pages, faults	Interrupts
chips	bus controllers: PCI, USB	network cards: Ethernet	disk controllers: IDE, SCSI	MMU, RAM	CPU, registers

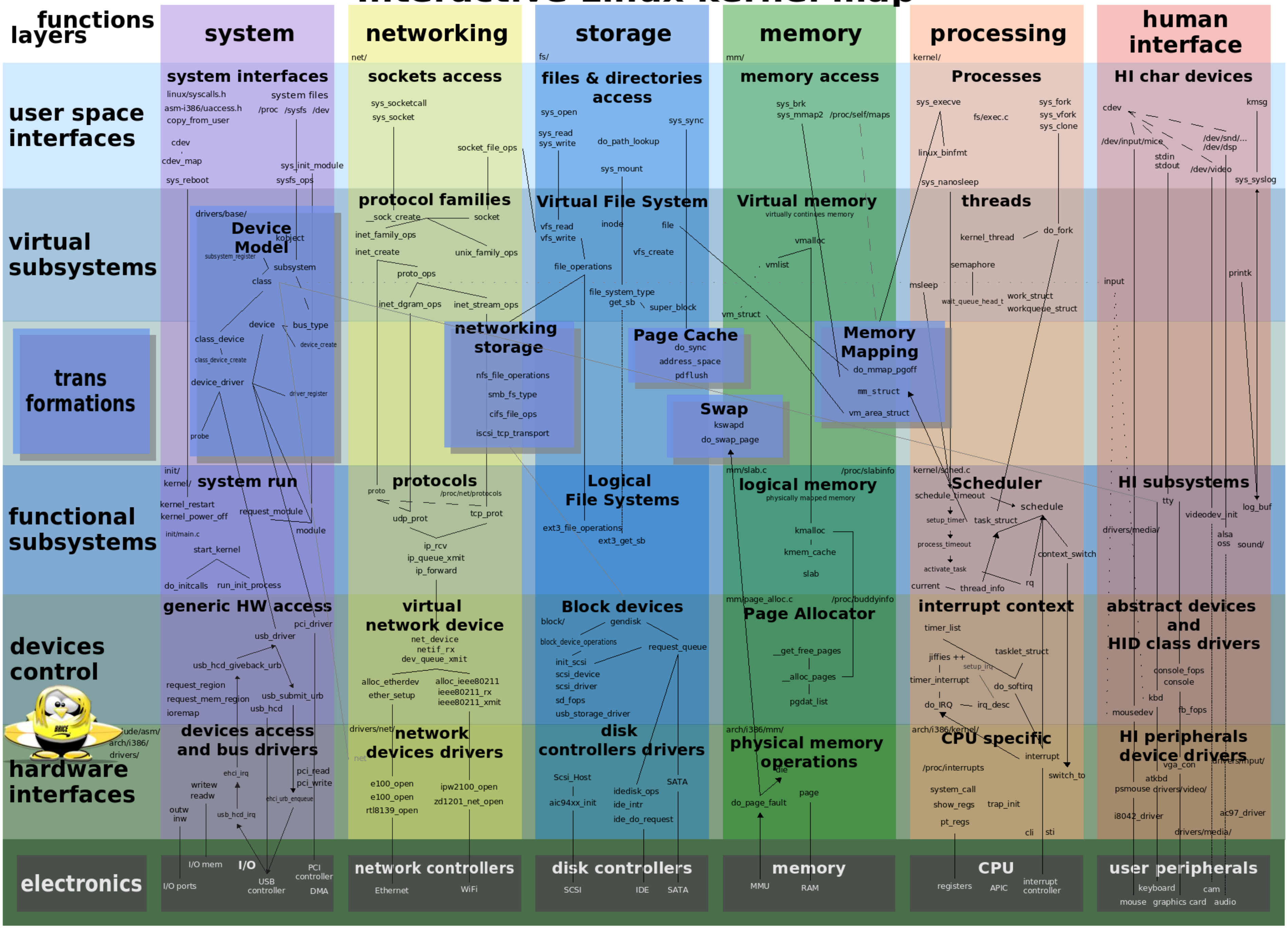
Designed with OpenOffice.org by (cc) (by-nc-sa) Constantine Shulyupin, www.linuxdriver.co.il

Linux kernel diagram



functions layers	system	networking	storage	memory	processing	human interface
user space interfaces	System calls	Sockets	files and directories	memory access	Processes	char devices
virtual subsystems	proc, sysfs file systems	protocol families	Virtual fle system	Virtual memory	Tasks	input subsystem
bridges	Device Model	NFS	page cache	memory mapping Swap	synchronization	
logical	system run, modules, generic HW access	protocols: TCP, UDP,IP	logical filesystems ext2, ext3	logical memory	Scheduler	HI class drivers
hardware interfaces	bus drivers	network devices and drivers	Block devices and drivers	Page Allocator	interrupts core	HI peripherals drivers
electronics	bus controllers: PCI, USB	network cards: Ethernet, WiFi	disk controllers: IDE, SCSI	MMU, RAM	CPU	display keyboard mouse, audio

Interactive Linux kernel map



functions layers	system	networking net/	storage fs/	memory mm/	processing kernel/	human interface
user space interfaces	**system interfaces** linux/syscalls.h asm-i386/uaccess.h copy_from_user system files /proc /sysfs /dev cdev cdev_map sys_init_module sys_reboot sysfs_ops	**sockets access** sys_socketcall sys_socket socket_file_ops	**files & directories access** sys_open sys_sync sys_read sys_write do_path_lookup sys_mount	**memory access** sys_brk sys_mmap2 /proc/self/maps	**Processes** sys_execve sys_fork sys_vfork sys_clone fs/exec.c linux_binfmt sys_nanosleep	**HI char devices** cdev kmsg /dev/input/mice /dev/snd/... /dev/dsp stdin stdout /dev/video sys_syslog
virtual subsystems	drivers/base/ **Device Model** kobject subsystem_register subsystem class	**protocol families** __sock_create socket inet_family_ops inet_create unix_family_ops proto_ops inet_dgram_ops inet_stream_ops	**Virtual File System** vfs_read vfs_write inode file vfs_create file_operations file_system_type get_sb super_block	**Virtual memory** virtually continues memory vmalloc vmlist vm_struct	**threads** kernel_thread do_fork semaphore msleep wait_queue_head_t work_struct workqueue_struct	input printk
trans formations	device bus_type class_device device_create class_device_create device_driver driver_register probe	**networking storage** nfs_file_operations smb_fs_type cifs_file_ops iscsi_tcp_transport	**Page Cache** do_sync address_space pdflush **Swap** kswapd do_swap_page	**Memory Mapping** do_mmap_pgoff mm_struct vm_area_struct		
functional subsystems	init/ kernel/ **system run** kernel_restart kernel_power_off request_module module init/main.c start_kernel do_initcalls run_init_process	**protocols** proto /proc/net/protocols udp_prot tcp_prot ip_rcv ip_queue_xmit ip_forward	**Logical File Systems** ext3_file_operations ext3_get_sb	mm/slab.c /proc/slabinfo **logical memory** physically mapped memory kmalloc kmem_cache slab	kernel/sched.c **Scheduler** schedule_timeout schedule setup_timer task_struct process_timeout context_switch activate_task rq current thread_info	**HI subsystems** tty log_buf videodev_init drivers/media/ alsa oss sound/
devices control	**generic HW access** pci_driver usb_driver usb_hcd_giveback_urb request_region request_mem_region ioremap usb_submit_urb usb_hcd	**virtual network device** net_device netif_rx dev_queue_xmit alloc_etherdev ether_setup alloc_ieee80211 ieee80211_rx ieee80211_xmit	**Block devices** block/ gendisk block_device_operations request_queue init_scsi scsi_device scsi_driver sd_fops usb_storage_driver	mm/page_alloc.c /proc/buddyinfo **Page Allocator** __get_free_pages __alloc_pages pgdat_list	**interrupt context** timer_list jiffies ++ tasklet_struct setup_irq timer_interrupt do_softirq do_IRQ irq_desc	**abstract devices and HID class drivers** console_fops console kbd mousedev fb_fops
hardware interfaces	ude/asm/ arch/i386/ drivers/ **devices access and bus drivers** ehci_irq pci_read pci_write writew readw ehci_urb_enqueue outw inw usb_hcd_irq	drivers/net/ net **network devices drivers** e100_open e100_open rtl8139_open ipw2100_open zd1201_net_open	**disk controllers drivers** Scsi_Host aic94xx_init SATA idedisk_ops ide_intr ide_do_request	arch/i386/mm/ **physical memory operations** die do_page_fault page	arch/i386/kernel/ **CPU specific** interrupt /proc/interrupts switch_to system_call show_regs trap_init pt_regs cli sti	**HI peripherals device drivers** vga_con drivers/input/ atkbd psmouse drivers/video/ i8042_driver ac97_driver drivers/media/
electronics	**I/O** I/O mem I/O ports USB controller PCI controller DMA	**network controllers** Ethernet WiFi	**disk controllers** SCSI IDE SATA	**memory** MMU RAM	**CPU** registers APIC interrupt controller	**user peripherals** keyboard mouse graphics card cam audio

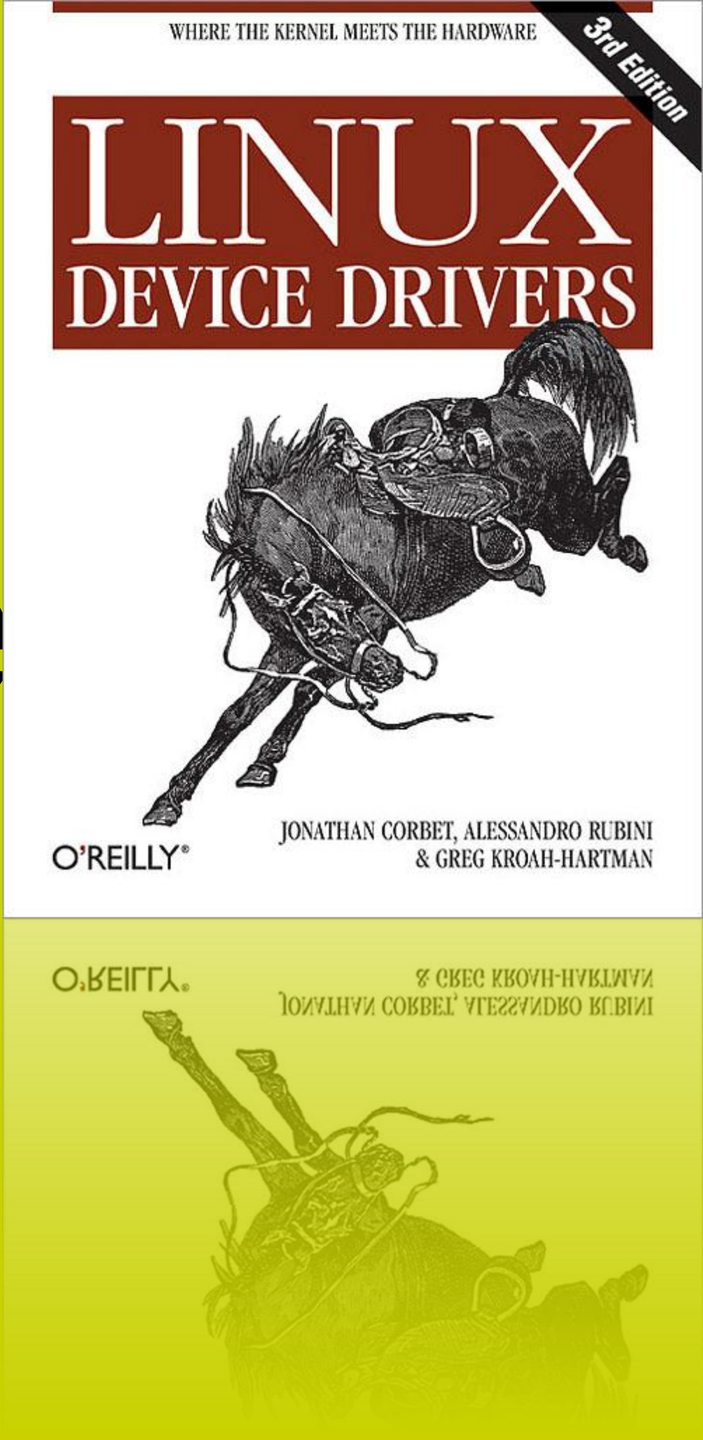
(cc) (nc) 2007 Constantine Shulyupin, www.linuxdriver.co.il/kernel_map, kernel_map@linuxdriver.co.il

Ver 0.3, 7/21/07

Are you ready ?

Linux Device Driver

Introduction


WHERE THE KERNEL MEETS THE HARDWARE
3rd Edition
LINUX
DEVICE DRIVERS
JONATHAN CORBET, ALESSANDRO RUBINI
& GREG KROAH-HARTMAN
O'REILLY®

Audience

- People who want to become kernel hackers but don’t know where to start. Give an interesting overview of the kernel implementation as well.
- Understanding the kernel internals and some of the design choices made by the Linux developers and how to write device drivers,
- Start playing with the code base and should be able to join the group of developers. Linux is still a work in progress, and there’s always a place for new programmers to jump into the game.
- You may just skip the most technical sections, and stick to the standard API used by device drivers to seamlessly integrate with the rest of the kernel.

Role of a device driver

- Flexibility
  - “what capabilities are to be provided” (the mechanism)
  - “how those capabilities can be used” (the policy)
    - The two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs.
- The driver should deal with making the hardware available, leaving all the issues about *how to use the hardware* to the applications.
- Loadable module
  - Ability to extend at runtime the set of features offered by the kernel.
  - 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*.*

3 device driver classes

- char module : stream of bytes
  - *open, close, read, and write system calls.*
  - *dev/console, /dev/tty*
- block module
  - Host a file system (like a disk)
  - handle I/O operations that transfer whole blocks (512 usually)
  - data is managed internally by the kernel
- network module
  - exchange data with other hosts, usually some hardware device
  - the kernel calls functions related to packet transmission in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted.

Modularization of the kernel

- Some types of drivers work with additional layers of kernel support functions for a given type of device.
- Examples:
  - Every USB device is driven by a USB module that works with the USB subsystem, but the device itself shows up in the system as
    - a char device (a USB serial port, say),
    - a block device (a USB memory card reader),
    - or a network device (a USB Ethernet interface).
  - The file system type is a software driver, because it maps the low-level data structures to high-level data structures.
    - Independent of the data transfer to and from the disk, which is accomplished by a block device driver.
- Kernel developers collected class-wide features and exported them to driver implementers to avoid duplicating work and bugs, thus simplifying and strengthening the process of writing such drivers.

Security issue

- Only the super-user can load module
  - System call *init_module* checks if the invoking process is authorized to load a module into the kernel
  - Security is a policy issue handled at higher levels within the kernel, under the control of the system administrator
- Exception
  - Critical resources access privilege shall be checked by the driver
- **/!\** Security bug
  - *“memory overflow”*: protect buffer handling !
  - No leakage permitted: memory obtained from the kernel should be zeroed or otherwise be initialized before being made available to a user device
- Do not run kernels compiled by an untrusted friend.

Version numbering

- Check the kernel version and interdependencies
  - you need a particular version of one package to run a particular version of another package.
  - file *Documentation/Changes* in your kernel sources is the best source of such information if you experience any problems
- The even-numbered kernel versions (i.e., 2.6.*x) are* the stable ones that are intended for general distribution
- Check *http://lwn.net/Articles/2.6-kernel-api/* for Kernel API update

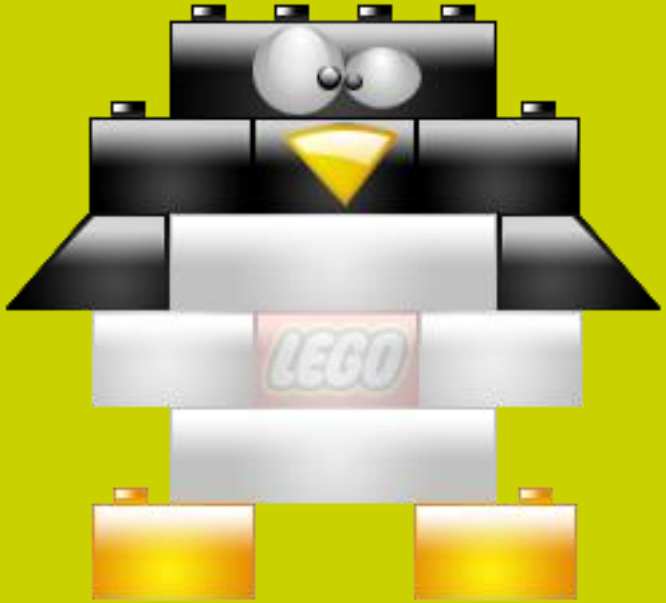
Joining the Kernel Development Community

- Larger community of developers
- Highly committed engineers working toward making Linux better
  - source of help, ideas, and critical review as well
  - first people you will likely turn to when looking for testers for a new driver
- *linux-kernel mailing* list, including Linus Torvalds
  - FAQ: http://www.tux.org/lkml
  - Linux kernel developers are busy people, and they are much more inclined to help people who have clearly done their homework first.

Module

- build and run a complete module
- basic code shared by all modules

“*Developing such expertise is an essential foundation for any kind of modularized driver*”



Hello Word module

```
#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 void hello_exit(void)
{
       printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

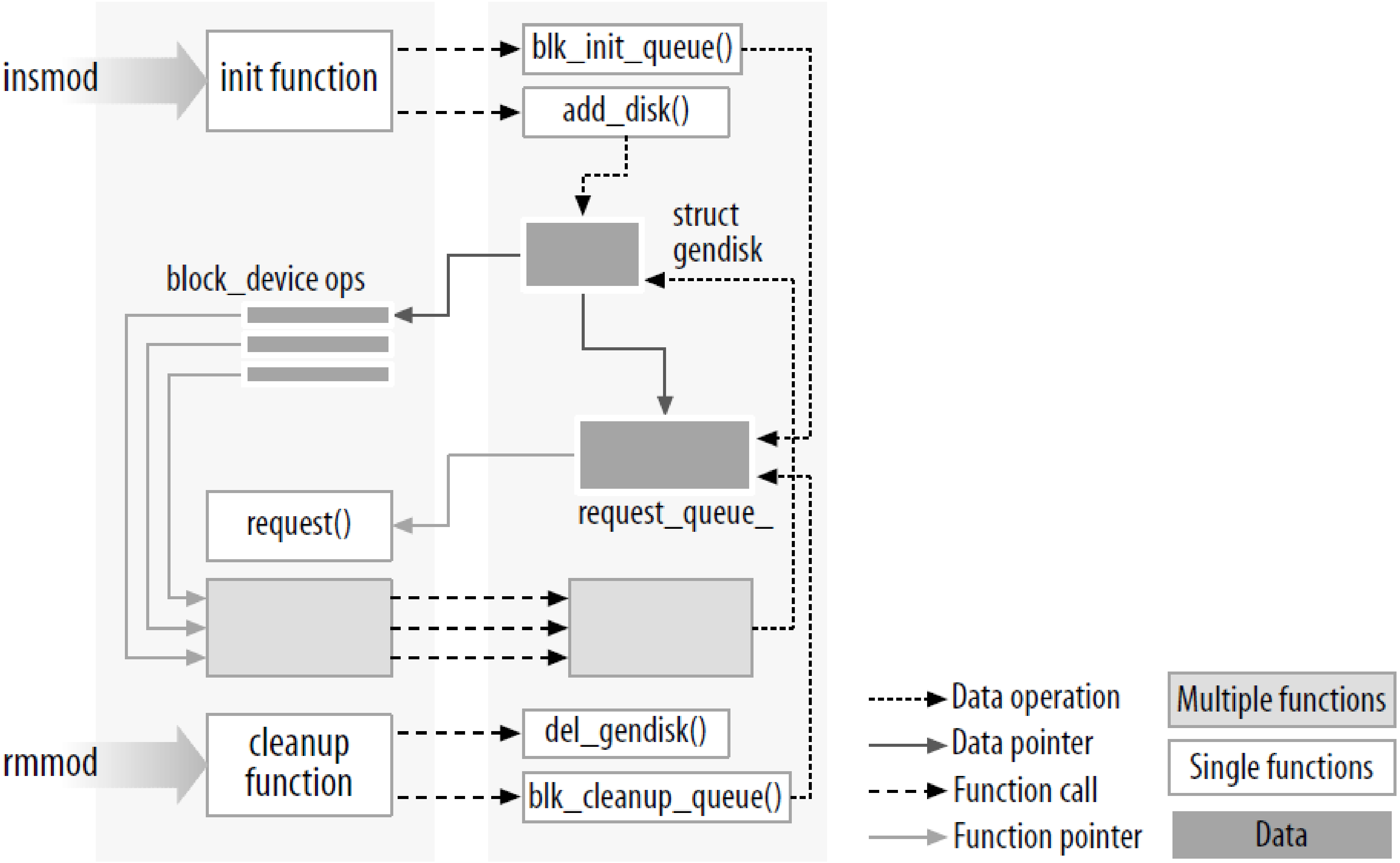
Load/unload a module

