Linux Device Driver

André Lépine, Enseignement Linux Embarqué, 4 Décembre 2022

ENSI CAEN

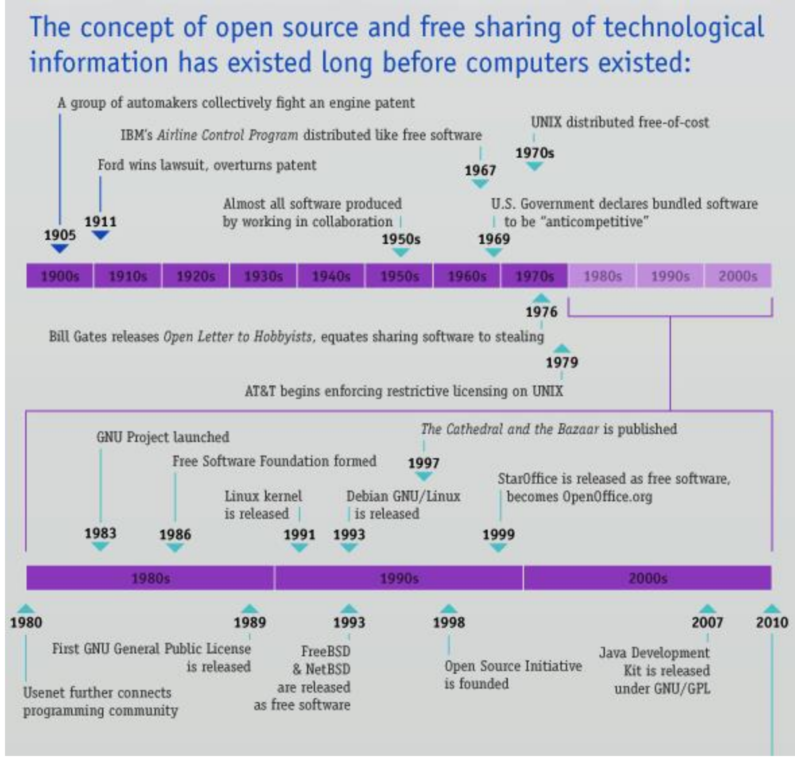
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

L'École des INGÉNIEURS Scientifiques

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)



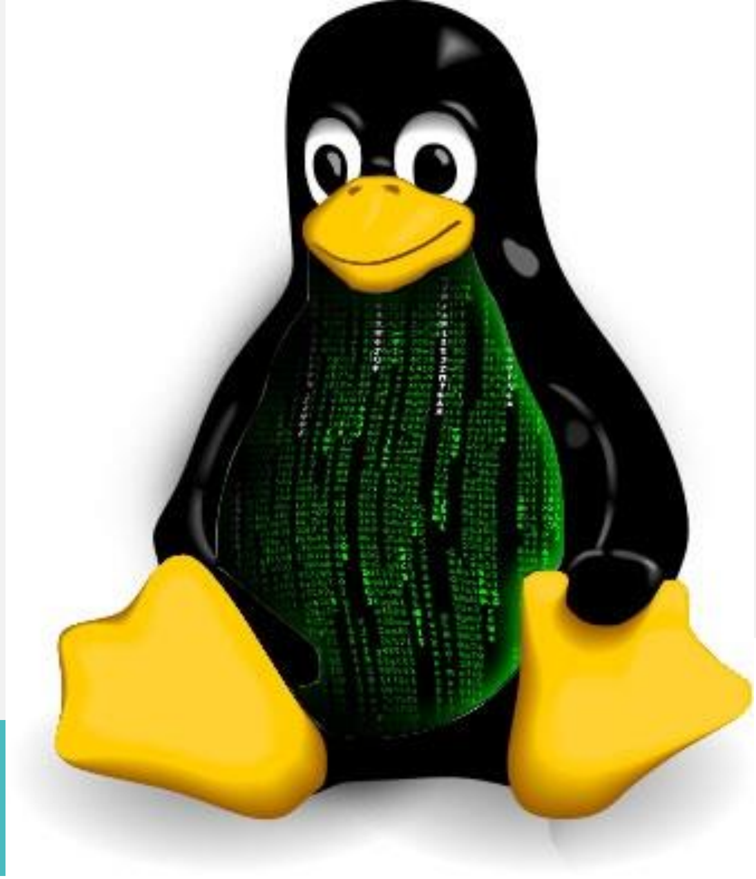
REFERENCES

- *"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If anything in this document becomes out of date, please send in patches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 Linux Kernel documentation — The Linux Kernel documentation](https://docs.kernel.org/index.html)
  https://docs.kernel.org/index.html
- *"The C programming language"*
  - Kernighan and Ritchie [Prentice Hall]