```
% 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#
```

Modularization

- Event driven programming
- The *exit* must carefully undo everything the *init* built up, or the pieces remain around until the system is rebooted
- Cost down development time: test successive version without rebooting the system each time
- A module is linked only to the kernel, and the only functions it can call are the ones exported by the kernel; there are no libraries to link to
- No debugger. A kernel fault kills the current process at least, if not the whole system

Linking a module to the kernel



User space and Kernel space

- Module runs in *kernel space,* whereas applications run in *user space*
- The kernel executes in the highest level (also called *supervisor mode*), whereas applications execute in the lowest level (the so-called *user mode*), where the processor regulates direct access to hardware and unauthorized access to memory
- Different memory mapping and different address space
- Kernel code executing a system call is working in the context of a process and is able to access data in the process's address space
- Code that handles interrupts is asynchronous and not related to any process.

Concurrency in the Kernel

- Several processes can be trying to use your driver at the same time
- Interrupt handlers run asynchronously and can be invoked at the same time that your driver is trying to do something else
- Linux can run on symmetric multiprocessor systems, with the result that your driver could be executing concurrently on more than one CPU
- 2.6, kernel code has been made preemptible
- Kernel code, including driver code, must be *reentrant*—it must be capable of running in more than one context at the same time
  - Data structures must be carefully designed to keep multiple threads of execution separate, and the code must take care to access shared data in ways that prevent corruption of the data

The current process

```
<linux/sched.h>

printk(KERN_INFO "The process is \"%s\" (pid %i)\n",
current->comm, current->pid);
```

Kernel stack is not large

- The kernel has a very small stack; as small as a single, 4096-byte page
- Large structures should be allocate dynamically at call time

Double underscore

- Function names starting with a double underscore (_ _) are low-level components and should be used with caution.

Compilation

```
% cat makefile
obj-m := module.o
module-objs := file1.o file2.o
% make -C ~/kernel-2.6 M=`pwd` modules
```

- See *Documentation/kbuild* directory in the kernel sources

Platform Dependency

- Kernel code can be optimized for a specific processor in a CPU family to get the best from the target platform
- Modern processors have introduced new capabilities:
  - Faster instructions for entering the kernel,
  - Interprocessor locking,
  - Copying data,
  - 36-bit addresses to address more than 4 GB of physical memory
- How to deliver module code
  - Distribute driver with source and scripts to compile it on the user’s system
  - Release under a GPL-compatible license, contribute to the mainline kernel

The Kernel Symbol Table

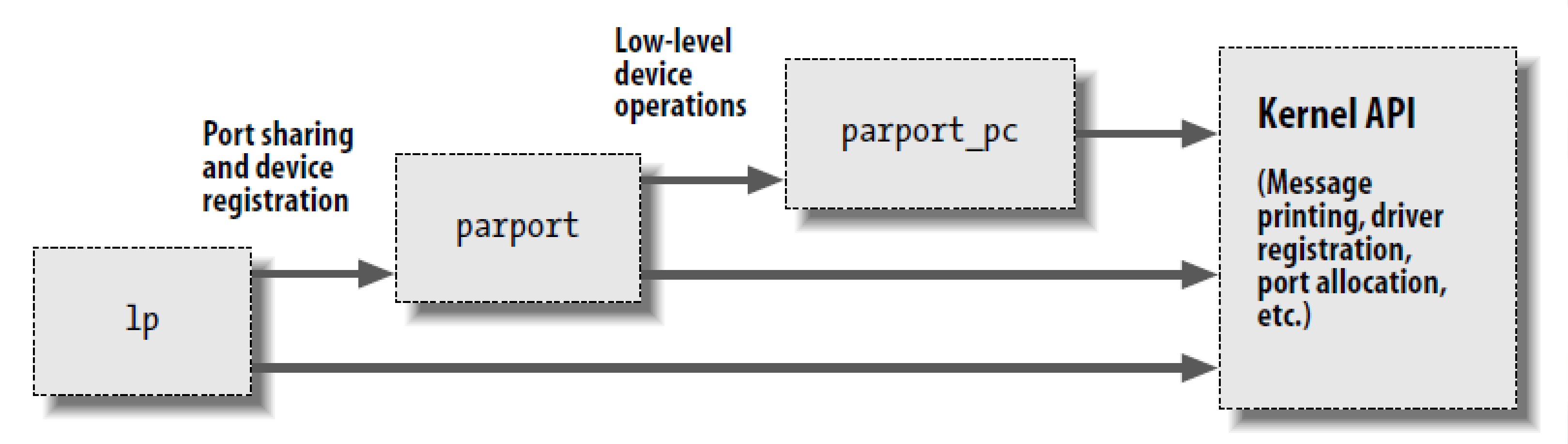
- When a module is loaded, any symbol exported by the module becomes part of the kernel symbol table
- New modules can use symbols exported and can be stack on top
  - New abstraction is implemented in the form of a device driver
  - It offers a plug for hardware-specific implementations

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_GPL(name);
```

- The _GPL version makes the symbol available to GPL-licensed modules only.
- See <*linux/module.h*>

Stacking modules

- `modprobe` functions is the same way as `insmod`, but it also loads any other modules that are required by the module you want to load
- Example:
  - Each input USB device module stacks on the *usbcore* and *input* modules


Low-level device operations
Port sharing and device registration
parport_pc
parport
lp
Kernel API
(Message printing, driver registration, port allocation, etc.)

Error Handling During Initialization

```
int __init my_init_function(void)
{
        int err;

        /* registration takes a pointer and a name */
        err = register_this(ptr1, "skull");
        if (err) goto fail_this;
        err = register_that(ptr2, "skull");
        if (err) goto fail_that;

        return 0; /* success */

fail_that: unregister_this(ptr1, "skull");
fail_this: return err; /* propagate the error */
}
```

Cleanup

```
void __exit my_cleanup_function(void)
{
        unregister_those(ptr3, "skull");
        unregister_that(ptr2, "skull");
        unregister_this(ptr1, "skull");
        return;
}
```

Module Parameters

- Values supplied during the module initialization

```
% insmod hellop howmany=10 whom="Mom"
```

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

- Values supplied as a comma-separated list

```
module_param_array(name,type,num,perm);
```

Char device

- suitable for most simple hardware devices
- easier to understand than block or network drivers
- aim is to write a *modularized* char driver

Overview

- Reuse this materials

ch3.zip

A typical */proc/devices* file

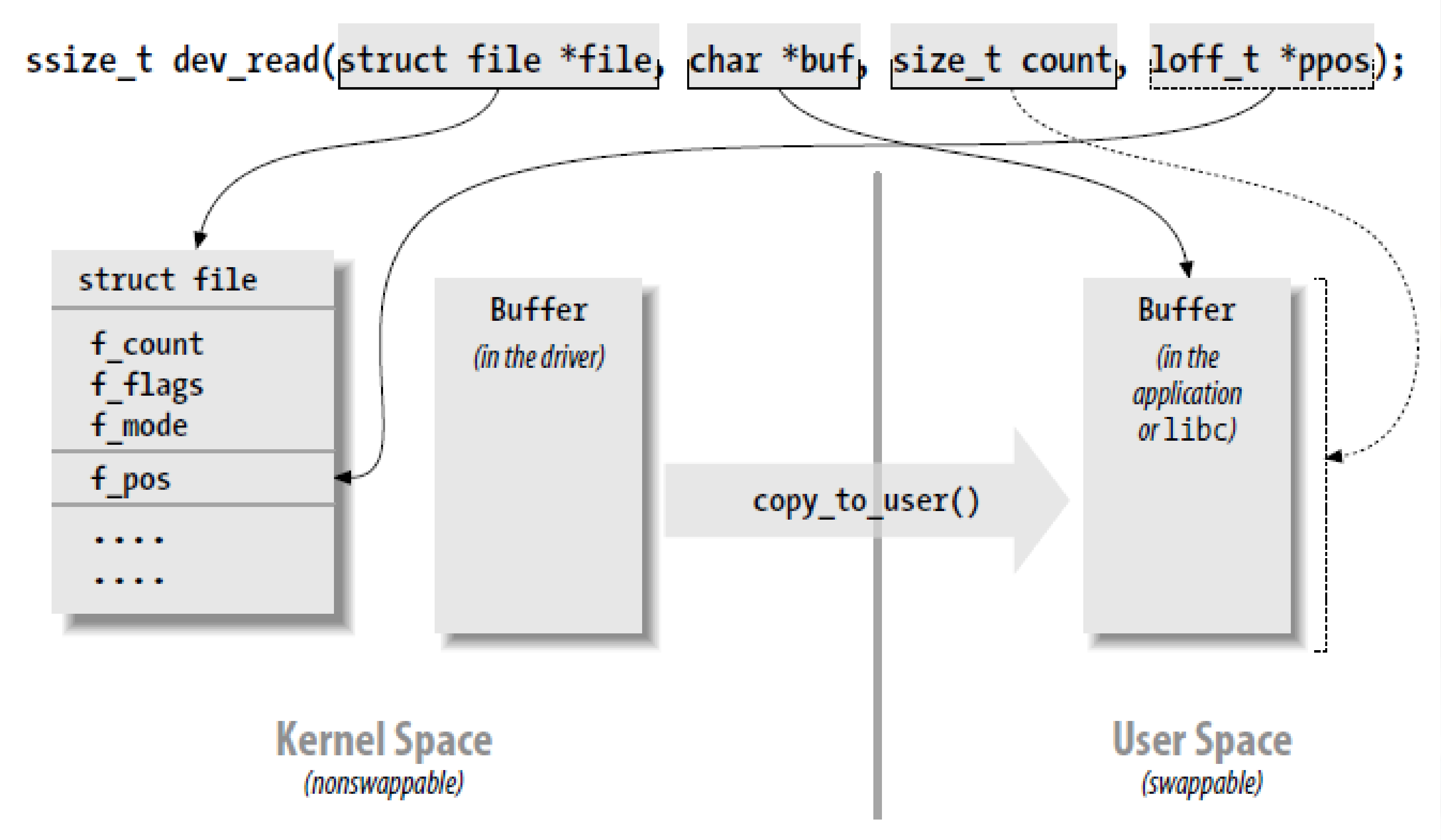
```
Character devices:
1 mem
2 pty
3 ttyp
4 ttyS
6 lp
7 vcs
10 misc
13 input
14 sound
```

Driver usage in userspace

- Making it accessible to userspace application by creating a device node: mknod /dev/demo c 202 128
- Using normal the normal file API :

```
fd = open("/dev/demo", O_RDWR);
ret = read(fd, buf, bufsize);
ret = write(fd, buf, bufsize);
```

The arguments to read



Concurrency and Race Conditions

- system tries to do more than one thing at once

- concurrency-related bugs are some of the easiest to create and some of the hardest to find

Linux Semaphore Implementation

- Semaphore for multi instance resources sharing
- Mutex for single exclusive resource sharing

```
/* with value */
void sema_init(struct semaphore *sem, int val);

/* concurency only */
DECLARE_MUTEX(name); /* mutex is sema to 1 */
DECLARE_MUTEX_LOCKED(name); /* already to 0 */

 /* dynamically */
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

Reader/Writer Semaphores

- It is often possible to allow multiple concurrent readers, as long as nobody is trying to make any changes

```
void init_rwsem(struct rw_semaphore *sem);

void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);

void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

Up and down

- In the Linux world, the *P/V* functions are called *down/up*

```
/* create unkillable processes */
void down(struct semaphore *sem);

/* allow the user-space process that is waiting on a
 * semaphore to be interrupted by the user */
int down_interruptible(struct semaphore *sem);

/* if the semaphore is not available at the time
 * of the call, down_trylock returns immediately
 * with a nonzero return value */
int down_trylock(struct semaphore *sem);

void up(struct semaphore *sem);
```

Completion

- Any process trying to read from the device will wait until some other process writes to the device.

```
DECLARE_COMPLETION(comp);
ssize_t complete_read (struct file *filp, char __user *buf, ...)
{
        printk(KERN "process %i going to sleep\n", current->pid);
        wait_for_completion(&comp);
        return 0; /* EOF */
}
ssize_t complete_write (struct file *filp, const char __user
*buf, ...)
{
        printk(KERN "process %i awakening...\n", current->pid);
        complete(&comp);
        return count; /* succeed, to avoid retrial */
}
```

Spinlocks

- A spinlock is a mutual exclusion device that can have only two values: “locked” and “unlocked.”
- If the lock has been taken by somebody else, the code goes into a tight loop where it repeatedly checks the lock until it becomes available. This loop is the “spin” part of a spinlock.
- Intended for use on multiprocessor systems

```
void spin_lock(spinlock_t *lock);

/* the previous interrupt state is stored in flags *:
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);

void spin_lock_irq(spinlock_t *lock);

/* disables software interrupts before taking the lock, but leaves
hardware interrupts enabled. */
void spin_lock_bh(spinlock_t *lock)
```

Atomic Variables

- Sometimes, a shared resource is a simple integer value
- Even a simple operation such as `N_op++;` requires locking
- An `atomic_t` holds an `int` value on all supported architectures. Because of the way this type works on some processors, however, the full integer range may not be available; thus, you should not count on an `atomic_t` holding more than 24 bits.

```
atomic_t v = ATOMIC_INIT(0);
void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add/sub(int i, atomic_t *v);
void atomic_inc/dec(atomic_t *v);
int atomic_inc/dec/sub_and_test(atomic_t *v); /* check is null */
int atomic_sub_and_test(int in, atomic_t *v); /* check is null */
int atomic_add_negative(int i, atomic_t *v);
int atomic_add/sub_return(int i, atomic_t *v);
int atomic_inc/dec_return(atomic_t *v);
```

Bit Operations

- Manipulating individual bits in an atomic manner

```
void set/clear/change_bit(nr, void *addr);
test_bit(nr, void *addr);
int test_and_set/clear/change_bit(nr, void *addr);
```

- CPU optimized with assembly implementation

SeqLock

- Allow readers free access to the resource but require to check for collisions with writers and, when such a collision happens, retry.

```
seqlock_t lock1 = SEQLOCK_UNLOCKED;
unsigned int seq;
do {
        seq = read_seqbegin(&the_lock);
        /* Do what you need to do */
} while read_seqretry(&the_lock, seq);
```

```
/* in writer side */
void write_seqlock(seqlock_t *lock);
void write_sequnlock(seqlock_t *lock);
```

- Mind the `irqsave/irqrestore/isr/bh` extention for both read and write

Example

- Wrap-up example of:
  - Linux Kernel Module
  - Char device
  - Race condition

ch5.zip

Advanced Char Driver Operations

- Device-specific operations such as changing the speed of a serial port, setting the volume on a soundcard, configuring video-related parameters on a framebuffer are not handled by the file operations

IOCTL

- Most drivers need—in addition to the ability to read and write the device—the ability to perform various types of hardware control via the device driver.

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

- The *inode* and *filp* pointers are the values corresponding to the file descriptor *fd* passed on by the application and are the same parameters passed to the *open* method.
- The *cmd* argument is passed from the user unchanged, and the optional *arg* argument is passed in the form of an unsigned long.

IOCTL command

- Command contains a type, a number, a read or write flag.
- Macros helps in manipulating the commands

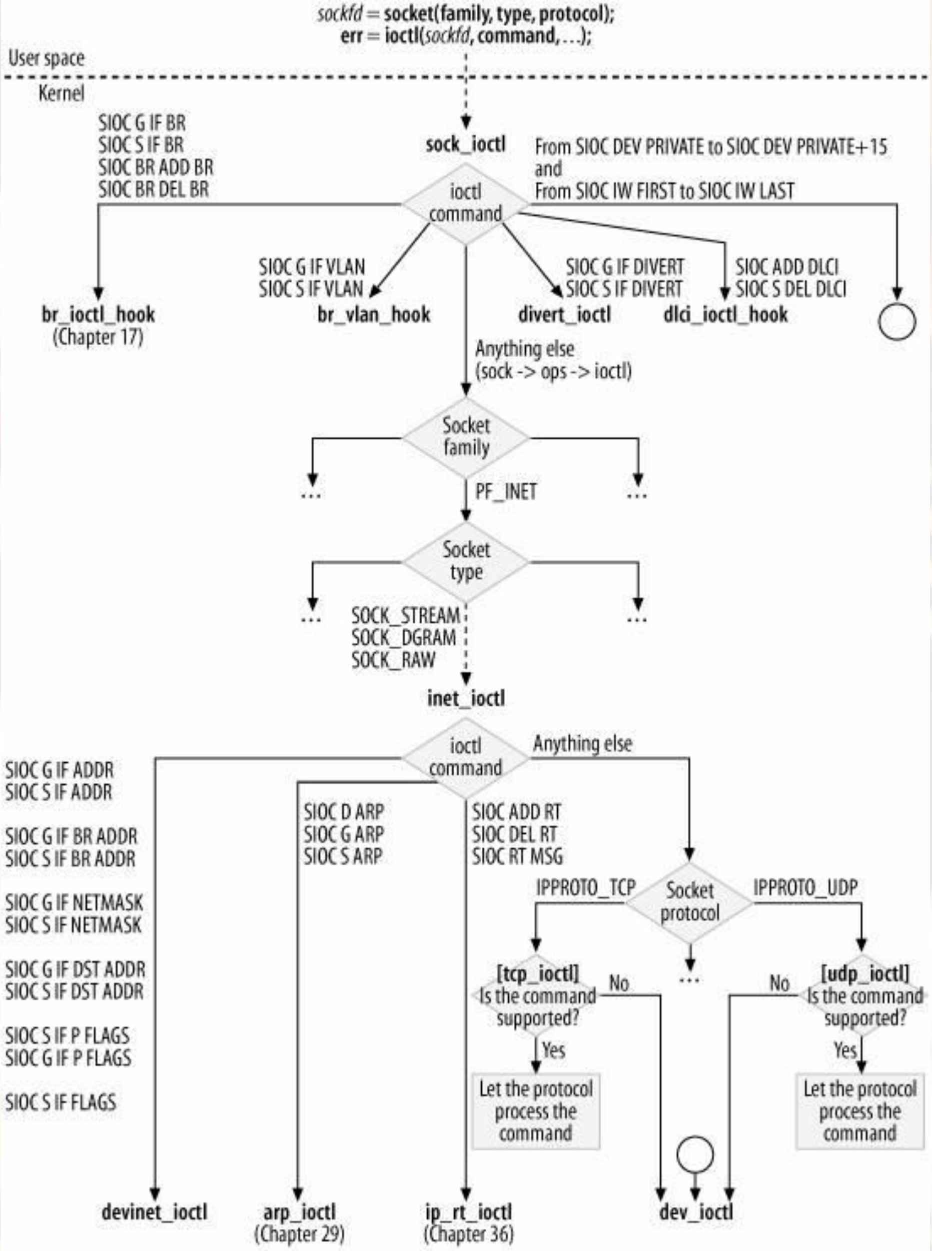
```
if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;
if (_IOC_DIR(cmd) & _IOC_READ)
       err = !access_ok(VERIFY_WRITE, (void __user *)arg,
                       _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
       err = !access_ok(VERIFY_READ, (void __user *)arg,
                        _IOC_SIZE(cmd));
if (err) return -EFAULT;
```

IOCTL usage

- Use the ioctl() system call.

```
int fd, val;

fd = open("/dev/demo", O_RDWR);

ioctl(fd, DEMO_CMD1, & val);
```

IOCTL example



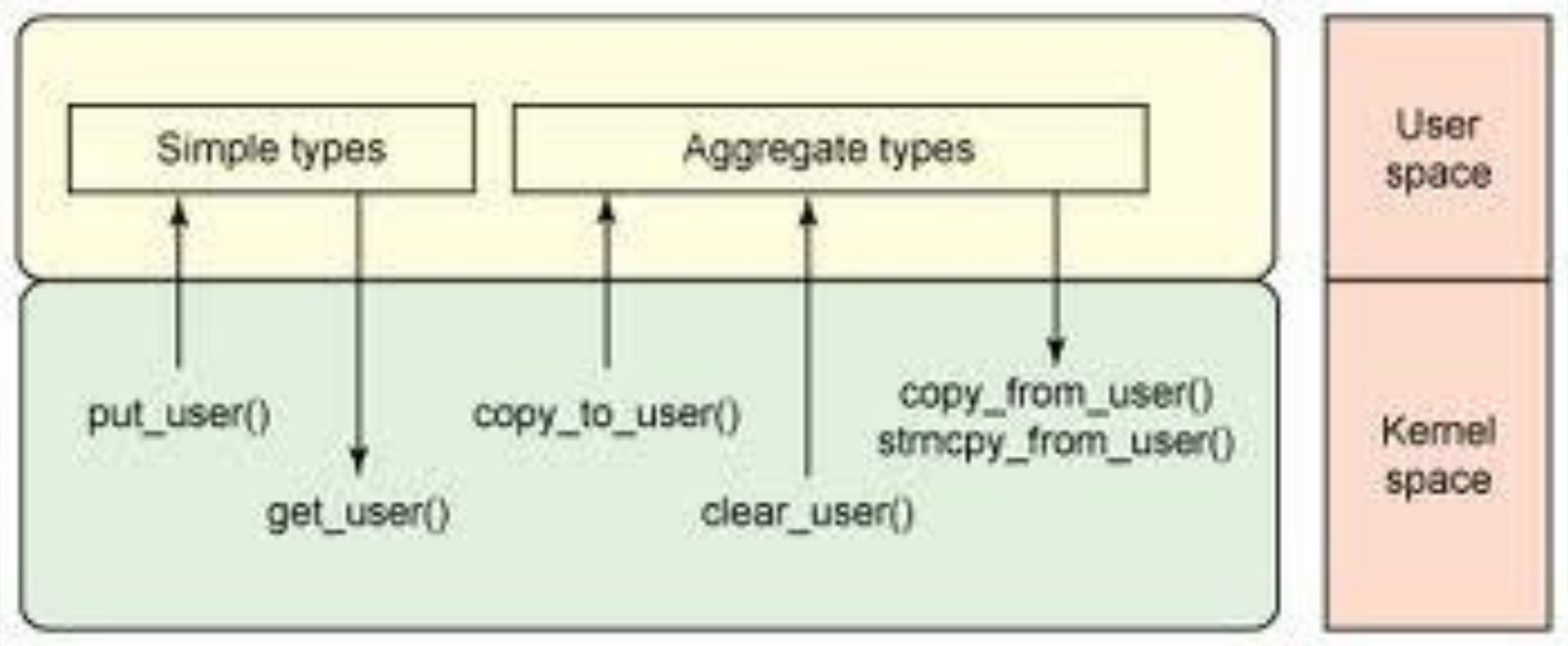
```
sockfd = socket(family, type, protocol);
err = ioctl(sockfd, command,...);
```

IOCTL passing parameters

- The driver and the user space of the calling process does not share the sale memory space
- Functions are needed to exchange data

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

int internal;
put_user (internal, (char __user*)arg);
get_user (internal, (char __user*)arg);
```

IOCTL parameters



Example with write

```
static ssize_t mydriver3_write(struct file *file,
const char *buf, size_t count, loff_t *ppos)
{
        size_t real;
        real = min((size_t)BUF_SIZE, count);
        if (real)
                if (copy_from_user(buffer, buf, real))
                        return -EFAULT;
                num = real;
        printk(KERN_DEBUG "mydriver3: wrote %d/%d chars
        %s\n", real, count, buffer);
        return real;
}
```

Example with read

```
static ssize_t mydriver3_read(struct file *file, char
*buf, size_t count, loff_t *ppos)
{
        size_t real;
        real = min(num, count);
        if (real)
                if (copy_to_user(buf, buffer, real))
                        return -EFAULT;
                num = 0;
        printk(KERN_DEBUG "mydriver3: read %d/%d chars
        %s\n", real, count, buffer);
        return real;
}
```

Test of the read/write example

```
# insmod mydriver3.ko
# echo -n salut > /dev/mydriver3
mydriver3: wrote 5/5 chars salut
$ cat /dev/mydriver3
salut
```

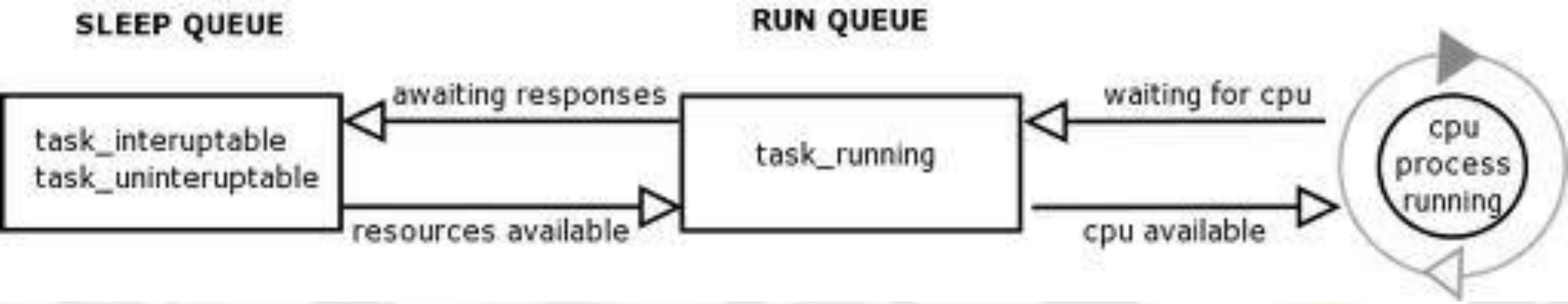
The Implementation of the ioctl Commands

```
int quantum;
/* Set by pointer */
ioctl(fd,SCULL_IOCSQUANTUM, &quantum);
/* Set by value */
ioctl(fd,SCULL_IOCTQUANTUM, quantum);
/* Get by pointer */
ioctl(fd,SCULL_IOCGQUANTUM, &quantum);
/* Get by return value */
quantum = ioctl(fd,SCULL_IOCQQUANTUM);
/* Exchange by pointer */
ioctl(fd,SCULL_IOCXQUANTUM, &quantum);
/* Exchange by value */
quantum = ioctl(fd,SCULL_IOCHQUANTUM, quantum);
```

Advanced Char Driver Operations

Sleeping

Task status and queue



Wait queue

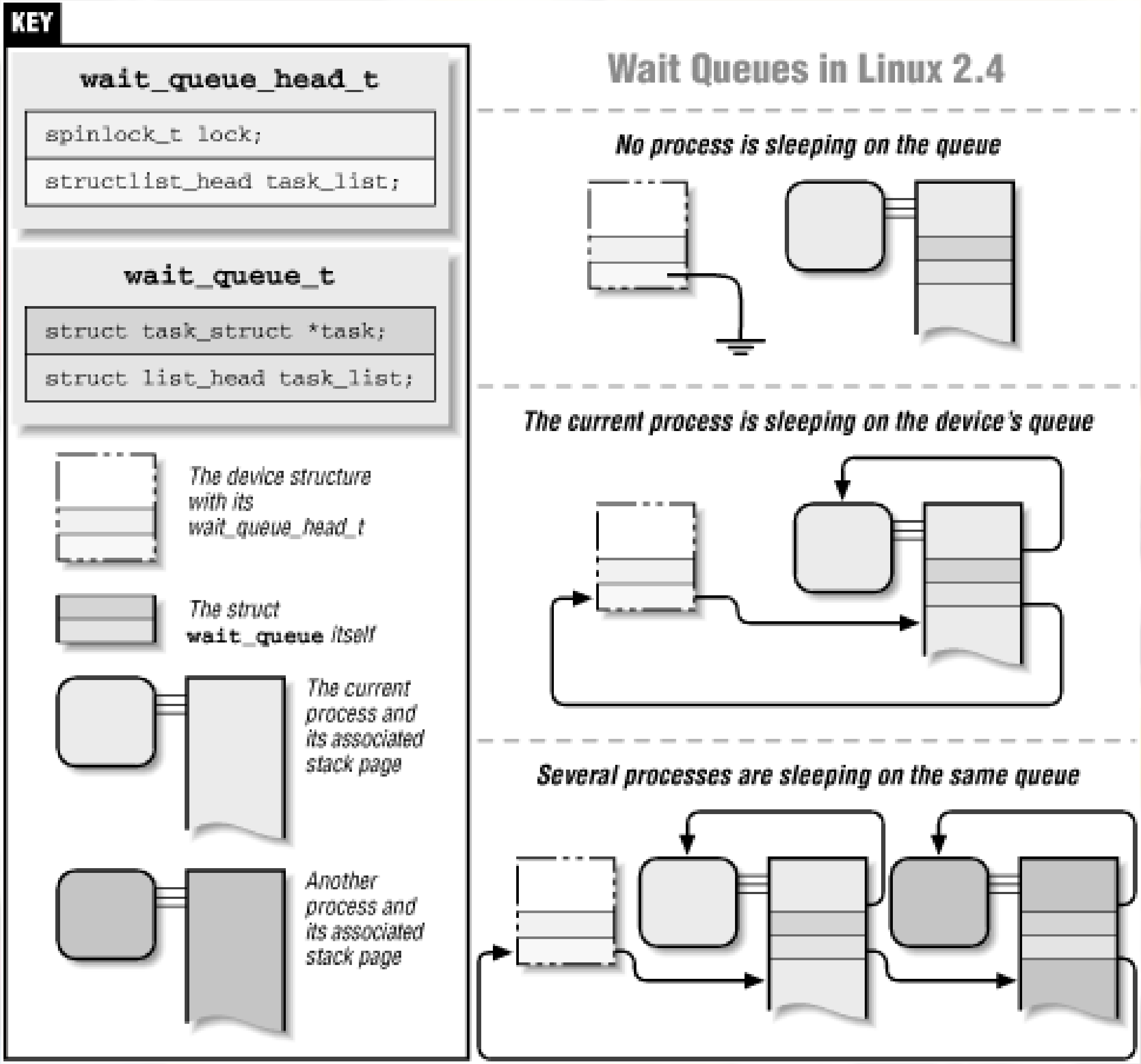
- Never sleep when you are running in an atomic context, if you have disabled interrupts.
- Check that holding a semaphore does not block the process that will eventually wake you up.
- After wake up you can make no assumptions about the state of the system after you wake up, and you must check to ensure that the condition you were waiting for is, indeed, true.
- A wait queue is like a list of processes, all waiting for a specific event.

Statical declaration:

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

Dynamic declaration:

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```


KEY
wait_queue_head_t
spinlock_t lock;
structlist_head task_list;
wait_queue_t
struct task_struct *task;
struct list_head task_list;
The device structure with its wait_queue_head_t
The struct wait_queue itself
The current process and its associated stack page
Another process and its associated stack page
Wait Queues in Linux 2.4
No process is sleeping on the queue
The current process is sleeping on the device's queue
Several processes are sleeping on the same queue

Simple sleeping

- Sleep

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

- Wake up

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

- Rational example
  - If a process calls *read* but no data is (yet) available or if a process calls *write* and there is no space in the buffer, the process must block.

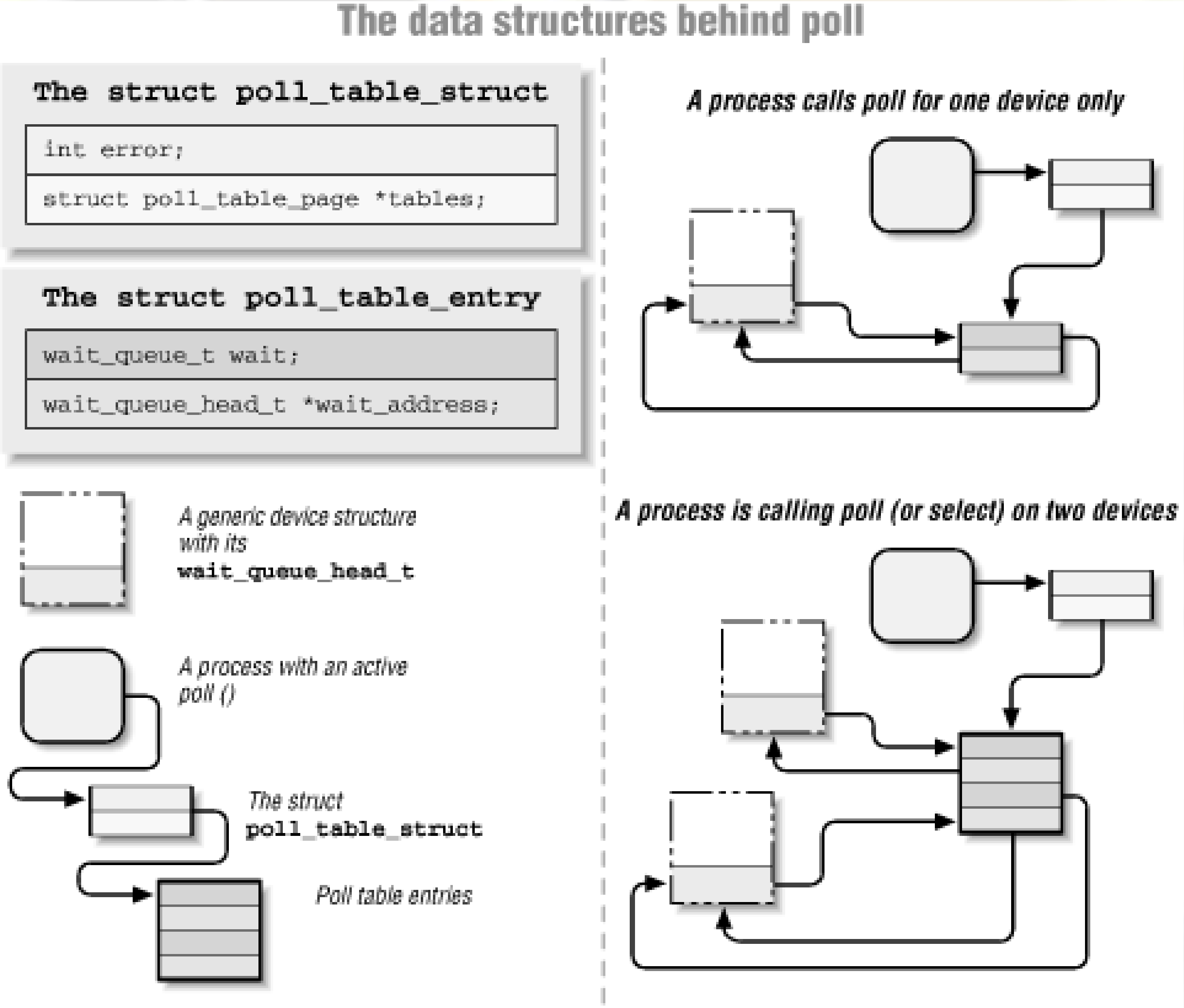
- Extra : `wake_up_nr, wake_up_all, wake_up_sync (+interruptible)`

Sleeping example

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;
ssize_t sleepy_read (struct file *filp, char __user *buf,
                     size_t count, loff_t *pos)
{
       wait_event_interruptible(wq, flag != 0);
       flag = 0;
       return 0; /* EOF */
}
ssize_t sleepy_write (struct file *filp, const char __user *buf,
                      size_t count,loff_t *pos)
{
       flag = 1;
       wake_up_interruptible(&wq);
       return count; /* succeed, to avoid retrial */
}
```

Poll and select

- Applications that use nonblocking I/O often use the *poll, select,* and *epoll* system calls. Each allow a process to determine whether it can read from or write to one or more open files without blocking.
- If no file descriptors are currently available for I/O, the kernel causes the process to wait on the wait queues for all file descriptors passed to the system call.


The data structures behind poll
The struct poll_table_struct
int error;
struct poll_table_page *tables;
The struct poll_table_entry
wait_queue_t wait;
wait_queue_head_t *wait_address;
A generic device structure with its wait_queue_head_t
A process with an active poll ()
The struct poll_table_struct
Poll table entries
A process calls poll for one device only
A process is calling poll (or select) on two devices

Poll example

- Check of any read or write

```
static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
         struct scull_pipe *dev = filp->private_data;
         unsigned int mask = 0;
         down(&dev->sem);
         poll_wait(filp, &dev->inq, wait);
         poll_wait(filp, &dev->outq, wait);
         if (dev->rp != dev->wp) mask |= POLLIN | POLLRDNORM; /* readable */
         if (spacefree(dev))     mask |= POLLOUT | POLLWRNORM; /* writable */
         up(&dev->sem);
         return mask;
}
```

- Something to read

```
dev->wp += new_data_size; wake_up(& dev->inq);
```

- Ready for write

```
Addspace(dev); wake_up(& dev->outq);
```

Asynchronous notification

- Let’s imagine a process that executes a long computational loop at low priority but needs to process incoming data as soon as possible. This application could be written to call *poll* regularly to check for data, but, for many situations, there is a better way. By enabling asynchronous notification, this application can receive a signal whenever data becomes available and need not concern itself with polling.

```
fcntl(STDIN_FILENO, F_SETOWN, getpid( ));
oflags = fcntl(STDIN_FILENO, F_GETFL);
fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
```

```
static int scull_p_fasync(int fd, struct file *filp, int mode)
{
         struct scull_pipe *dev = filp->private_data;
         return fasync_helper(fd, filp, mode, &dev->async_queue);
}
if (dev->async_queue) kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
```

Time, Delays, and Deferred Work

- Measuring time lapses and comparing times
- Knowing the current time
- Delaying operation for a specified amount of time
- Scheduling asynchronous functions

Comparing time

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;
j = jiffies; /* read the current value */
stamp_1 = j + HZ; /* 1 second in the future */
stamp_half = j + HZ/2; /* half a second */
stamp_n = j + n * HZ / 1000; /* n milliseconds */
u64 get_jiffies_64(void);
```

- On 32-bit platforms the counter wraps around only once every 50 days, your code should be prepared to face that event.

```
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

Getting time

- Time of the day

```
void do_gettimeofday(struct timeval *tv);
```

- Format conversion

```
unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies,
                         struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies,
                        struct timeval *value);

unsigned long mktime (unsigned int year, unsigned int mon,
                      unsigned int day, unsigned int hour,
                      unsigned int min, unsigned int sec);
```

Short Delays

- All not always implemented depending on platform
- Busy waiting

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

- Putting the calling process in sleep for a given number of milliseconds

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds)
```

Long delay

- Informing the processor is hardly unused : not blocking but the hugly

```
while (time_before(jiffies, j1)) cpu_relax( );
```

- Requesting the kernel to reallocate CPU, but still polling

```
while (time_before(jiffies, j1)) schedule( );
```

- Cheating with the event queuing

```
wait_queue_head_t wait;
init_waitqueue_head (&wait);
wait_event_interruptible_timeout(wait, 0, delay);
```

- The process will no more be running

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (delay);
```

Kernel timer

- Whenever you need to schedule an action to happen later, without blocking the current process until that time arrives, kernel timers are the tool for you.

```
#include <linux/timer.h>
struct timer_list {
         /* ... */
         unsigned long expires;
         void (*function)(unsigned long);
         unsigned long data;
};
void init_timer(struct timer_list *timer); /* dynamic */
struct timer_list TIMER_INITIALIZER(_function, _expires, _data); /* static */

void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
int mod_timer(struct timer_list *timer, unsigned long expires);
```

Tasklet

- If the hardware interrupt must be managed as quickly as possible, most of the data management can be safely delayed to a later time.
- The kernel executed the tasklet asynchronously and quickly, for a short period of time, in the context of a “soft interrupt” in atomic mode.

```
struct tasklet_struct {
      /* ... */
      void (*func)(unsigned long);
      unsigned long data;
};
void tasklet_init(struct tasklet_struct *t,
void (*func)(unsigned long), unsigned long data);
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

WorkQueue

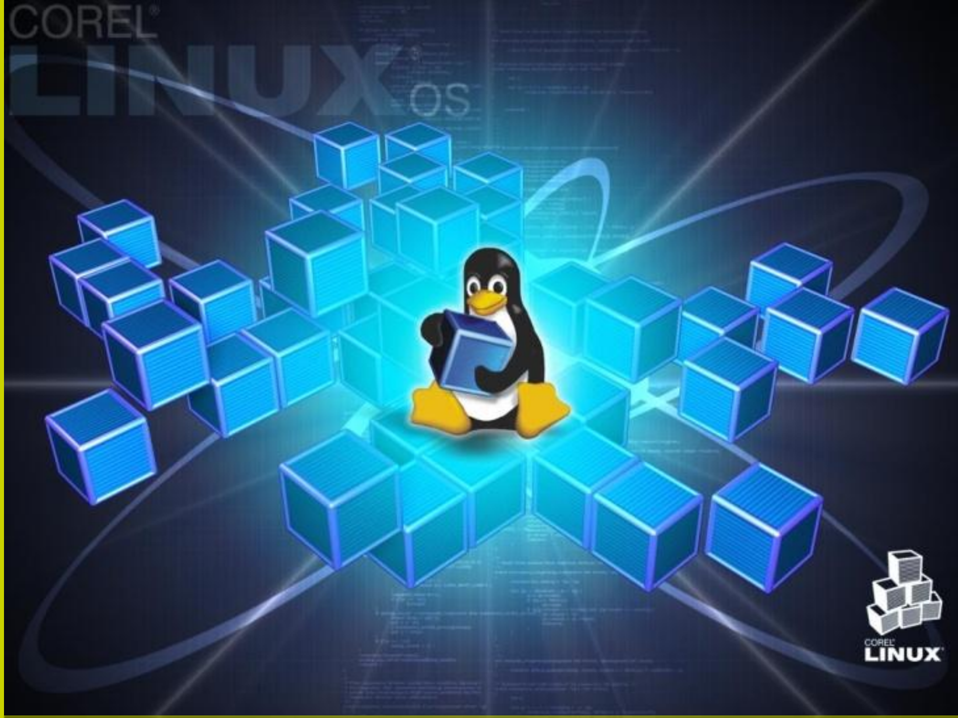
- Workqueue functions may have higher latency but need not be atomic.
- Run in the context of a special kernel process with more flexibility.
  Functions can sleep.

```
struct workqueue_struct *create_workqueue(const char *name);
int queue_work(struct workqueue_struct *queue,
               struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue,
                       struct work_struct *work,
                       unsigned long delay);
int cancel_delayed_work(struct work_struct *work);
void flush_workqueue(struct workqueue_struct *queue);
void destroy_workqueue(struct workqueue_struct *queue);
```

The Shared Queue

- If you only submit tasks to the queue occasionally, it may be more efficient to simply use the shared, default workqueue that is provided by the kernel.