ZOOM INTO THE LINUX KERNEL

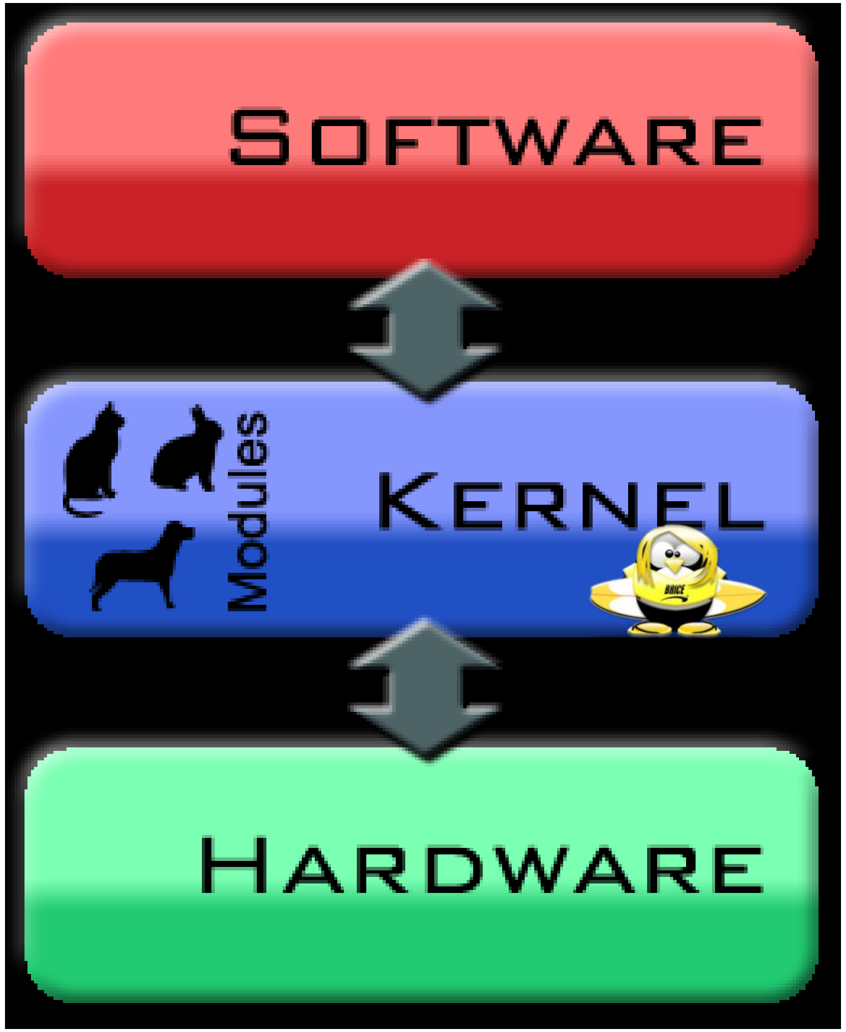


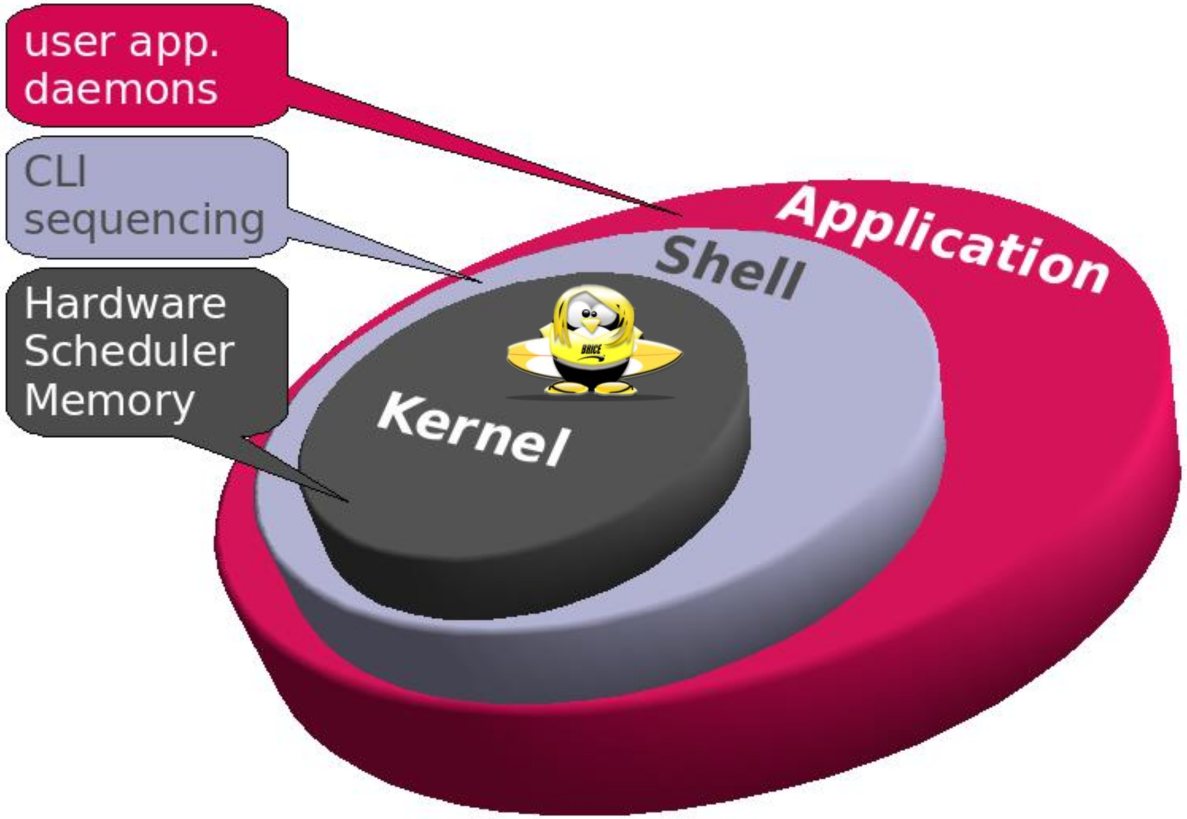
- The Linux® kernel is the main component of a Linux operating system (OS) and is the core interface between a computer's hardware and its processes. It communicates between the 2, managing resources as efficiently as possible.