```
DECLARE_WORK(name, void (*function)(void *), void *data);
int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work,
                          unsigned long delay);
```



Allocating Memory

- Memory in device drivers, controlled by MMU
- How to optimize memory resources
- Kernel offers a unified memory management interface to the drivers, then knowledge of internal details of memory management is useless (segmentation, paging…)

Kmalloc / kfree

- Do not clear memory it obtains
- The allocated region is also contiguous in **physical memory**
- The virtual address range used by *kmalloc* and *__get_free_pages* features a one-to-one mapping to physical memory, possibly shifted by a constant PAGE_OFFSET value.
- Available only in page-sized chunks (2nKB)

```
void *kmalloc(size_t size, int flags);
void kfree();
```

- Most common flags:
  - GFP_KERNEL in process context for kernel memory allocation
    - GFP_NOFS and GFP_NOIO for more restrictions
  - GFP_ATOMIC in interrupt, tasklets and timer context that cannot sleep
  - GFP_USER for user space allocation

Extra memory zones

- DMA-capable memory is memory that lives in a preferential address range, where peripherals can perform DMA access.
- High memory is a mechanism used to allow access to (relatively) large amounts of memory on 32-bit platforms.
- Flags:
  - __GFP_DMA and __GFP_HIGHMEM

Big chunk of memory

- Needs to allocate big chunks of memory

```
unsigned long __get_free_pages(unsigned int flags,
                               unsigned int order);
void free_pages(unsigned long addr, unsigned long order);
```

- Order is the base-two logarithm of the number of pages, (i.e., log2N). For example, 0 → 1 page, 3 → 8 pages
- Still virtual memory address handled by the MMU but with direct mapping with physical memory

Caches

- Allocating many objects of the same size, over and over
- Kernel facilities: special pools for high-volume objects: *lookaside cache*
- Mainly used for USB and SCSI

```
/* create a cache for quanta */
scullc_cache = kmem_cache_create("scullc", scullc_quantum, 0,
                               SLAB_HWCACHE_ALIGN, NULL, NULL);

/* Allocate a quantum using the memory cache */
dptr->data[i] = kmem_cache_alloc(scullc_cache, GFP_KERNEL);

/* And these lines release memory: */
kmem_cache_free(scullc_cache, dptr->data[i]);
```

Memory pools

- There are places in the kernel where memory allocations cannot be allowed to fail. As a way of guaranteeing allocations in those situations, the kernel developers created an abstraction known as a *memory pool (or “mempool”).* A memory pool is really just a form of a lookaside cache that tries to always keep a list of free memory around for use in emergencies.

```
/* setup */
cache = kmem_cache_create(. . .);
pool = mempool_create(MY_POOL_MINIMUM, mempool_alloc_slab,
                      mempool_free_slab,cache);
/* objects allocation and free */
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
/* releasing */
void mempool_destroy(mempool_t *pool);
```

Vmalloc

- Allocates a contiguous memory region in the *virtual address space.*
- Pages are not consecutive in physical memory ➔ less efficient
- One of the fundamental Linux memory allocation mechanisms

```
void *vmalloc(unsigned long size);
void vfree(void * addr);
void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

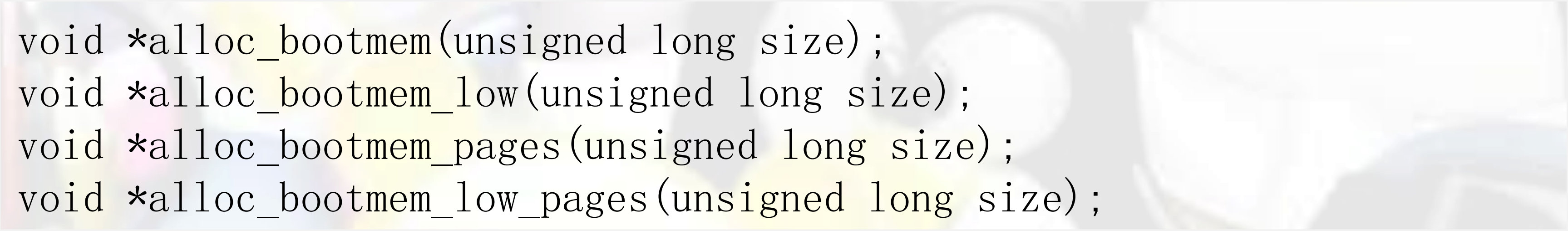
- The address range used by *vmalloc* and *ioremap* is completely synthetic, and each allocation builds the (virtual) memory area by suitably setting up the page tables.
- Cannot be used in atomic context: it uses *kmalloc(GFP_KERNEL)*
- In the range from VMALLOC_START to VMALLOC_END.

Vmalloc & IO-remap

- To be used for the microprocessor, on top of the processor’s MMU.
  - Not suitable for a driver that needs a real physical address (such as a DMA address, used by peripheral hardware to drive the system’s bus)
  - The right time to call *vmalloc* is when you are allocating memory for a large sequential buffer that exists only in software.
  - *vmalloc* has more overhead than *__get_free_pages*
    - retrieve the memory and build the page tables
  - It doesn’t make sense to call *vmalloc* to allocate just one page.
- *ioremap* is most useful for mapping the (physical) address of a PCI buffer to (virtual) kernel space. For example, it can be used to access the frame buffer of a PCI video device; such buffers are usually mapped at high physical addresses, outside of the address range for which the kernel builds page tables at boot time.

Acquiring a Huge Buffers at Boot Time

- Allocation at boot time is the only way to retrieve consecutive memory pages while bypassing the limits imposed by *__get_free_pages*
- It bypasses all memory management policies by reserving a private memory pool. This technique is inelegant and inflexible, but it is also the least prone to failure.
- A module can’t allocate memory at boot time; only drivers directly linked to the kernel can do that !
  - A device driver using this kind of allocation can be installed or replaced only by rebuilding the kernel and rebooting the computer.
  - private use reduces the amount of RAM left for normal system operation.



```
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
```

Communicating with Hardware

- The driver is the abstraction layer between software concepts and hardware circuitry
- A driver can access I/O ports and I/O memory while being portable across Linux platforms.

I/O Registers and Conventional Memory

- A programmer accessing I/O registers must be careful to avoid being tricked by CPU (or compiler) optimizations that can modify the expected I/O behavior.
- A driver must ensure that no caching is performed and no read or write reordering takes place when accessing registers.
  - Example : A *rmb* (read memory barrier) guarantees that any reads appearing before the barrier are completed prior to the execution of any subsequent read.

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb( );
writel(dev->registers.control, DEV_GO);
```

I/O Port

- Exclusive access to the ports: the kernel provides a registration interface that allows your driver to claim the ports it needs

```
struct resource *request_region(unsigned long first,
                                unsigned long n, const char *name);
void release_region(unsigned long start, unsigned long n);
int check_region(unsigned long first, unsigned long n);
```

- Ports can be access in 8/16/32 bits, and also per string

```
unsigned inb/w/l(unsigned port);
void outb/w/l(unsigned char/short/long byte, unsigned port);
```

- Much of the source code related to port I/O is platform-dependent

I/O Memory

- The main mechanism used to communicate with devices is through memory-mapped registers and device memory.
- I/O memory is simply a region of RAM-like locations that the device makes available to the processor over the bus
  - Example : video data, Ethernet packets, device registers

```
struct resource *request_mem_region(unsigned long start,
                                    unsigned long len, char *name);
void release_mem_region(unsigned long start, unsigned long len);
int check_mem_region(unsigned long start, unsigned long len);
```

- You must also ensure that this I/O memory has been made accessible to the kernel.

```
void *ioremap(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
```

Accessing I/O Memory

```
unsigned int ioread8/16/32(void *addr);
void iowrite8/16/32(u8 value, void *addr);
```

- addr should be an address obtained from *ioremap* (perhaps with an integer offset); the return value is what was read from the given I/O memory.
- If you need to operate on a block of I/O memory

```
void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

Interrupt Handling

- It is always undesirable to have the processor wait on external events
- An *interrupt* is simply a signal that the hardware can send when it wants the processor's attention

DEVICES
Memory
CPU
KERNEL
Disk
APPLICATIONS
MAIL

Installing an Interrupt Handler

- A driver need only register a handler for its device’s interrupts, and handle them properly when they arrive.
- The kernel keeps a registry of interrupt lines. A module is expected to request irq channel before using it, and to release it when it’s done.

```
int request_irq(unsigned int irq,
                void (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                const char *dev_name, void *dev_id)
void free_irq(unsigned int irq, void *dev_id);
```

- Flags: SA_INTERRUPT, SA_SHIRQ
- dev_name: The string passed to request_irq is used in /proc/interrupts to show the owner of the interrupt
- void *dev_id: this pointer is used for shared interrupt lines.

Implementing a handler

- Give feedback to device about interrupt reception
- Transfer data according to the meaning of the interrupt being serviced.
- Awake processes sleeping on the device.

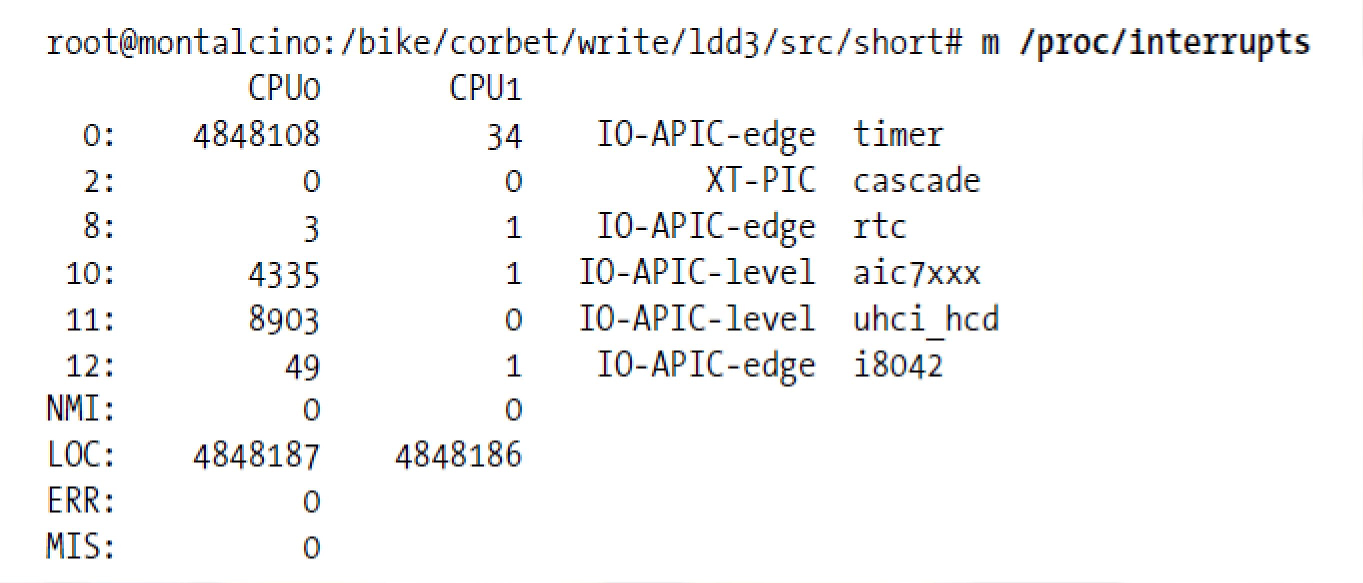
```
void irq_handle (int irq, void* dev, struct pt_regs* regs)
{
    wake_up_interruptible (&q);
}

//--------------------------------------------------------------
static int device_open (struct inode *inode, struct file *file)
{
    irq = request_irq (7, irq_handle, SA_INTERRUPT, “my_irq", NULL);
    return 0;
}
```

Interrupt sharing

- Shared interrupts are installed through request_irq just like nonshared ones
- there are two differences:
  - The SA_SHIRQ bit must be specified
  - The dev_id argument must be unique.

The /proc Interface

- Whenever a hardware interrupt reaches the processor, an internal counter is incremented, providing a way to check whether the device is working as expected.
- Reported interrupts are shown in */proc/interrupts.*



```
root@montalcino:/bike/corbet/write/ldd3/src/short# m /proc/interrupts
           CPU0       CPU1
  0:    4848108         34    IO-APIC-edge  timer
  2:          0          0          XT-PIC  cascade
  8:          3          1    IO-APIC-edge  rtc
 10:       4335          1   IO-APIC-level  aic7xxx
 11:       8903          0   IO-APIC-level  uhci_hcd
 12:         49          1    IO-APIC-edge  i8042
NMI:          0          0
LOC:    4848187    4848186
ERR:          0
MIS:          0
```

Auto-detecting the IRQ Number

- How to determine which IRQ line is going to be used by the device.
- The driver needs the information in order to correctly install the handler

```
insmod ./short.ko irq=x
```

- Auto-detection of the interrupt number is a basic requirement
- The technique is quite simple: the driver tells the device to generate interrupts and watches what happens. If everything goes well, only one interrupt line is activated

Kernel-assisted probing

- The Linux kernel offers a low-level facility for probing the interrupt number. It works for only non-shared interrupts

```
unsigned long mask;
mask = probe_irq_on( );
outb_p(0x10,short_base+2); /* enable reporting */
outb_p(0x00,short_base); /* clear the bit */
outb_p(0xFF,short_base); /* set the bit: interrupt! */
outb_p(0x00,short_base+2); /* disable reporting */
udelay(5); /* give it some time */
short_irq = probe_irq_off(mask);
if (short_irq = = 0) { /* none of them? */
    printk(KERN_INFO "short: no irq reported by probe\n");
}
```

- Hardware that is capable of working in a shared interrupt mode provides better ways of finding the configured interrupt number.

Top and Bottom Halves

- One of the main problems with interrupt handling is how to perform lengthy tasks within a handler.
- Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long.
- These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind.
- The so-called *top half* is the routine that actually responds to the interrupt—the one you register with *request_irq.*
- The *bottom half* is a routine that is scheduled by the top half to be executed later, at a safer time.

Tasklet: top half

```
void short_do_tasklet(unsigned long);
DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);

irqreturn_t short_tl_interrupt(int irq, void *dev_id,
                               struct pt_regs *regs)
{
    short_incr_tv(&tv_head);
    tasklet_schedule(&short_tasklet);
    short_wq_count++; /* record that an interrupt arrived */
    return IRQ_HANDLED;
}
```

Tasklets : bottom half

```
void short_do_tasklet (unsigned long unused)
{
  /* The bottom half reads the tv array, filled by the top half,
   * and prints it to the circular text buffer, which is then
   * consumed by reading processes
   */
  /* First write the number of interrupts that occurred before
   * this bh
   */
  /*
   * Then, write the time values. Write exactly 16 bytes at a time,
   * so it aligns with PAGE_SIZE
   */
  wake_up_interruptible(&short_queue); /* awake any reading process */
}
```

Workqueues

- Since the *workqueue* function runs in process context, it can sleep if need be.

```
static struct work_struct short_wq;
INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
/-------------------------------------------------------------------/
irqreturn_t short_wq_interrupt(int irq, void *dev_id,
                               struct pt_regs *regs)
{
        /* Grab the current time information. */
        do_gettimeofday((struct timeval *) tv_head);
        short_incr_tv(&tv_head);
        /* Queue the bh. Don't worry about multiple enqueueing */
        schedule_work(&short_wq);
        short_wq_count++; /* record that an interrupt arrived */
        return IRQ_HANDLED;
}
```

Homework

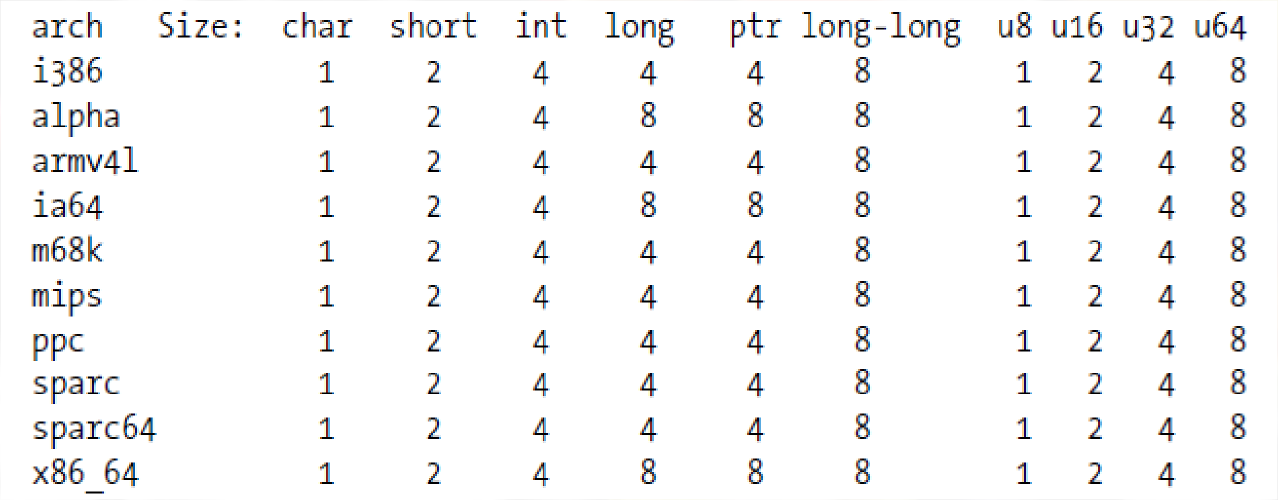
- Do a SART diagram of the ‘Write-Buffering Example’

Data Types in the Kernel

- three main classes highly portable.

Data types

- C types



arch	Size:	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386		1	2	4	4	4	8	1	2	4	8
alpha		1	2	4	8	8	8	1	2	4	8
armv4l		1	2	4	4	4	8	1	2	4	8
ia64		1	2	4	8	8	8	1	2	4	8
m68k		1	2	4	4	4	8	1	2	4	8
mips		1	2	4	4	4	8	1	2	4	8
ppc		1	2	4	4	4	8	1	2	4	8
sparc		1	2	4	4	4	8	1	2	4	8
sparc64		1	2	4	4	4	8	1	2	4	8
x86_64		1	2	4	8	8	8	1	2	4	8

- Assigning an Explicit Size to Data Items

```
u8; /* unsigned byte (8 bits) */
u16; /* unsigned word (16 bits) */
u32; /* unsigned 32-bit value */
u64; /* unsigned 64-bit value */
```

- Kernel type_t

```
pid_t;
Size_t
```

Others Portability Issues

- Time Intervals
  - 1000 jiffies per second
  - msec*HZ/1000 jiffies per millisecond
- Page Size
  - PAGE_SIZE: memory page size (often 4Kbytes)
  - PAGE_SHIFT: number of bits to shift an address to get it page numbers (12)

```
int order = get_order(16*1024);
buf = get_free_pages(GFP_KERNEL, order);
```

- Byte Order
  - __BIG_ENDIAN or __LITTLE_ENDIAN

Data Alignment

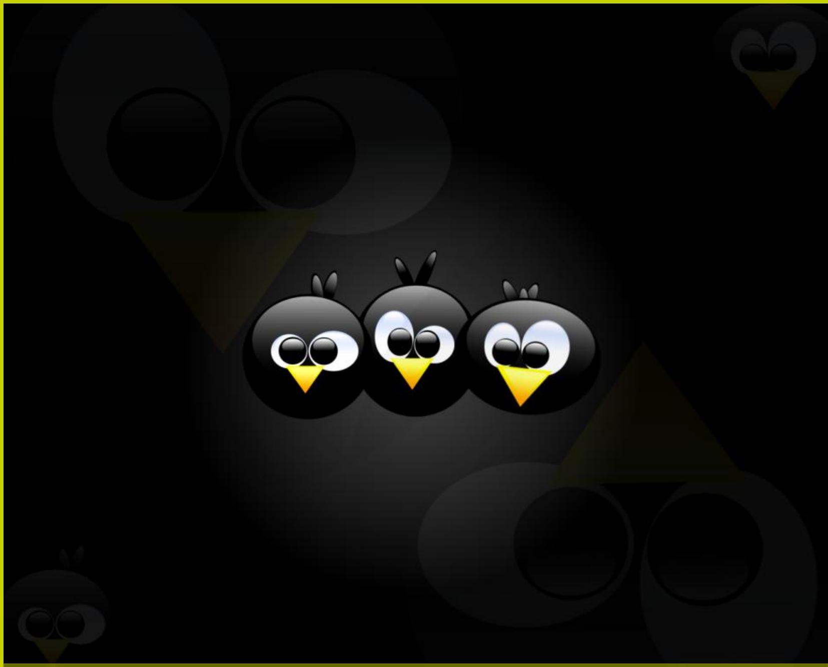
- How to read a 4-byte value stored at an address that isn't a multiple of 4 bytes
- Many modern architectures generate an exception every time the program tries unaligned data transfers
- Packed structure

```
struct {
        u16 id;
        u64 lun;
        u16 reserved1;
        u32 reserved2;
} __attribute__ ((packed)) scsi;
```

- Without the `__attribute__ ((packed))`, the `lun` field would be preceded by two filler bytes or six if we compile the structure on a 64-bit platform.

Pointers and Error Values

- A function returning a pointer type can return an error value with: `void *ERR_PTR(long error);` where error is the usual negative error code.
- The caller can use *IS_ERR* to test whether a returned pointer is an error code or not: `long IS_ERR(const void *ptr);`
- If you need the actual error code, it can be extracted with: `long PTR_ERR(const void *ptr);`
- You should use PTR_ERR only on a value for which IS_ERR returns a true value, any other value is a valid pointer.



Linked lists

- To reduce the amount of duplicated code, the kernel developers have created a standard implementation of circular, doubly linked lists
- It is your responsibility to implement a locking scheme

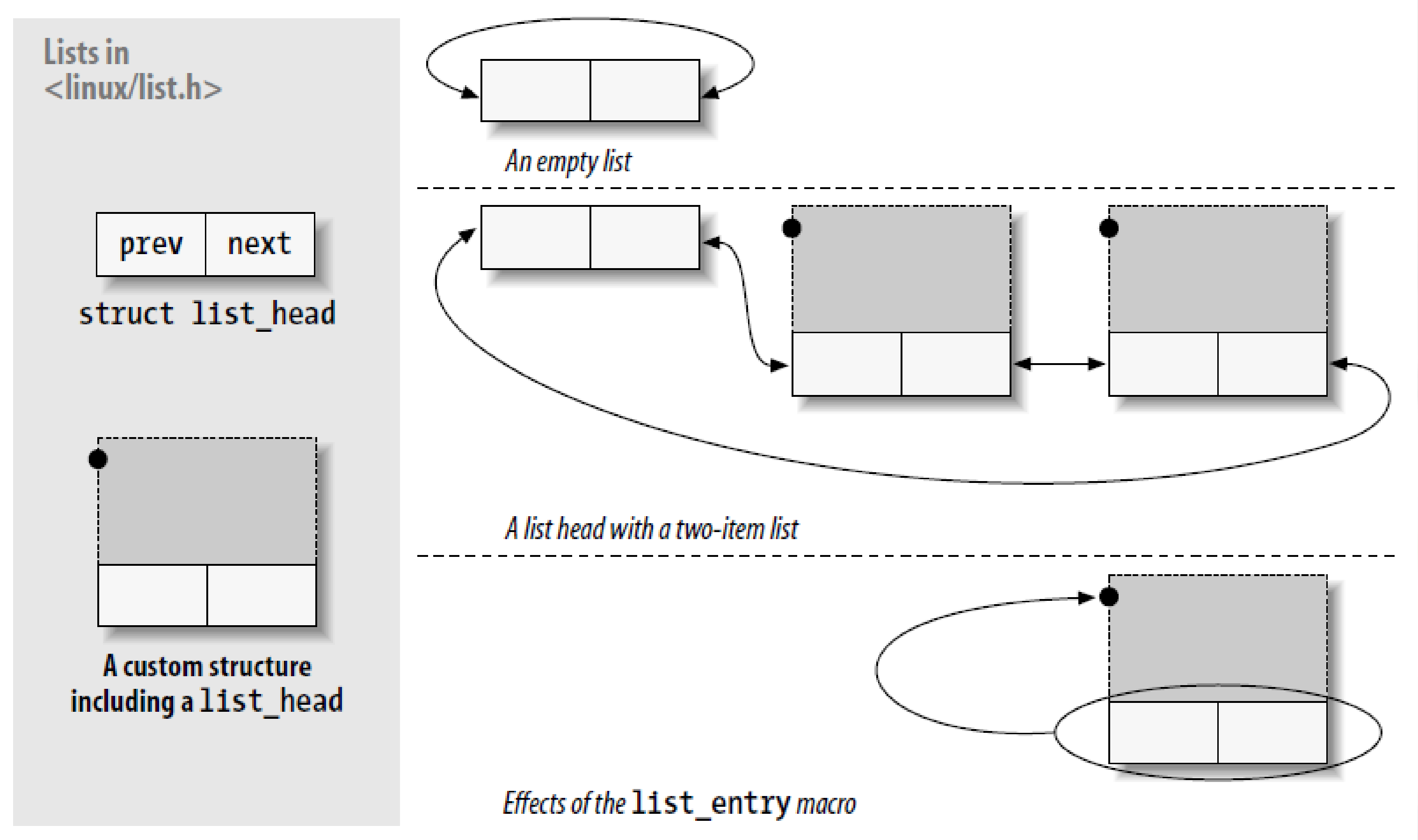
List head

```
struct list_head {
        struct list_head *next, *prev;
};
```

- To use the Linux list facility in your code, you need only embed a `list_head` inside the structures that make up the list

```
struct todo_struct {
        struct list_head list;
        int priority; /* driver specific */
        /* ... add other driver-specific fields */
};
```

The list head data structure



List making

```
list_add(struct list_head *new, struct list_head *head);
list_add_tail(struct list_head *new, struct list_head *head);
list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
list_empty(struct list_head *head); /* check the list is empty */
/* join */
list_splice(struct list_head *list, struct list_head *head);

/* maps a list_head structure pointer back into a pointer to the
structure that contains */
list_entry(struct list_head *ptr, type_of_struct, field_name);
```

Example

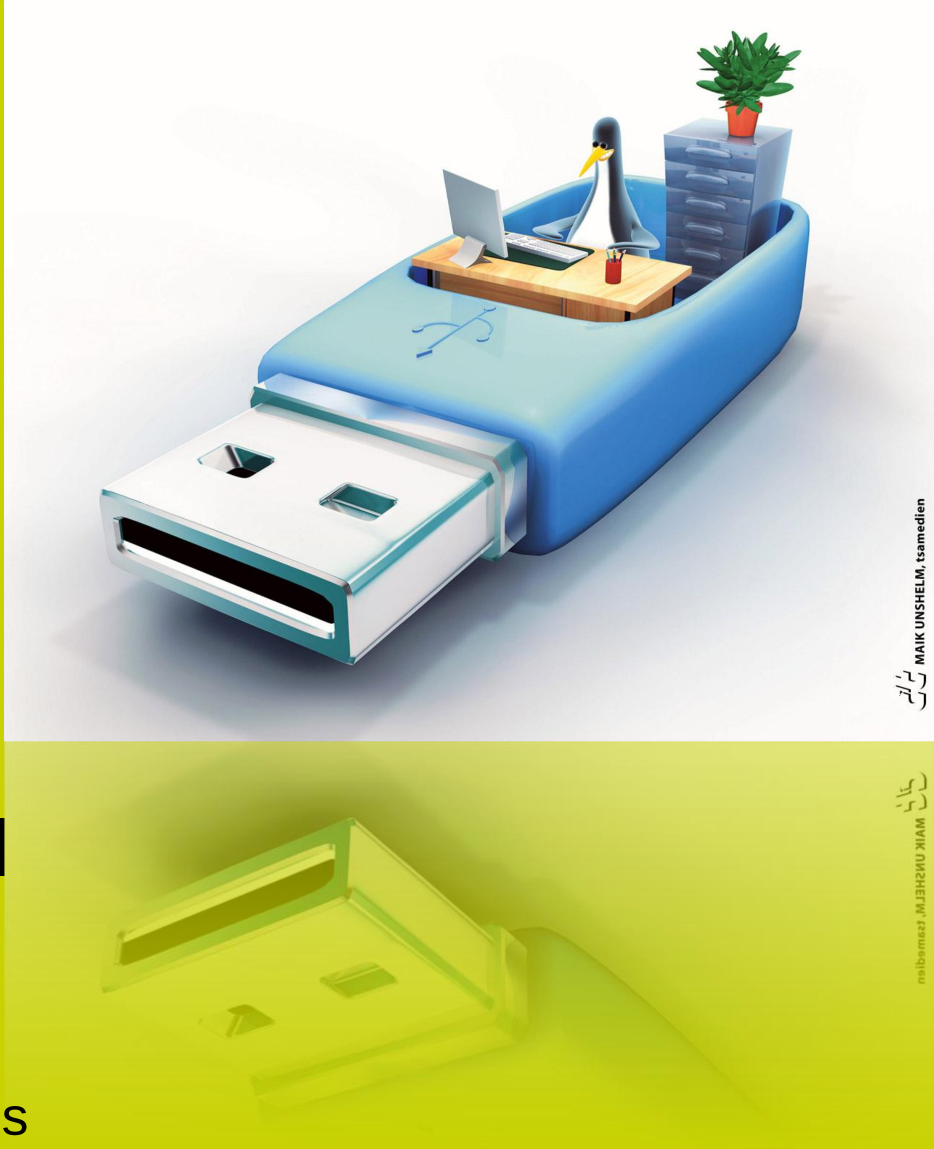
```
void todo_add_entry(struct todo_struct *new)
{
  struct list_head *ptr;
  struct todo_struct *entry;
  for (ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next){
    entry = list_entry(ptr, struct todo_struct, list);
    if (entry->priority < new->priority) {
      list_add_tail(&new->list, ptr);
      return;
    }
  }
  list_add_tail(&new->list, &todo_struct)
}
```

List browsing

```
list_for_each(struct list_head *cursor,
              struct list_head *list);
list_for_each_prev(struct list_head *cursor,
                   struct list_head *list);
list_for_each_safe(struct list_head *cursor,
                   struct list_head *next,
                   struct list_head *list);

/* no need to use list_entry with this */
list_for_each_entry(type *cursor,
                    struct list_head *list,
                    member);
list_for_each_entry_safe(type *cursor,
                         type *next,
                         struct list_head *list,
                         member);
```

Example

```
void todo_add_entry(struct todo_struct *new)
{
  struct list_head *ptr;
  struct todo_struct *entry;
  list_for_each(ptr, &todo_list) {
    entry = list_entry(ptr, struct todo_struct, list);
    if (entry->priority < new->priority) {
      list_add_tail(&new->list, ptr);
      return;
    }
  }
  list_add_tail(&new->list, &todo_struct)
}
```



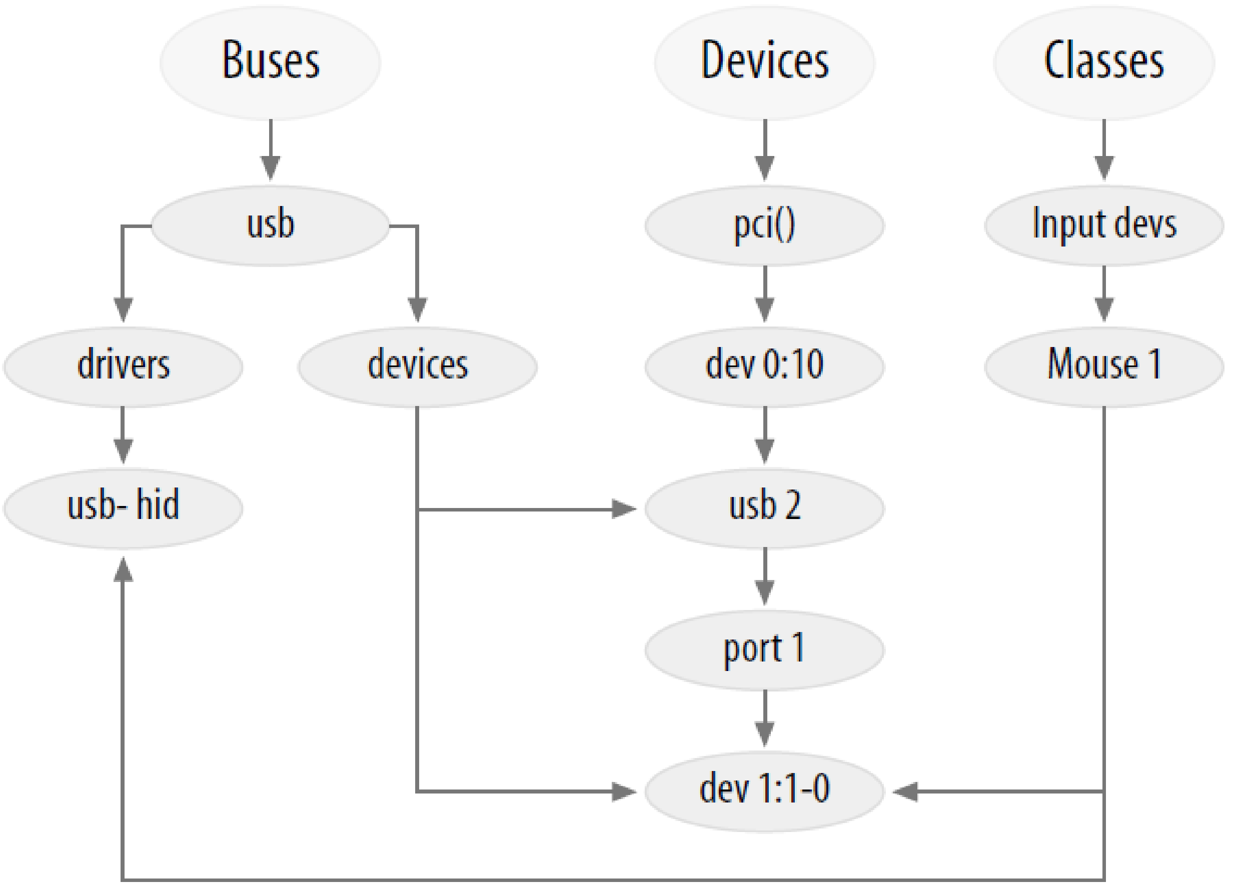
MAIK UNSHELM, tsamedien

The Linux Device Model

- Device classes
- Hot-pluggable devices
- Object lifecycles

Buses, Devices, and Drivers

- The core “devices” tree shows how the mouse is connected to the system
- The “bus” tree tracks what is connected to each bus
- The under “classes” concerns itself with the functions provided by the devices, regardless of how they are connected.



Kobject basics

- The *kobject* is the fundamental structure that holds the device model together
  - *Reference counting of objects*
  - *Sysfs representation*
  - *Data structure glue*
  - *Hotplug event handling*

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
};
```

- `struct cdev *device = container_of(kp, struct cdev, kobj);`

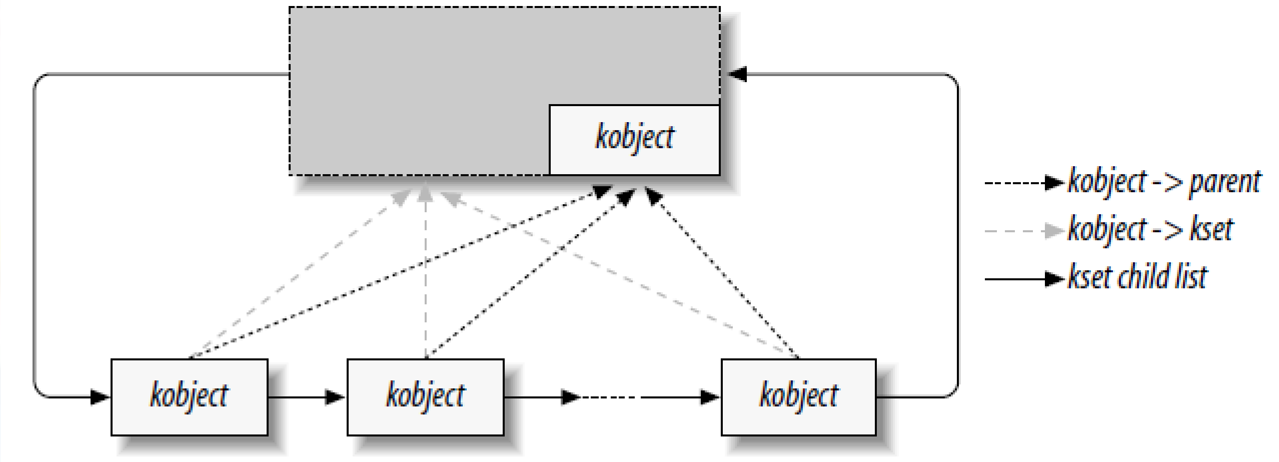
Kobject handling

```
void kobject_init(struct kobject *kobj);
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);

void my_object_release(struct kobject *kobj)
{
        struct my_object *mine = container_of(kobj,
                                  struct my_object, kobj);
        /* Perform any cleanup on this object, then... */
        kfree(mine);
}
```

Kobject Hierarchies, Ksets, and Subsystems

- At a glance… For experts ;)

```
void/int kobject_init/add/register/del(struct kobject *kobj);
void/int kset_init/add/register/del(struct kset *kset);
void/int subsystem_init/un/register(struct subsystem *subsys);
```



Kobject types

- The *release* method is not stored in the *kobject* itself
- It is associated with the type of the structure that contains the kobject

```
struct kobj_type {
        void (*release)(struct kobject *);
        struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;
};

struct kobj_type *get_ktype(struct kobject *kobj);
```

Low-Level Sysfs Operations

- Kobjects are the mechanism behind the sysfs virtual filesystem.
  - For every directory found in sysfs, there is a *kobject*
- Every kobject exports some attributes, which appear in that *kobject*'s sysfs directory as files containing kernel-generated information.
  - Sysfs entries for kobjects are always directories, so a call to *kobject_add* results in the creation of a directory in sysfs
  - The name assigned to the kobject (with *kobject_set_name*) is the name used for the sysfs directory
  - The sysfs entry is located in the directory corresponding to the kobject's parent pointer. If parent is NULL when *kobject_add* is called,it is set to the *kobject* embedded in the new kobject's kset; thus,the sysfs hierarchy usually matches the internal hierarchy created with ksets
- For example, */sys/devices* sysfs represents all system devices

Sysfs ops & params

```
struct attribute {
        char *name;
        struct module *owner;
        mode_t mode; /* S_IRUGO read-only,
                        S_IWUSR write access to root only */
};

struct sysfs_ops {
        ssize_t (*show)(struct kobject *kobj,
                        struct attribute *attr,
                        char *buffer);
        ssize_t (*store)(struct kobject *kobj,
                         struct attribute *attr,
                         const char *buffer, size_t size);
};
```

Non default attributes

- If you wish to add a new attribute to a kobject's sysfs directory, simply fill in an attribute structure and pass it to:

```
int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
```

Binary attributes

- Handle larger chunks of binary data that must be passed, untouched, between user space and the device

```
int sysfs_create/remove_bin_file(struct kobject *kobj,
                                 struct bin_attribute *attr);
```

```
struct bin_attribute {
        struct attribute attr;
        size_t size;
        ssize_t (*read)(struct kobject *kobj, char *buffer,
                        loff_t pos, size_t size);
        ssize_t (*write)(struct kobject *kobj, char *buffer,
                         loff_t pos, size_t size);
};
```

Hotplug Event Generation

- A hotplug event is a notification to user space from the kernel that something has changed in the system’s configuration.
- They are generated whenever a *kobject* is created or destroyed
  - New device plugged in with a USB cable
- Hotplug events turn into an invocation of */sbin/hotplug* which can respond to each event by loading drivers, creating device nodes, mounting partitions, or taking any other action that is appropriate.
- Before the event is handed to user space,code associated with the *kobject* (or,more specifically,the kset to which it belongs) has the opportunity to add information for user space or to disable event generation entirely.

Hotplug Operations

- The filter hotplug operation is called whenever the kernel is considering generating an event for a given *kobject*. If filter returns 0,the event is not created.
- The *name* parameters is provided to user space when user-space hotplug programm is involked

```
struct kset_hotplug_ops {
       int (*filter)(struct kset *kset, struct kobject *kobj);
       char *(*name)(struct kset *kset, struct kobject *kobj);
       int (*hotplug)(struct kset *kset, struct kobject *kobj,
       char **envp, int num_envp, char *buffer,
       int buffer_size);
};
```

Hotplug environment variables

- Everything else that the hotplug script might want to know is passed in the environment. The *hotplug* method gives an opportunity to add useful environment variables
- *kset* and *kobject* describe the object for which the event is being generated. The *envp* array is a place to store additional environment variable definitions (in the usual NAME=value format); it has *num_envp* entries available. The variables themselves should be encoded into *buffer*, which is *buffer_size* bytes long.

```
int (*hotplug)(struct kset *kset, struct kobject *kobj,
               char **envp, int num_envp, char *buffer,
               int buffer_size);
```

Buses, Devices, and Drivers

- Not mandatory for basic drivers, but better to know
- What is happening inside the PCI,USB,etc. layers

Buses

- A channel between the processor and one or more devices
- All devices are connected via a bus, even if it is an internal, virtual,“ platform” bus
- Buses can plug into each other

```
struct bus_type {
        char *name;
        struct subsystem subsys;
        struct kset drivers;
        struct  kset devices;
        int (*match)(struct device *dev, struct device_driver *drv);
        struct device *(*add)(struct device * parent, char * bus_id);
        int (*hotplug) (struct device *dev, char **envp,
        int num_envp, char *buffer, int buffer_size);
        /* Some fields omitted */
};
```

Bus methods

- `match` : Whenever a new device or driver is added for this bus
  - return a nonzero value.
  - bus level, because the core kernel cannot know how to match
  - might be as simple as
    - return !strncmp(dev->bus_id, driver->name, strlen(driver->name));
- `hotplug` : This method allows the bus to add variables to the environment prior to the generation of a hotplug event in user space

```
envp[0] = buffer;
if (snprintf(buffer, buf_size,“MYBUS_VERSION=%s“,Version) >= buf_size)
    return -ENOMEM;
envp[1] = NULL;
return 0;
```

- Operation on all attached device or driver

```
int bus_for_each_dev(struct bus_type *bus, struct device *start,
                     void *data, int (*fn)(struct device *, void *));
```

Bus attributes

- Almost every layer in the Linux device model provides an interface for the addition of attributes

```
struct bus_attribute {
        struct attribute attr;
        ssize_t (*show)(struct bus_type *bus, char *buf);
        ssize_t (*store)(struct bus_type *bus, const char *buf,
        size_t count);
};
```

- Compile-time creation and initialization of bus_attribute structures:
  - `BUS_ATTR(name, mode, show, store);`
- Attributes belonging to a bus is created explicitly with*:*
  - `int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);`

Devices

- The device structure contains the information that the device model core needs to model the system

```
struct device {
        struct device *parent;
        struct kobject kobj;
        char bus_id[BUS_ID_SIZE];
        struct bus_type *bus;
        struct device_driver *driver;
        void *driver_data;
        void (*release)(struct device *dev);
};
```

- Device registration
  - `int device_register(struct device *dev);`

Device attributes

- Device entries in sysfs can have attributes.