- The kernel is so named because—like a seed inside a hard shell—it exists within the OS and controls all the major functions of the hardware, whether it's a phone, laptop, server, or any other kind of computer

DISTRIBUTIONS AND REAL WORLD

SYSTEM VIEW

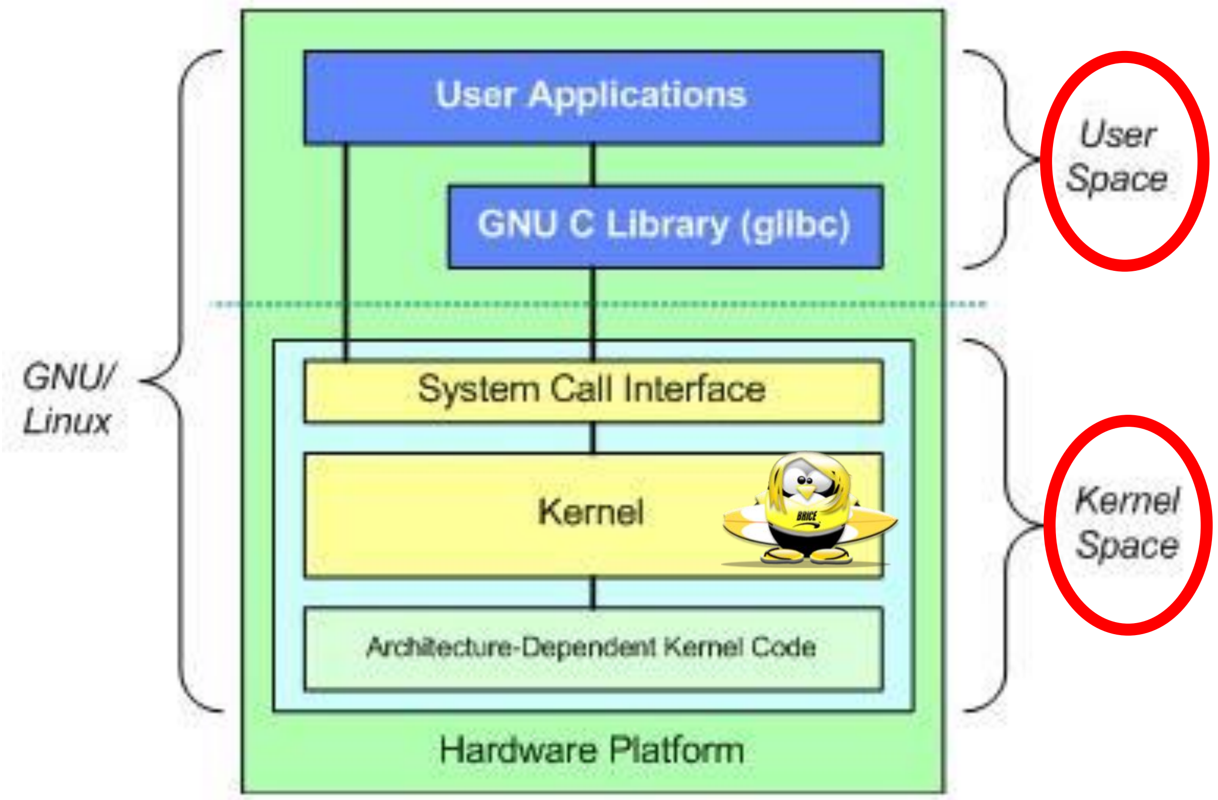

SOFTWARE
Modules
KERNEL
BRICE
HARDWARE

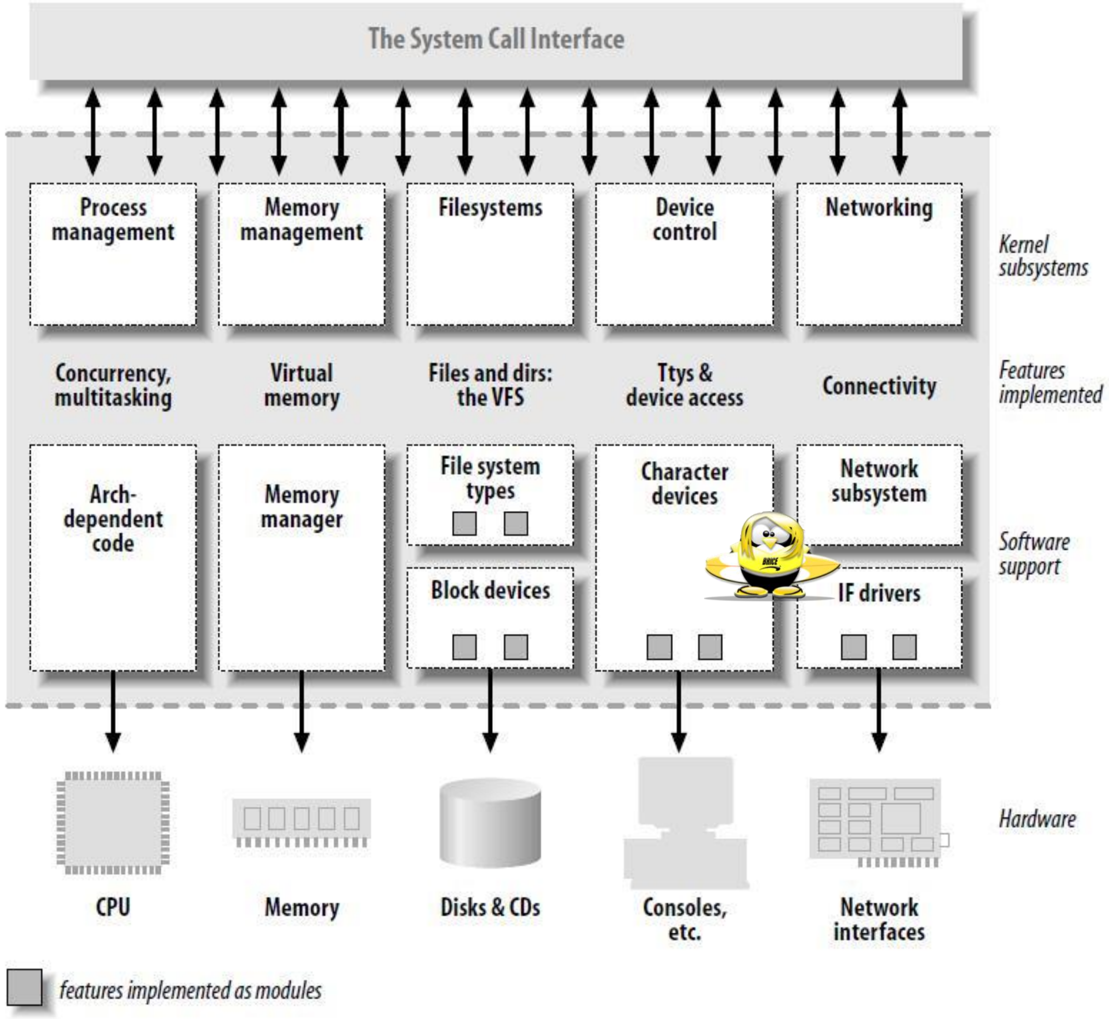

user app.
daemons
CLI
sequencing
Hardware
Scheduler
Memory
Application
Shell
Kernel
BRICE

THE KERNEL RULES

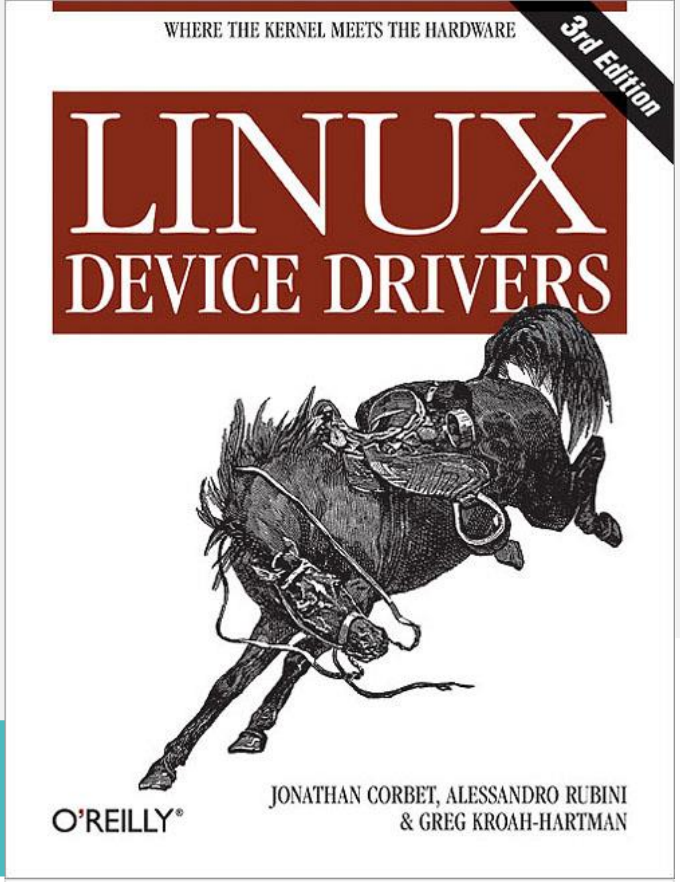
- Programming a KLM is more complex than developing in user land.
  - " Débogage dans l'espace noyau avec KGDB " magazine LM 88
- The whole system relies on the Kernel. Bad programming might impact some critical items and hang the system.