```
struct device_attribute {
        struct attribute attr;
        ssize_t (*show)(struct device *dev, char *buf);
        ssize_t (*store)(struct device *dev, const char *buf,
                         size_t count);
};
```

- Compile-time creation and initialization of device_attribute structures:
  - `DEVICE_ATTR(name, mode, show, store);`
- Attributes belonging to a bus is created explicitly with*:*

-
  ```
  int device_create_file(struct device *device,
                         struct device_attribute *entry);
  ```

Device Drivers

- The device model tracks all of the drivers known to the system

```
struct device_driver {
        char *name;
        struct bus_type *bus;
        struct kobject kobj;
        struct list_head devices;
        int (*probe)(struct device *dev);
        int (*remove)(struct device *dev);
        void (*shutdown) (struct device *dev);
};
```

- Device registration
  - `int driver_register(struct device_driver *drv);`

Device Driver attributes

- Device entries in sysfs can have attributes.

```
struct driver_attribute {
        struct attribute attr;
        ssize_t (*show)(struct device_driver *drv, char *buf);
        ssize_t (*store)(struct device_driver *drv, const char *buf,
        size_t count);
};
```

- Compile-time creation and initialization of device_attribute structures:
  - `DEVICE_ATTR(name, mode, show, store);`
- Attributes belonging to a bus is created explicitly with*:*

-
  ```
  int device_create_file(struct device *device,
                         struct device_attribute *entry);
  ```

Classes

- A class is a higher-level view of a device that abstracts out low-level implementation details.
- Drivers may see a SCSI disk or an ATA disk, but at the class level, they are all simply disks. Classes allow user space to work with devices based on what they do, rather than how they are connected or how they work.
- Classe_device, registration, attribute…

Platform drivers & embedded systems

- On embedded systems, devices are often not connected through a bus allowing enumeration, hotplugging, and providing unique identifiers for devices.
- However, we still want the devices to be part of the device model.
- The solution to this is the platform driver / platform device infrastructure.
- The platform devices are the devices that are directly connected to the CPU, without any kind of bus.

Initialization of a platform driver

- Example of the iMX serial port driver, in drivers/serial/imx.c.
- The driver instantiates a platform driver structure:

```
static struct platform_driver serial_imx_driver = {
        .probe = serial_imx_probe,
        .remove = serial_imx_remove,
        .driver = {
                .name = "imx-uart",
                .owner = THIS_MODULE,
        },
};
```

And registers/unregisters it at init/cleanup:

```
platform_driver_register(&serial_imx_driver);
```

Instantiation of a platform device

- As platform devices cannot be detected dynamically, they are statically defined, direct instantiation of platform device structures on ARM
- The matching between a device and the driver is simply done using the name.