- The glibc library does not exist in the kernel, but some functions are implemented in the lib directory of the kernel sources.
- A KLM is programmed in C, but the KLM structure is object oriented.
- Coding style is written in the Documentation/CodingStyle directory of the Kernel sources.
- A KLM shall support all architectures, namely regarding the endian-ness.

ARCHITECTURE OF THE GNU/LINUX OPERATING SYSTEM





LINUX DEVICE DRIVER



- Introduction

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.

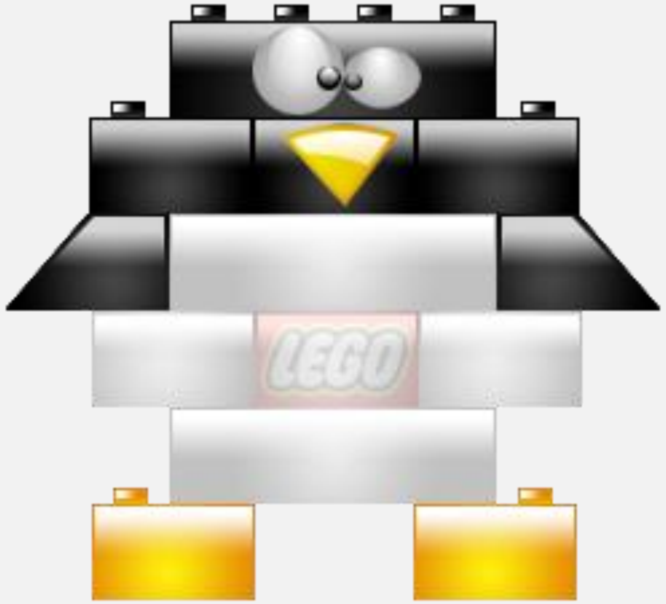
VERSION NUMBERING

- 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/*](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("GPLV2");

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'
% sudo insmod ./hello.ko
% sudo rmmod hello
% dmesg -T
…
[13-03-2022 6pm11] Hello, world
[13-03-2022 6pm11] Goodbye cruel world
```

COMPILATION

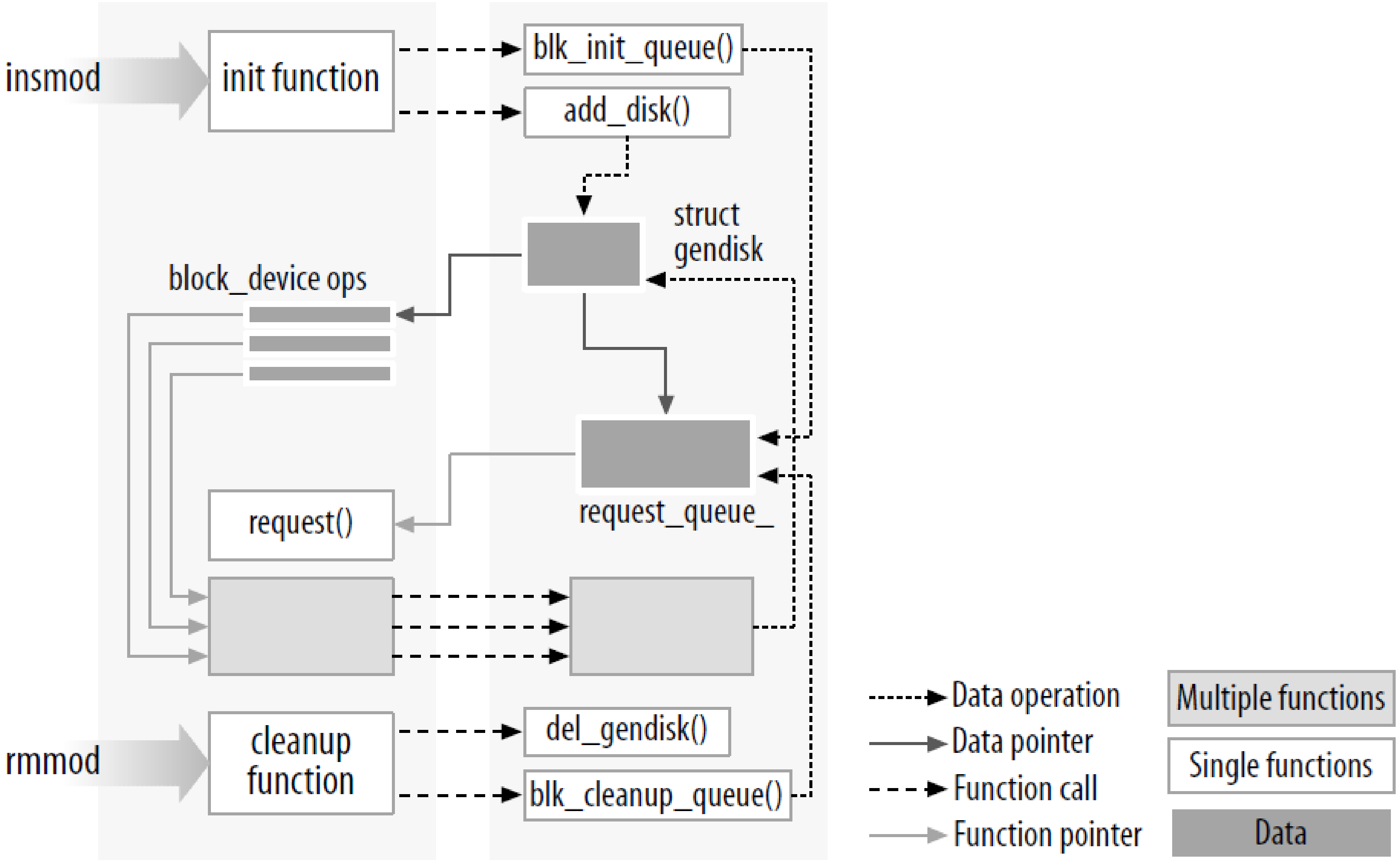
- Gcc from project GNU
- See *Documentation/kbuild* directory in the kernel sources

```
% cat makefile
obj-m := myKLM.o
myKLM-objs := mySourceFile1.o mySourceFile2.o
all:
	make -C /lib/modules/3.5.0-17-generic/build  M=`pwd` modules
clean:
	make -C /lib/modules/3.5.0-17-generic/build  M=`pwd` clean

% gcc -C /lib/modules/5.11.0-37-generic/build  M=`pwd` modules
```

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 allocated dynamically at call time

Double underscore

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

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>*

ERROR HANDLING DURING INITIALIZATION

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

        /* registration takes a pointer and a name */
        err = registerSomeKernelObjectX(ptr1, …);
        if (err) goto fail_this;
        err = registerSomeKernelObjectY(ptr2, …);
        if (err) goto fail_that;

        return 0; /* success */

fail_that: unregisterSomeKernelObjectX(ptr1, …);
fail_this: return err; /* propagate the error */
}
```

CLEANUP

```
void __exit my_cleanup_function(void)
{
        unregisterSomeKernelObjectZ(ptr3, "skull");
        unregisterSomeKernelObjectY(ptr2, "skull");
        unregisterSomeKernelObjectX(ptr1, "skull");
        return;
}
```