```
static struct platform_driver serial_imx_driver = {
        .probe = serial_imx_probe,
        .remove = serial_imx_remove,
        .driver = {
                .name = "imx-uart",
                .owner = THIS_MODULE,
        },
};
```

Display Subsystem Support

HDMI Tx integration

Display Subsystem Support

- Extract from driver/video/omap2 in linux kernel 2.6.29 of l25.11
- The DSS manages different types of display and gives common API for them:
  - Display: roof display interface used for all display
  - Manager: display manager
  - RFBI: MIPI DBI, or RFBI (Remote Framebuffer Interface), support
  - DispC: the new external IRQ handler framework allows multiple consumers to register arbitrary IRQ mask
  - DPI:
  - SDI: Serial Display Interface
  - DSI:
  - Overlay:
  - VENC: Video Encoder support
- It has been reviewed on linux-omap and linux-fbdev-devel mailing lists. The patches can be found from http://gitorious.org/linux-omap-dss2/linux
  - **From**: Tomi Valkeinen tomi.valkeinen@nokia.com
  - **To**: linux-kernel@vger.kernel.org
  - **Subject**: [PATCH 00/18] OMAP: DSS2: Intro

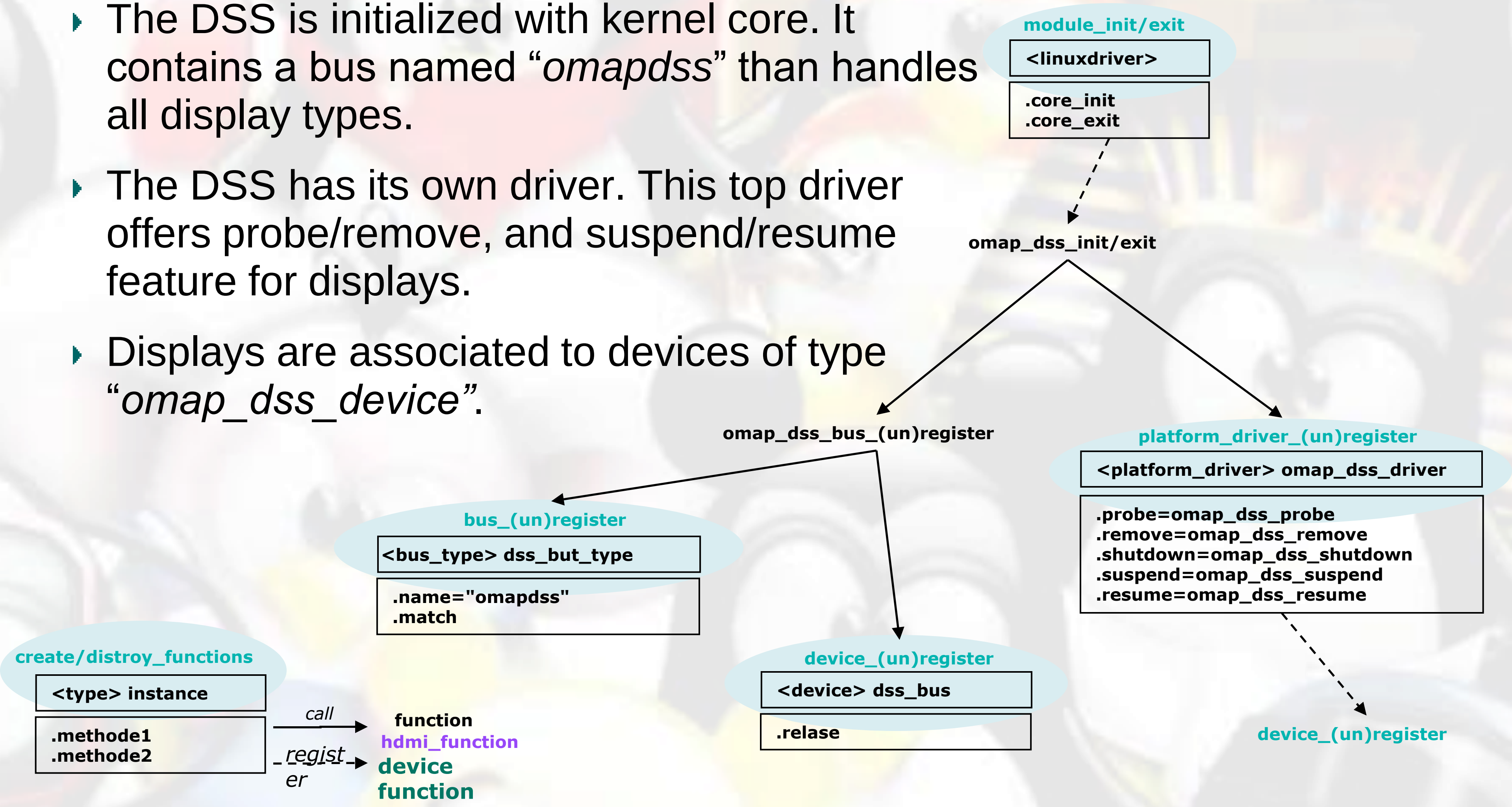
Display drivers in Omap2

- Supported display:
  - Acx565akm LCD panel (from Sony ?)
  - Nokia N800 Internet Tablette
  - LTE430WQ : Samsung 4.3 Inch LCD Panel
  - LS037V7DW01 : Sharp 3.7 Inch LCD Panel
  - Sil9022 : HDMI Tx from Silicon Image

DSS core

/driver/video/omap2/dss

- The DSS is initialized with kernel core. It contains a bus named “*omapdss*” than handles all display types.
- The DSS has its own driver. This top driver offers probe/remove, and suspend/resume feature for displays.
- Displays are associated to devices of type “*omap_dss_device*”.



module_init/exit

<linuxdriver>

.core_init
.core_exit

omap_dss_init/exit

omap_dss_bus_(un)register

platform_driver_(un)register

<platform_driver> omap_dss_driver

.probe=omap_dss_probe
.remove=omap_dss_remove
.shutdown=omap_dss_shutdown
.suspend=omap_dss_suspend
.resume=omap_dss_resume

bus_(un)register

<bus_type> dss_but_type

.name="omapdss"
.match

device_(un)register

<device> dss_bus

.relase

device_(un)register

create/distroy_functions

<type> instance

.methode1
.methode2

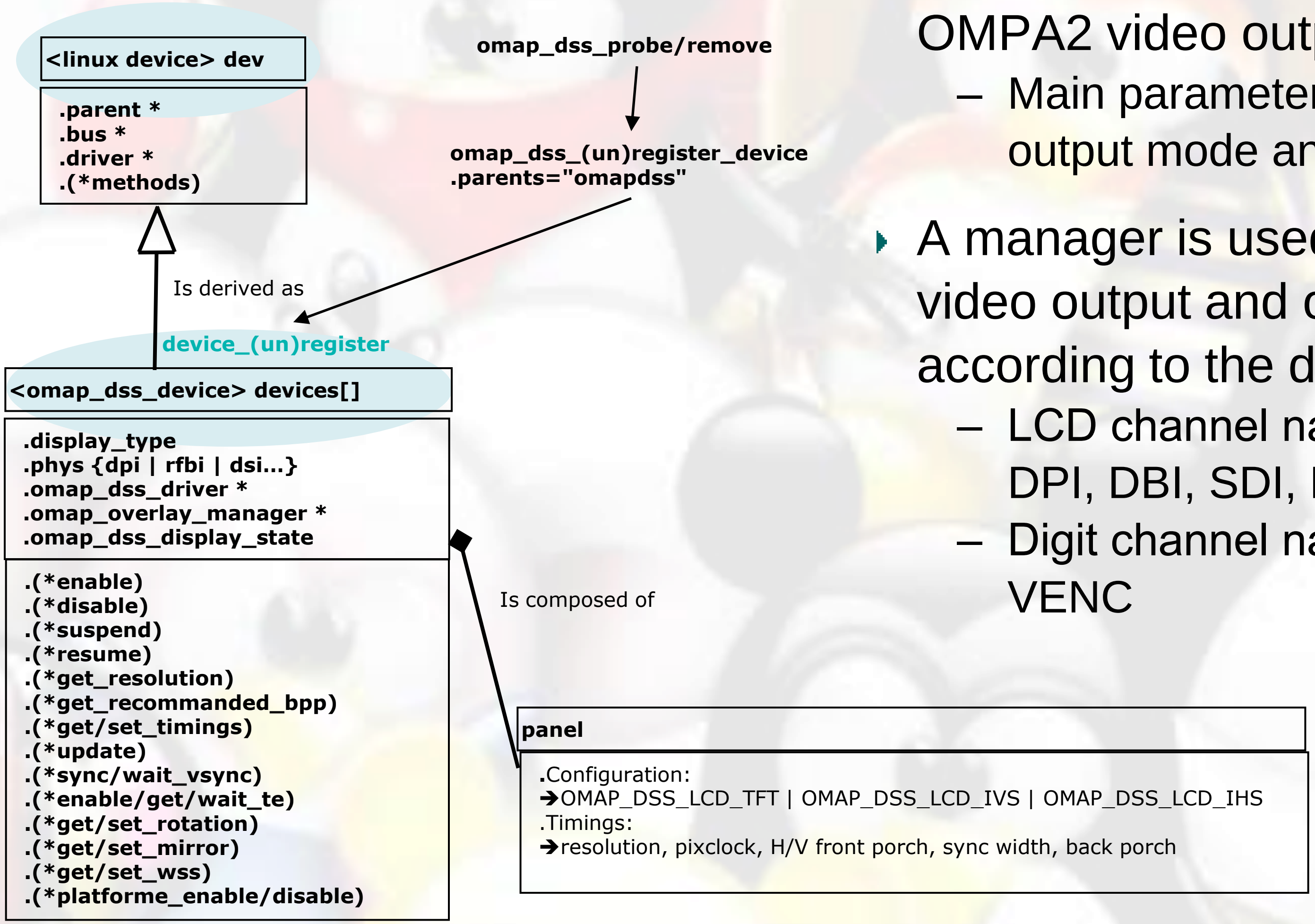
function

hdmi_function

device function

DSS devices

/driver/video/omap2/dss



<linux device> dev

.parent *
.bus *
.driver *
.(*methods)

omap_dss_probe/remove

omap_dss_(un)register_device
.parents="omapdss"

Is derived as

device_(un)register

<omap_dss_device> devices[]

.display_type
.phys {dpi | rfbi | dsi…}
.omap_dss_driver *
.omap_overlay_manager *
.omap_dss_display_state

.(*enable)
.(*disable)
.(*suspend)
.(*resume)
.(*get_resolution)
.(*get_recommanded_bpp)
.(*get/set_timings)
.(*update)
.(*sync/wait_vsync)
.(*enable/get/wait_te)
.(*get/set_rotation)
.(*get/set_mirror)
.(*get/set_wss)
.(*platforme_enable/disable)

Is composed of

panel

.Configuration:
➔OMAP_DSS_LCD_TFT | OMAP_DSS_LCD_IVS | OMAP_DSS_LCD_IHS
.Timings:
➔resolution, pixclock, H/V front porch, sync width, back porch

- Each devices type has it's own piece of SW to setup the OMPA2 video output
  - Main parameters are video output mode and timings.
- A manager is used to control the video output and overlay according to the display type:
  - LCD channel named "*lcd*" for DPI, DBI, SDI, DSI and HDMI
  - Digit channel named "*tv*" for VENC

I want more…

Micro kernel style

- Contains only generic code
  - Synchronization, scheduling, IPC
- All layers are written in a independent style
  - Existing interfaces and protocols shall be used
  - Each module is like an independent process without any access to other modules context nor variables sharing
  - Better handling of resources, only the actives modules can access resources
- Linking with the kernel
  - 1 single compiled object per module
  - Statically with the kernel code

    *_OR_*
  - Dynamically called at boot time or on demand depending on kernel setup

Dynamic linking

- Module object is in a separated file
- Exported symbols are added into the system exported symbol list
  - Entry points of modules
- External process may have unreferenced elements
  - Unknown elements are listed in the import directory
- Link is done dynamically when the system calls the module.
  - A loader sets up the module execution
  - Items (symbols, functions) required by some external processes are added to their address spaces
  - Addresses cannot be known before but only computed at the module loading (potential issues)
- Example : see System.map on Android, /etc/ld.so.conf for Unix

Fichier objet

- Implémentation très partielle du concept de micronoyau
- Code pouvant être lié dynamiquement à l'exécution
  - Lors du boot (rc scripts)
  - A la demande (noyau configuré avec option CONFIG_KMOD)
- Mais le code peut aussi, généralement, être lié statiquement au code du noyau (approche monolithique traditionnelle)

Liaisons dynamiques

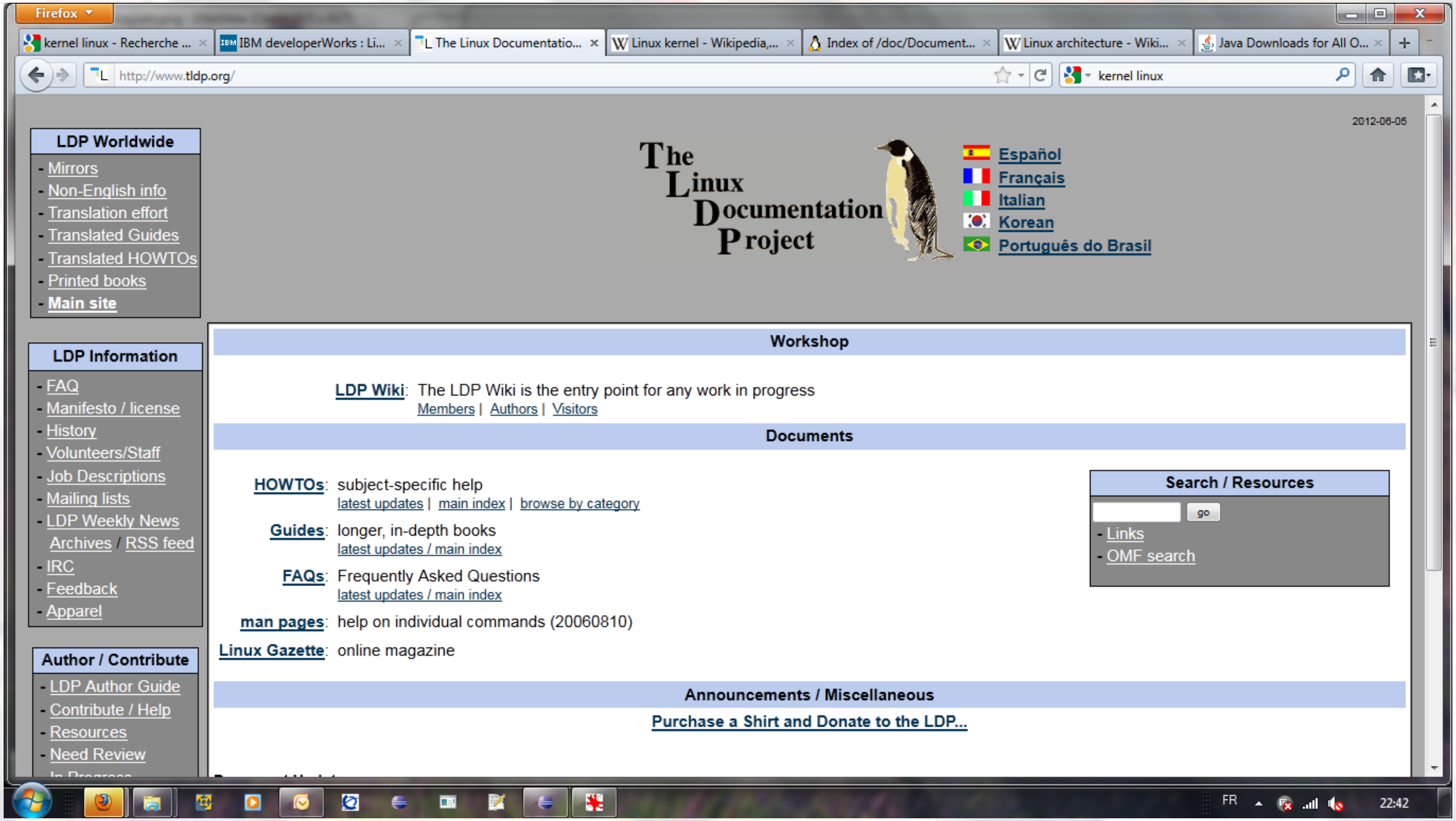
- Le code d’une librairie n’est pas copié dans l’exécutable à la compilation
- Il reste dans un fichier séparé
- Le linker ne fait pratiquement rien à la compilation
  - Il notes les librairies dont un exécutable à besoin
- Le gros du travail est fait au chargement ou à l’exécution
  - Par le loader de l’OS
- Le loader cherche/charge les bibliothèques dont un programme a besoin
  - Ajoute les éléments nécessaire à l’espace d’adressage du processus

Liaisons dynamiques

- Comportement différent suivant les OS
  - Chargement des bibliothèques au démarrage du processus
  - Chargement des bibliothèques quand le processus en a
- besoin (delay loading)
- Liaisons dynamiques développées pour Multics en 1960
- Utiliser la liaison dynamique implique des relocations
- Les adresses des sauts ne sont pas connues à la
- compilation
- Elles ne sont connues que lorsque le programme et les
- bibliothèques sont chargées
- Impossible de pré allouer des espaces
- Conflits
- Limitations de l'espace mémoire en 32 bits

Linux documentation

LDP


Firefox
kernel linux - Recherche ...
IBM developerWorks : Li...
The Linux Documentatio...
Linux kernel - Wikipedia,...
Index of /doc/Document...
Linux architecture - Wiki...
Java Downloads for All O...
http://www.tldp.org/
kernel linux
FR
22:42

2012-06-05

LDP Worldwide

- Mirrors
- Non-English info
- Translation effort
- Translated Guides
- Translated HOWTOs
- Printed books
- **Main site**

The Linux Documentation Project

Español
Français
Italian
Korean
Português do Brasil

LDP Information

- FAQ
- Manifesto / license
- History
- Volunteers/Staff
- Job Descriptions
- Mailing lists
- LDP Weekly News
  Archives / RSS feed
- IRC
- Feedback
- Apparel

Author / Contribute

- LDP Author Guide
- Contribute / Help
- Resources
- Need Review
- In Progress

Workshop

LDP Wiki: The LDP Wiki is the entry point for any work in progress
Members | Authors | Visitors

Documents

HOWTOs: subject-specific help
latest updates | main index | browse by category

Guides: longer, in-depth books
latest updates / main index

FAQs: Frequently Asked Questions
latest updates / main index

man pages: help on individual commands (20060810)

Linux Gazette: online magazine

Search / Resources

go

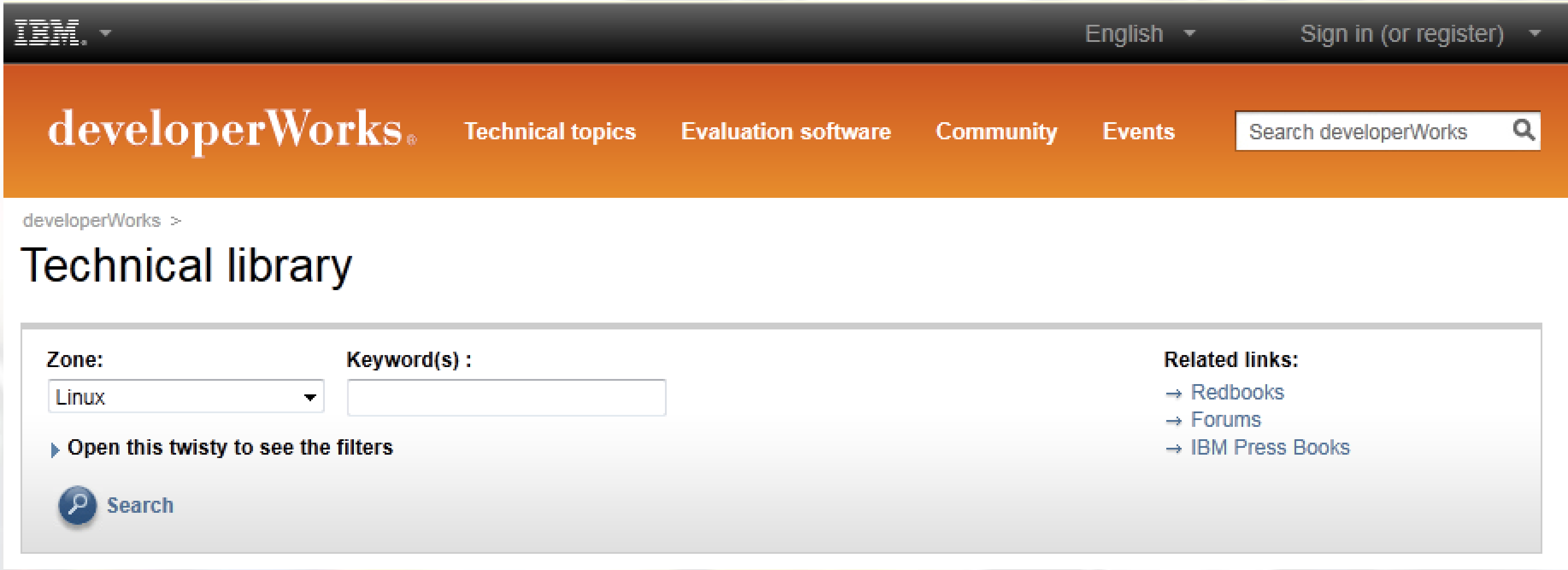
- Links
- OMF search

Announcements / Miscellaneous

Purchase a Shirt and Donate to the LDP...

IBM developer works

- http://www.ibm.com/developerworks/views/linux/libraryview.jsp?type_by=Tutorials



Technical library

Zone: Linux

Keyword(s) :

Open this twisty to see the filters

Related links:
→ Redbooks
→ Forums
→ IBM Press Books

Kernel Documentation

- ABI/ - info on kernel <-> userspace ABI and relative interface stability. BUG-HUNTING - brute force method of doing binary search of patches to find bug. Changes - list of changes that break older software packages. CodingStyle - how the boss likes the C code in the kernel to look. development-process/ - An extended tutorial on how to work with the kernel development process. DMA-API.txt - DMA API, pci_ API & extensions for non-consistent memory machines. DMA-ISA-LPC.txt - How to do DMA with ISA (and LPC) devices. DocBook/ - directory with DocBook templates etc. for kernel documentation. HOWTO - the process and procedures of how to do Linux kernel development. IPMI.txt - info on Linux Intelligent Platform Management Interface (IPMI) Driver. IRQ-affinity.txt - how to select which CPU(s) handle which interrupt events on SMP. IRQ.txt - description of what an IRQ is. ManagementStyle - how to (attempt to) manage kernel hackers. RCU/ - directory with info on RCU (read-copy update). SAK.txt - info on Secure Attention Keys. SM501.txt - Silicon Motion SM501 multimedia companion chip SecurityBugs - procedure for reporting security bugs found in the kernel. SubmitChecklist - Linux kernel patch submission checklist. SubmittingDrivers - procedure to get a new driver source included into the kernel tree. SubmittingPatches - procedure to get a source patch included into the kernel tree. VGA-softcursor.txt - how to change your VGA cursor from a blinking underscore. accounting/ - documentation on accounting and taskstats. acpi/ - info on ACPI-specific hooks in the kernel. aoe/ - description of AoE (ATA over Ethernet) along with config examples. applying-patches.txt - description of various trees and how to apply their patches. arm/ - directory with info about Linux on the ARM architecture. atomic_ops.txt - semantics and behavior of atomic and bitmask operations. auxdisplay/ - misc. LCD driver documentation (cfag12864b, ks0108). basic_profiling.txt - basic instructions for those who wants to profile Linux kernel. binfmt_misc.txt - info on the kernel support for extra binary formats. blackfin/ - directory with documentation for the Blackfin arch. block/ - info on the Block I/O (BIO) layer. blockdev/ - info on block devices & drivers btmrvl.txt - info on Marvell Bluetooth driver usage. bus-virt-phys-mapping.txt - how to access I/O mapped memory from within device drivers. cachetlb.txt - describes the cache/TLB flushing interfaces Linux uses. cdrom/ - directory with information on the CD-ROM drivers that Linux has. cgroups/ - cgroups features, including cpusets and memory controller. connector/ - docs on the netlink based userspace<->kernel space communication mod. console/ - documentation on Linux console drivers. cpu-freq/ - info on CPU frequency and voltage scaling. cpu-hotplug.txt - document describing CPU hotplug support in the Linux kernel. cpu-load.txt - document describing how CPU load statistics are collected. cpuidle/ - info on CPU_IDLE, CPU idle state management subsystem. cputopology.txt - documentation on how CPU topology info is exported via sysfs. cris/ - directory with info about Linux on CRIS architecture. crypto/ - directory with info on the Crypto API. dcdbas.txt - information on the Dell Systems Management Base Driver. debugging-modules.txt - some notes on debugging modules after Linux 2.6.3. dell_rbu.txt - document demonstrating the use of the Dell Remote BIOS Update driver. device-mapper/ - directory with info on Device Mapper. devices.txt - plain ASCII listing of all the nodes in /dev/ with major minor #'s. dontdiff - file containing a list of files that should never be diff'ed. driver-model/ - directory with info about Linux driver model. dvb/ - info on Linux Digital Video Broadcast (DVB) subsystem. early-userspace/ - info about initramfs, klibc, and userspace early during boot. edac.txt - information on EDAC - Error Detection And Correction eisa.txt - info on EISA bus support. fault-injection/ - dir with docs about the fault injection capabilities infrastructure. fb/ - directory with info on the frame buffer graphics abstraction layer. feature-removal-schedule.txt - list of files and features that are going to be removed. filesystems/ - info on the vfs and the various filesystems that Linux supports. firmware_class/ - request_firmware() hotplug interface info. frv/ - Fujitsu FR-V Linux documentation. gpio.txt - overview of GPIO (General Purpose Input/Output) access conventions. highuid.txt - notes on the change from 16 bit to 32 bit user/group IDs. timers/ - info on the timer related topics hw_random.txt - info on Linux support for random number generator in i8xx chipsets. hwmon/ - directory with docs on various hardware monitoring drivers. i2c/ - directory with info about the I2C bus/protocol (2 wire, kHz speed). i2o/ - directory with info about the Linux I2O subsystem. x86/i386/ - directory with info about Linux on Intel 32 bit architecture. ia64/ - directory with info about Linux on Intel 64 bit architecture. infiniband/ - directory with documents concerning Linux InfiniBand support. initrd.txt - how to use the RAM disk as an initial/temporary root filesystem. input/ - info on Linux input device support. io-mapping.txt - description of io_mapping functions in linux/io-mapping.h io_ordering.txt - info on ordering I/O writes to memory-mapped addresses. ioctl/ - directory with documents describing various IOCTL calls. iostats.txt - info on I/O statistics Linux kernel provides. irqflags-tracing.txt - how to use the irq-flags tracing feature. isapnp.txt - info on Linux ISA Plug & Play support. isdn/ - directory with info on the Linux ISDN support, and supported cards. java.txt - info on the in-kernel binary support for Java(tm). kbuild/ - directory with info about the kernel build process. kdump/ - directory with mini HowTo on getting the crash dump code to work. kernel-doc-nano-HOWTO.txt - mini HowTo on generation and location of kernel documentation files. kernel-docs.txt - listing of various WWW + books that document kernel internals. kernel-parameters.txt - summary listing of command line / boot prompt args for the kernel. kobject.txt - info of the kobject infrastructure of the Linux kernel. kprobes.txt - documents the kernel probes debugging feature. kref.txt - docs on adding reference counters (krefs) to kernel objects. laptops/ - directory with laptop related info and laptop driver documentation. ldm.txt - a brief description of LDM (Windows Dynamic Disks). leds/ - directory with info about LED handling under Linux. local_ops.txt - semantics and behavior of local atomic operations. lockdep-design.txt - documentation on the runtime locking correctness validator. logo.gif - full colour GIF image of Linux logo (penguin - Tux). logo.txt - info on creator of above logo & site to get additional images from. m68k/ - directory with info about Linux on Motorola 68k architecture. magic-number.txt - list of magic numbers used to mark/protect kernel data structures. mca.txt - info on supporting Micro Channel Architecture (e.g. PS/2) systems. md.txt - info on boot arguments for the multiple devices driver. memory-barriers.txt - info on Linux kernel memory barriers. memory-hotplug.txt - Hotpluggable memory support, how to use and current status. memory.txt - info on typical Linux memory problems. mips/ - directory with info about Linux on MIPS architecture. mmc/ - directory with info about the MMC subsystem mono.txt - how to execute Mono-based .NET binaries with the help of BINFMT_MISC. mutex-design.txt - info on the generic mutex subsystem. namespaces/ - directory with various information about namespaces netlabel/ - directory with information on the NetLabel subsystem. networking/ - directory with info on various aspects of networking with Linux. nmi_watchdog.txt - info on NMI watchdog for SMP systems. nommu-mmap.txt - documentation about no-mmu memory mapping support. numastat.txt - info on how to read Numa policy hit/miss statistics in sysfs. oops-tracing.txt - how to decode those nasty internal kernel error dump messages. padata.txt - An introduction to the "padata" parallel execution API parisc/ - directory with info on using Linux on PA-RISC architecture. parport.txt - how to use the parallel-port driver. parport-lowlevel.txt - description and usage of the low level parallel port functions. pcmcia/ - info on the Linux PCMCIA driver. pi-futex.txt - documentation on lightweight PI-futexes. pnp.txt - Linux Plug and Play documentation. power/ - directory with info on Linux PCI power management. powerpc/ - directory with info on using Linux with the PowerPC. preempt-locking.txt - info on locking under a preemptive kernel. printk-formats.txt - how to get printk format specifiers right prio_tree.txt - info on radix-priority-search-tree use for indexing vmas. rbtree.txt - info on what red-black trees are and what they are for. robust-futex-ABI.txt - documentation of the robust futex ABI. robust-futexes.txt - a description of what robust futexes are. rt-mutex-design.txt - description of the RealTime mutex implementation design. rt-mutex.txt - desc. of RT-mutex subsystem with PI (Priority Inheritance) support. rtc.txt - notes on how to use the Real Time Clock (aka CMOS clock) driver. s390/ - directory with info on using Linux on the IBM S390. scheduler/ - directory with info on the scheduler. scsi/ - directory with info on Linux scsi support. security/ - directory that contains security-related info serial/ - directory with info on the low level serial API. serial-console.txt - how to set up Linux with a serial line console as the default. sgi-ioc4.txt - description of the SGI IOC4 PCI (multi function) device. sgi-visws.txt - short blurb on the SGI Visual Workstations. sh/ - directory with info on porting Linux to a new architecture. sound/ - directory with info on sound card support. sparc/ - directory with info on using Linux on Sparc architecture. sparse.txt - info on how to obtain and use the sparse tool for typechecking. spi/ - overview of Linux kernel Serial Peripheral Interface (SPI) support. spinlocks.txt - info on using spinlocks to provide exclusive access in kernel. stable_api_nonsense.txt - info on why the kernel does not have a stable in-kernel api or abi. stable_kernel_rules.txt - rules and procedures for the -stable kernel releases. svga.txt - short guide on selecting video modes at boot via VGA BIOS. sysfs-rules.txt - How not to use sysfs. sysctl/ - directory with info on the /proc/sys/* files. sysrq.txt - info on the magic SysRq key. telephony/ - directory with info on telephony (e.g. voice over IP) support. unicode.txt - info on the Unicode character/font mapping used in Linux. unshare.txt - description of the Linux unshare system call. usb/ - directory with info regarding the Universal Serial Bus. video-output.txt - sysfs class driver interface to enable/disable a video output device. video4linux/ - directory with info regarding video/TV/radio cards and linux. vm/ - directory with info on the Linux vm code. volatile-considered-harmful.txt - Why the "volatile" type class should not be used w1/ - directory with documents regarding the 1-wire (w1) subsystem. watchdog/ - how to auto-reboot Linux if it has "fallen and can't get up". ;-) x86/x86_64/ - directory with info on Linux support for AMD x86-64 (Hammer) machines. zorro.txt - info on writing drivers for Zorro bus devices found on Amigas.

http://www.kernel.org/doc/Documentation

www.kernel.org/doc/Documentation 1/4

- A GUIDE TO THE KERNEL DEVELOPMENT PROCESS
- CodingStyle - how the boss likes the C code in the kernel to look
- DMA-API.txt - DMA API for non-consistent memory machines
- HOWTO - the process and procedures of how to do Linux kernel development
- IRQ.txt - description of what an IRQ is.
- arm/ - directory with info about Linux on the ARM architecture.
- block/ - info on the Block I/O (BIO) layer
- bus-virt-phys-mapping.txt - how to access I/O mapped memory from within device drivers

www.kernel.org/doc/Documentation 2/4

- console/ - documentation on Linux console drivers.
- cpu-load.txt - how CPU load statistics are collected
- crypto/ - directory with info on the Crypto API.
- debugging-modules.txt - notes on debugging modules after Linux 2.6.3.
- devices.txt - ASCII listing of all the nodes in /dev/ with major minor #'s.
- fb/ - directory with info on the frame buffer graphics abstraction layer.
- filesystems/ - info on the filesystems that Linux supports.
- gpio.txt - overview of GPIO access conventions.
- timers/ - info on the timer related topics
- i2c/ - directory with info about the I2C bus/protocol (2 wire, kHz speed).

www.kernel.org/doc/Documentation 3/4

- ioctl/ - directory with documents describing various IOCTL calls.
- kernel-docs.txt - listing of various WWW + books about kernel internals
- kobject.txt - info of the kobject infrastructure of the Linux kernel
- local_ops.txt - semantics and behavior of local atomic operations.
- memory-barriers.txt - info on Linux kernel memory barriers.
- memory.txt - info on typical Linux memory problems.
- mutex-design.txt - info on the generic mutex subsystem.
- nommu-mmap.txt - no-mmu memory mapping support.
- oops-tracing.txt - how to decode those nasty internal kernel error

www.kernel.org/doc/Documentation 4/4

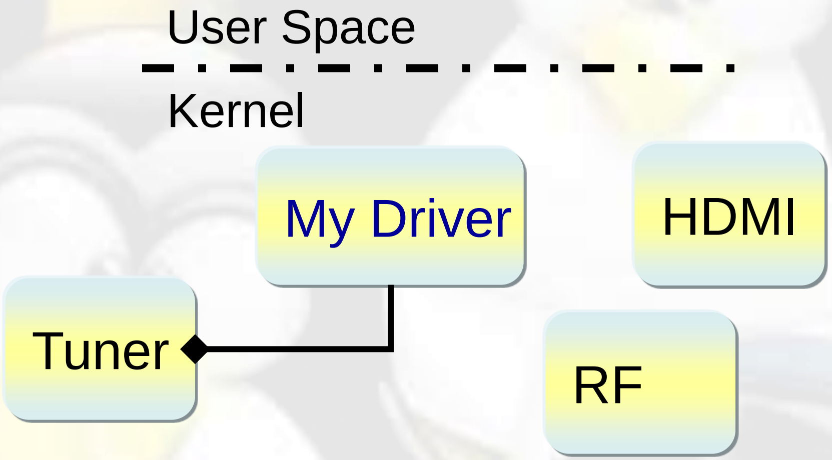
- pnp.txt - Linux Plug and Play documentation
- printk-formats.txt - how to get printk format specifiers right
- robust-futex-ABI.txt - documentation of the robust futex ABI.
- vm/ - directory with info on the Linux vm code.
- spinlocks.txt - info on using spinlocks to provide exclusive access in kernel.

Tips

Découvrir le Tux, c'est facile!
TUX
POUR
LES NULS
La première collection Informatique dans le monde
A mettre entre toutes les mains!
Cisoun

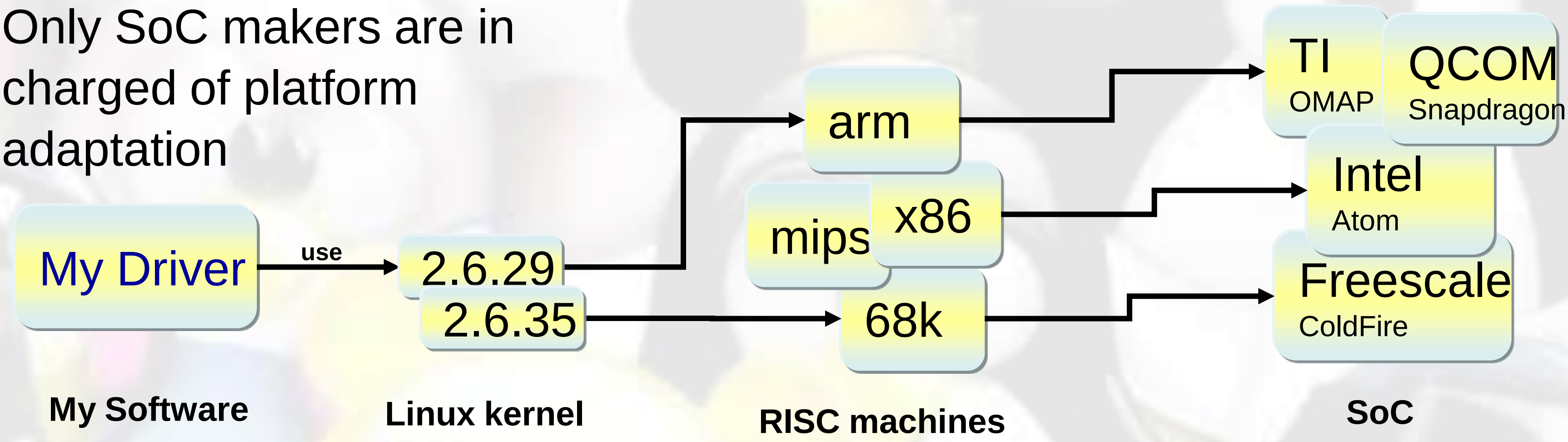
No architecture effort

- No specification documentation but a 2 pages “how-to” for developers
- Only if you are the first one to provide a feature, no architecture specification is needed. The Linux kernel supports a tremendous set of interfaces already optimized that fit each others and guarantee the kernel stability. Just be inspired while copying some to create yours
- Others vendors have already set their own drivers, just find the right place to put the right driver
- **Get legacy**, your driver should be “*a kind of*” with inherited structures



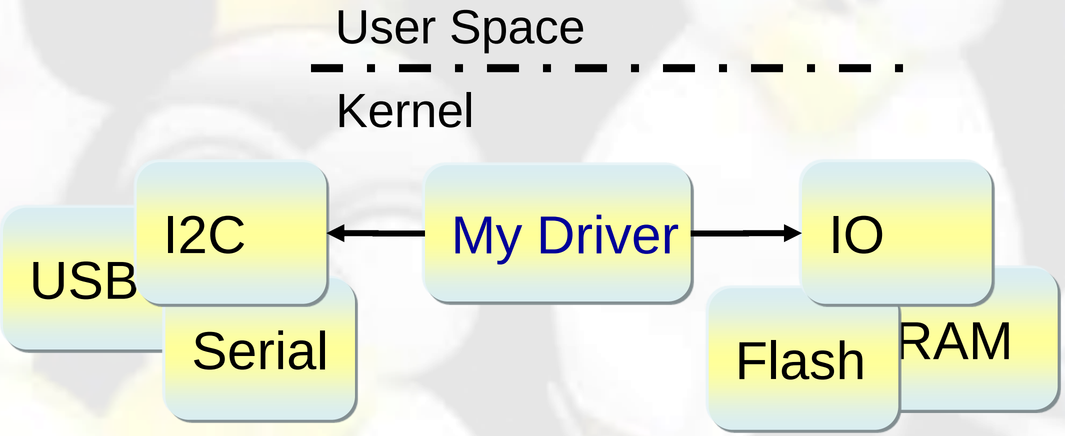
Use a kernel, not a platform

- Linux supports many interfaces. They are all deeply integrated in the Linux kernel
- Linux kernel version indicates a interfaces set maturity level in terms of hardware abstraction layer and generic platform adaptation
- Indeed, do not adapt to a platform but “**use” a Linux kernel version** that supports it
- Only SoC makers are in charged of platform adaptation



Hardware abstraction layering

- Common hardware abstraction interfaces like I2C, virtual memory, register mapping, GPIO, IRQ are already up and running, so just use it and do NOT re-invent the wheel…
- Linux kernel provides all necessary objects with their methods, just **“declare” what you use** and **implement your back-end**



I2C example

1. Declare some I2C client