MODULE PARAMETERS

- Values supplied during the module initialization

```
% insmod myModule fruit="banana" quantity=10
```

```
static char *param_fruit = "orange";
static int param_quantity = 1;
module_param_named(fruit, param_fruit, char*, S_IRUGO);
MODULE_PARM_DESC(fruit, "Healthy desert");
module_param_named(quantity, param_quantity, int, S_IRUGO);
MODULE_PARM_DESC(quantity, "Quantity of fruit");
```

- Values supplied as a comma-separated list

CHAR DEVICE

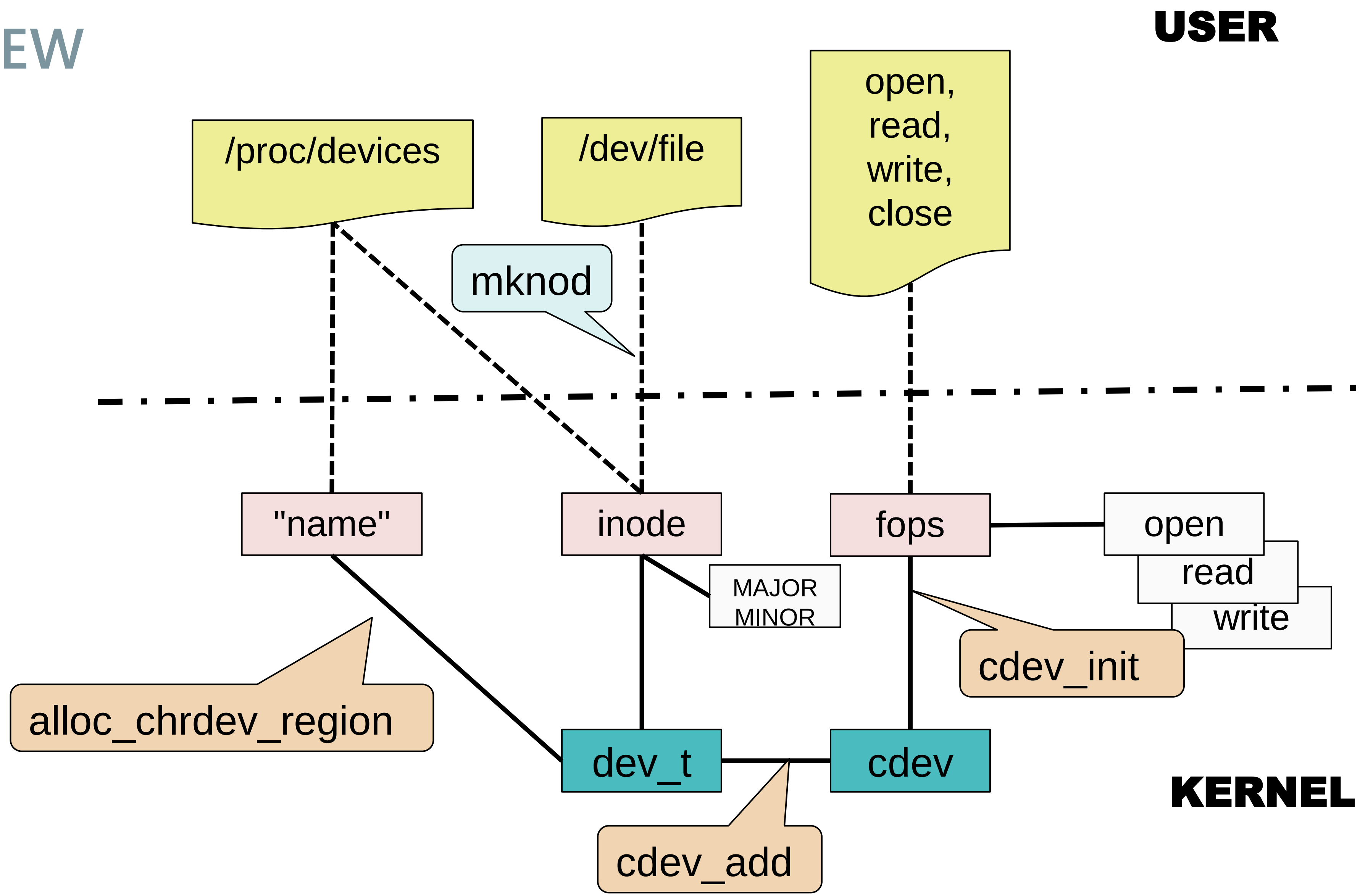
- suitable for most simple hardware devices
- easier to understand than block or network drivers
- aim is to write a *modularized* char driver

ACCESSING CHAR DRIVERS FROM USER LAND

```
>ll -s /dev
brw-rw---- 1 root disk 7, 6 2008-09-11 21:10 loop6
brw-rw---- 1 root disk 7, 7 2008-09-11 21:10 loop7
crw-rw---- 1 root lp 6, 0 2008-09-11 21:10 lp0
crw-r----- 1 root kmem 1, 1 2008-09-11 21:08 mem
crw-rw-rw- 1 root root 1, 3 2008-07-21 13:13 null
```

- “c” - Character mode driver
- Permissions settings
- Owner and Group
- MAJOR and MINOR device number
- The scheme for the numbers can be seen in ***/proc/devices***

OVERVIEW



USER
/proc/devices
/dev/file
open,
read,
write,
close
mknod
"name"
inode
fops
open
read
write
MAJOR
MINOR
cdev_init
alloc_chrdev_region
dev_t
cdev
KERNEL
cdev_add

INSTALLING A DEVICE NODE

- Two parts operation:

1- LKM must register itself to have a specific major and minor device number pair

```
/* statically */
int register_chrdev_region(dev_t myDev, unsigned int count, char *name;
/* dynamically */
int alloc_chrdev_region(dev_t *pmyDev, unsigned int firstminor, unsigned
int count, char *name);
/* helper */
MKDEV(major, minor), MAJOR(Device), MINOR(Device)
```

2- A top level administrator or script must create a node that connects the major/minor device number pair to a file system object within ***/dev***

- Linux provides utility for “system admin” or “system start-up” to create “nodes” with the file system

```
> mknod /dev/devicename c MAJOR MINOR
```

REMOVING A DEVICE NODE

```
unregister_chrdev_region(dev_t myDev, unsigned int count);
```

Caution

- Return 0 for success; negative is error
- Generally, bail out of module on error
- Clean all resources already allocated before leaving
- Registering and unregistering ONE device per module

ADMINISTRATION – INSTALLATION SCRIPT

```
module="EnsiCaen_ldd"
device="EnsiCaen_device"

# install the LKM and exit if insmod fails with an error
sudo insmod $module.ko verbose=1

# query the /proc/devices file
major=$(awk "\$2==\"$module\" {print \$1}" /proc/devices)
minor=0

# create the new file system node
sudo mknod /dev/$device c $major $minor

# ensure device file is readable by all
sudo chmod 644 /dev/$device
```

ADMINISTRATION – CLEAN UP SCRIPT

```
module="EnsiCaen_ldd"
device="EnsiCaen_device“

sudo rmmod $module
sudo rm -f /dev/$device
```

FILE OPERATIONS

- Drivers for most operating systems will require the implementation of a table of entry points
- Linux uses a special kernel structure, called ***struct file_operations, to supply these entry points***
- Structure is defined within the ***linux/fs.h header file***

```
static struct file_operations my_module_fops = {
        owner: THIS_MODULE,
        open: my_module_open,
        read: my_module_read,
        release: my_module_release
};
```

KERNEL STRUCTURES TO SUPPORT FILE OPERATIONS

- The file operations structure must be registered with the Linux operating system using a `struct cdev`

Init method

```
// init & register character driver's file operations
cdev_init (&myCDev, &myFops);
my_module_cdev.owner = THIS_MODULE;
my_module_cdev.ops = &myFops;
rc = cdev_add (&myCDev, myDev, 1);
if (rc) {
        printk(KERN_INFO "my_module: unable to add
        cdev struct.\n");
        return rc;
} /* endif */
```

Exit method

```
cdev_del (&myCDev);
unregister_chrdev_region (myDev, 1);
```

FILE OPERATION – EXAMPLE

```
int my_module_open (struct inode *pInode, struct file *fp)
{
        if (my_module_is_open) {
                return -EBUSY;
        } /* endif */
        my_module_is_open++;
…
```

```
int my_module_release (struct inode *pInode, struct file *fp)
{
        // indicate that future calls to open() will succeed
         my_module_is_open --;
        printk (KERN_INFO “my_module: my_module_release ... \n");
} /* end my_module _release */
```

DRIVER USAGE IN USERSPACE

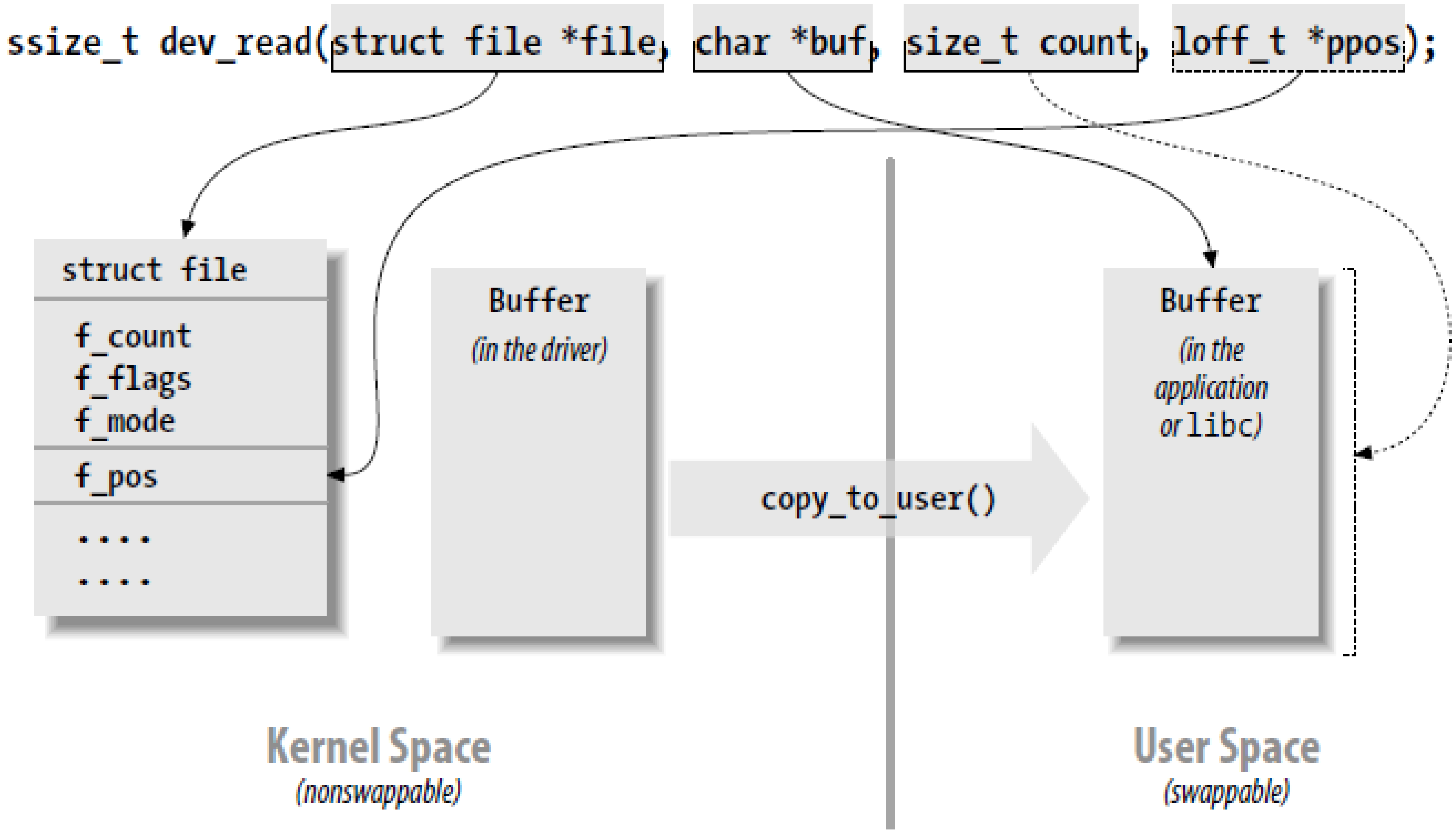
- Making it accessible to userspace application by creating a device node:
  mknod /dev/demo c 202 128