```
struct i2c_device_id this_i2c_id[] = {
   { "TDA19988",0},
};
MODULE_DEVICE_TABLE(i2c, this_i2c_id);
```

2. Provide its probe/remove methods with your driver specific stuff
   - Like some I2C start-up/stop-down sequence for the TDA

```
struct i2c_driver this_i2c_driver = {
   .id_table = this_i2c_id,
   .probe = this_i2c_probe,
   .remove = this_i2c_remove,
};
err = i2c_add_driver(&this_i2c_driver);
```

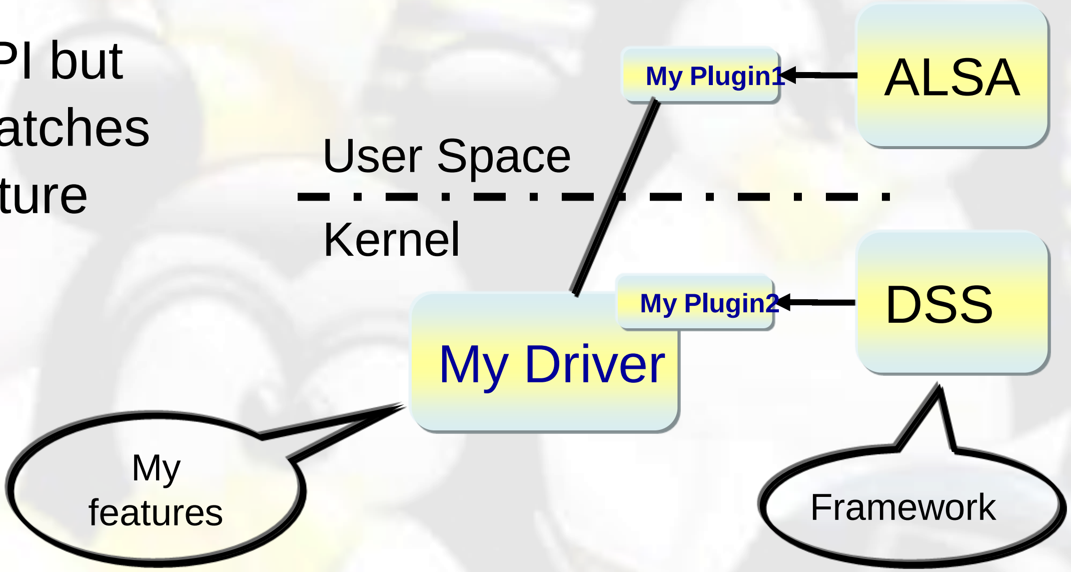
```
int this_i2c_probe(struct i2c_client *client,
                   const struct i2c_device_id *id)
{
   i2c_set_clientdata(client,…);
   my_tda_init(…);
}
```

3. Map your own R/W register functions on your I2C client

```
int my_block_write(struct i2c_client *client, u8 reg,
                   u16 alength, u8 *val)
{
   struct i2c_msg msg[1];

   msg->addr = client->addr;
   msg->flags = I2C_M_WR;
   msg->len = alength;
   memcpy(msg->buf, val, alength);

   return( i2c_transfer(client->adapter, msg, 1));
}
```

Plugins

- Overall system and framework description is needed to figure out the customer requirements and how you can provide your features
- Main concern is the “features v.s. framework” mapping
- Most part of features might be directly handled by the system using small interfacing piece of software called plugins.
- Do not provide proprietary API but design your functions so it matches existing system objects structure
- Provide small plugins in both kernel and user space with your core driver



Display Sub-System plugin example

1. Declare some DSS objects

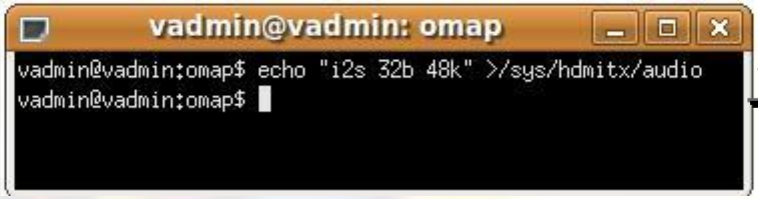
```
static struct omap_dss_driver hdmi_driver = {
   .enable = hdmi_panel_enable,
   .disable = hdmi_panel_disable,
   .suspend = hdmi_panel_suspend,
   .resume = hdmi_panel_resume,
};
```

2. Provide its methods with your driver specific stuff
   - Like some power up/down sequence for the TDA

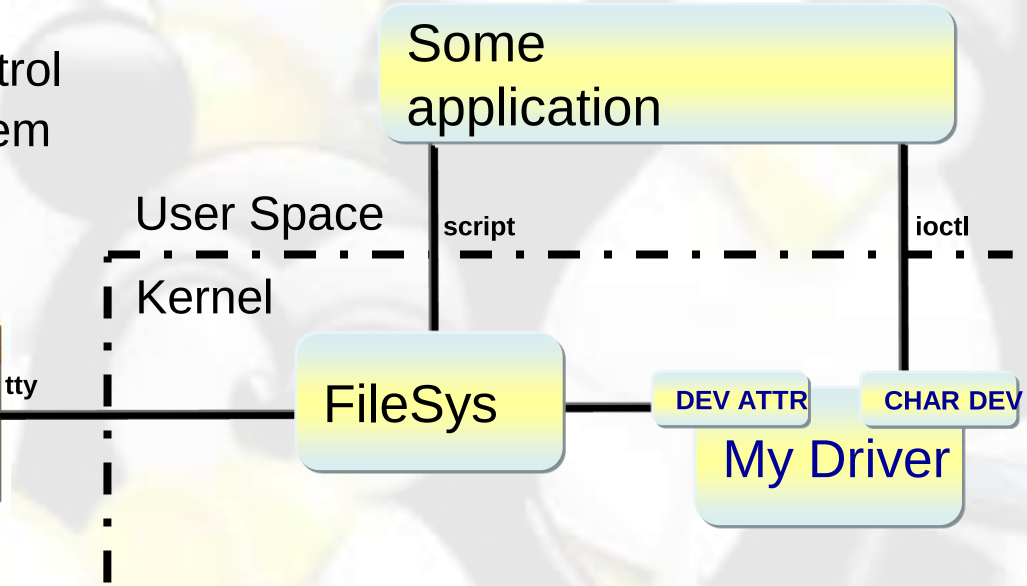
```
static int hdmi_panel_enable(struct omap_dss_device *dssdev)
{
   dssdev->platform_enable(dssdev);
   my_power_up(…);
   return 0;
}
```

Fine tuning of the driver during runtime

- If your features can only be tuned with extra control that is not provided by the system, if you like to give legacy of your core driver interface from kernel to user space, if you like to embed your debugging tools set, export input output control of your driver using IOCTL
- But if the customer does not like to compile some proprietary header file in the application, provide him device attribute control at runtime using Linux File System philosophy



```
vadmin@vadmin: omap
vadmin@vadmin:omap$ echo "i2s 32b 48k" >/sys/hdmitx/audio
vadmin@vadmin:omap$ █
```



Device attribute example

1. Declare some device attribute
   - Like audio, resolution, i2c

```
DEVICE_ATTR(audio, S_IRUGO|S_IWUSR, audio_show, audio_store);
DEVICE_ATTR(i2cW, S_IRUGO|S_IWUSR, NULL, i2cW_store);
```

2. Provide methods with your driver specific stuff
   - Like some i2c writting

```
ssize_t i2cW_store(struct device *dev,
                   struct device_attribute *attr,
                   const char *buf, size_t size)
{
   unsigned int page,address,mask,value;
   char desc_format[]="%x %x %x %x\n";

   sscanf(buf,desc_format,&page,&address,&mask,&value);
   return my_i2c_write(page,address,mask,value);
}
```

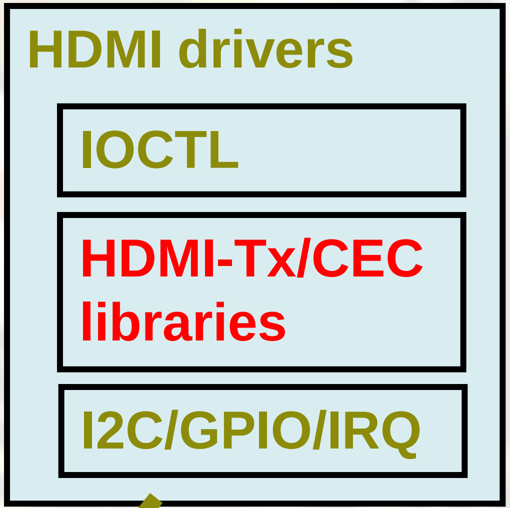
- **`user@host/$ echo “1 0x03 2” >/sys/hdmitx/i2cW`**
  write 0x02 in register 0x01 using mask 0x03

Linux module or Linux driver

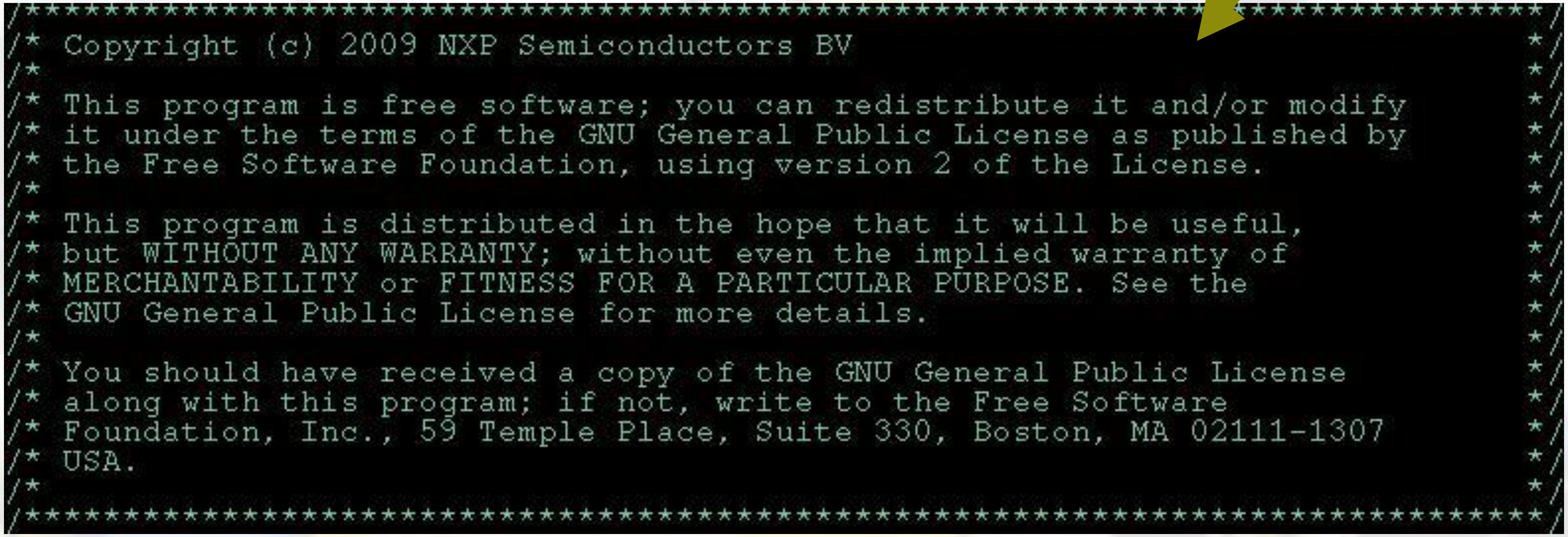
- Driver sources are in the kernel sources
- Some drivers may not be complied according to the Kernel configuration, some may be declared as module
- A module is a driver than is compiled apart but still need the kernel header files
- A compiled module can be uploaded and inserted in the target in live, I mean without recompiling the kernel nor flashing the platform nor rebooting the target

- Quiz : how many time for the kernel compilation ? For the module ?

Linux GPL license



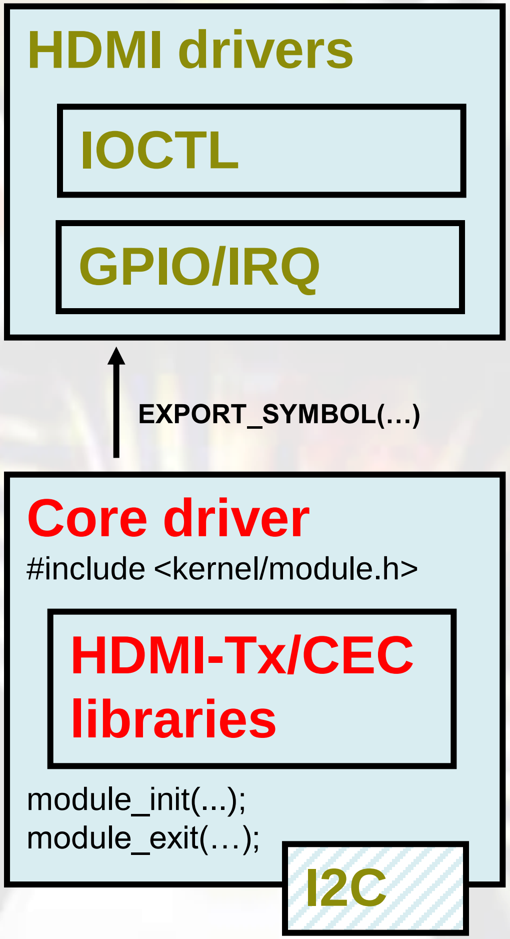
- HdmiTx and HdmiCec Linux drivers are **GPLv2**
  - Pure GPL software like GPIO and IRQ is used
  - MODULE_LICENSE(“GPL”)
- Core driver (DevLib and BSL libraries) is claimed as **proprietary**, but it is compiled with GPL sources. So it “becomes” GPL, even used as a module.

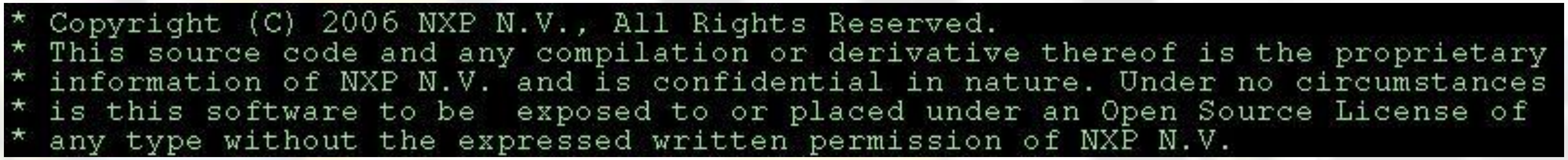


```
/*************************************************************************/
/* Copyright (c) 2009 NXP Semiconductors BV                              */
/*                                                                       */
/* This program is free software; you can redistribute it and/or modify  */
/* it under the terms of the GNU General Public License as published by  */
/* the Free Software Foundation, using version 2 of the License.          */
/*                                                                       */
/* This program is distributed in the hope that it will be useful,       */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of        */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the          */
/* GNU General Public License for more details.                          */
/*                                                                       */
/* You should have received a copy of the GNU General Public License     */
/* along with this program; if not, write to the Free Software           */
/* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307   */
/* USA.                                                                  */
/*                                                                       */
/*************************************************************************/
```

How to get ride of GPL license

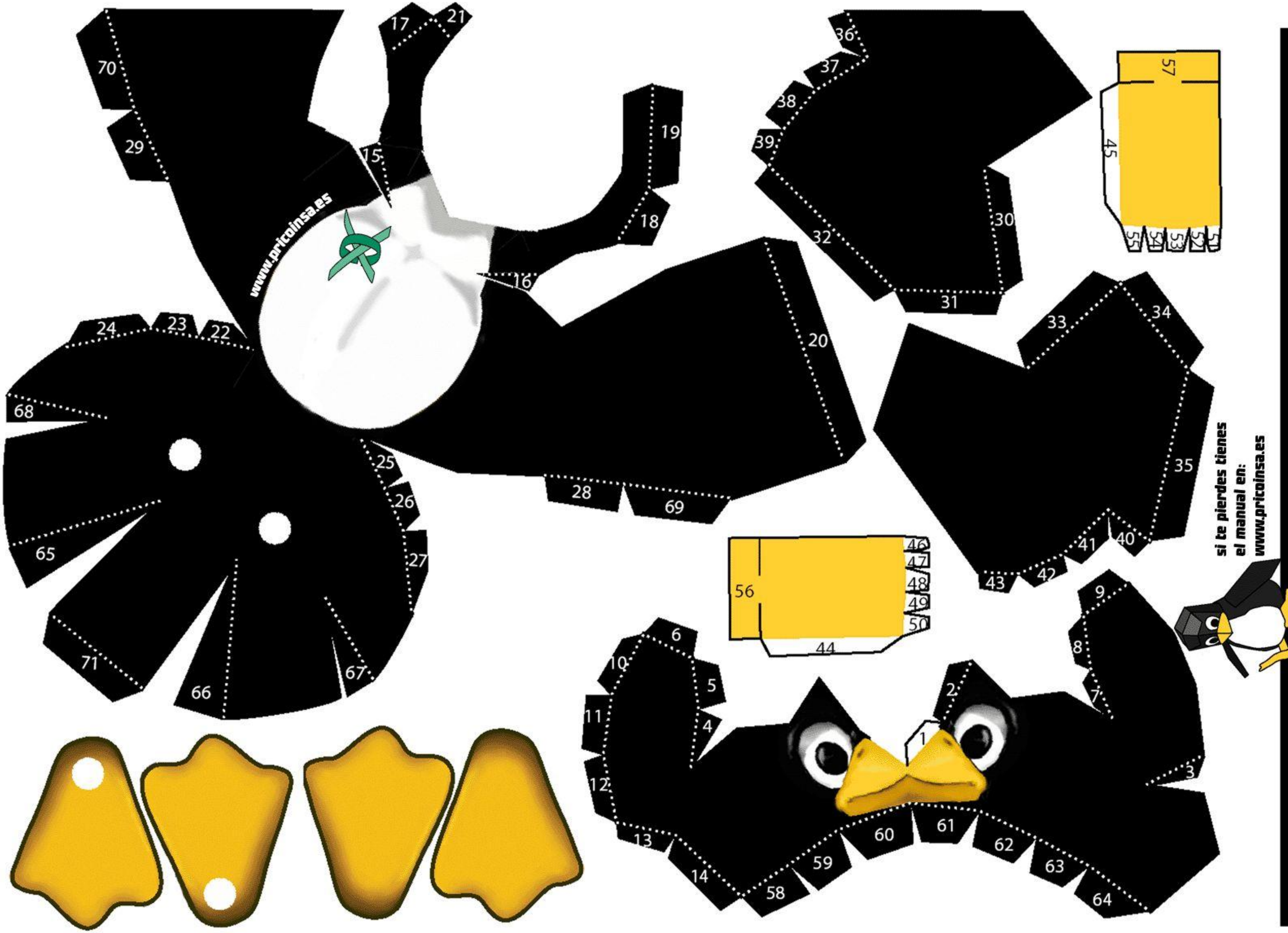
- Core driver (DevLib and BSL libraries) is claimed as **proprietary**
- If compiled without GPL dependency in a external module, it can be kept proprietary:
  - Only #include <kernel/module.h> is tolerated at compilation using module_init() and module_exit() Linux primitives
  - Then each function of core driver API should be exported in Linux kernel using EXPORT_SYMBOL()
    - See tmdlHdmiTx_Functions.h
  - If I2C client is not tolerated in the core driver, then export functions in another GPL driver.





```
* Copyright (C) 2006 NXP N.V., All Rights Reserved.
* This source code and any compilation or derivative thereof is the proprietary
* information of NXP N.V. and is confidential in nature. Under no circumstances
* is this software to be  exposed to or placed under an Open Source License of
* any type without the expressed written permission of NXP N.V.
```

Break !!!



si te pierdes tienes el manual en: www.pricoinsa.es