- Using normal the normal le API :

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

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

THE ARGUMENTS TO READ



FILE OPERATION – EXAMPLE

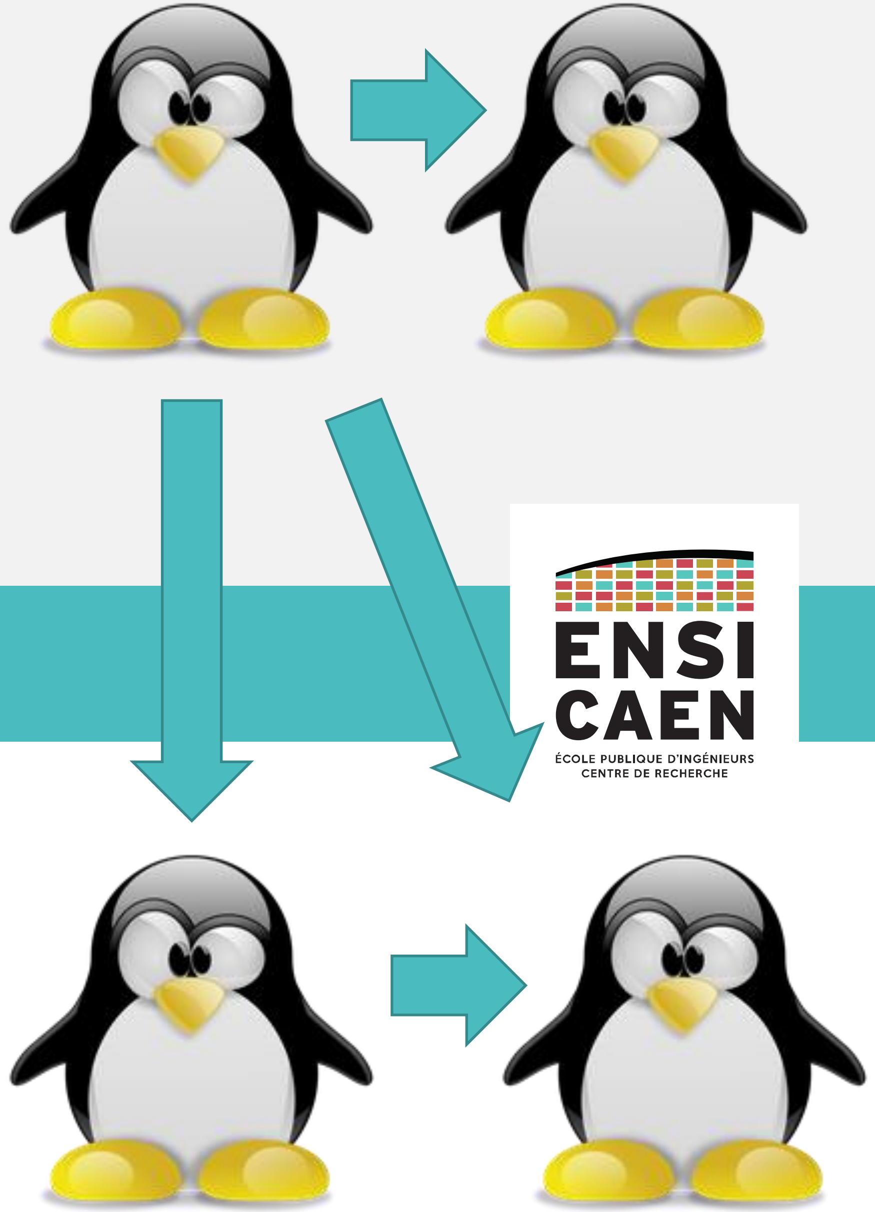
```
unsigned long copy_to_user(void __user *to, const void *from,
unsigned long n);
unsigned long copy_from_user(void *to, const void __user *from,
unsigned long n);
put_user (variable, ptr);
get_user (variable, ptr);
```

```
int my_module_read (struct file *fp, char __user *buffer,
                    size_t len, loff_t *offset)
{
        available = LOCAL_BUF_SIZE - *offset;
        if (len > available) len = available;
        copy_to_user (buffer, LOCAL_BUF_ADD + *offset, len);
        *offset += len;
        return len;
}
```

Size of some driver internal buffer

@ of some driver internal buffer

KERNEL FRAMEWORK



Factorization

Coherent interface

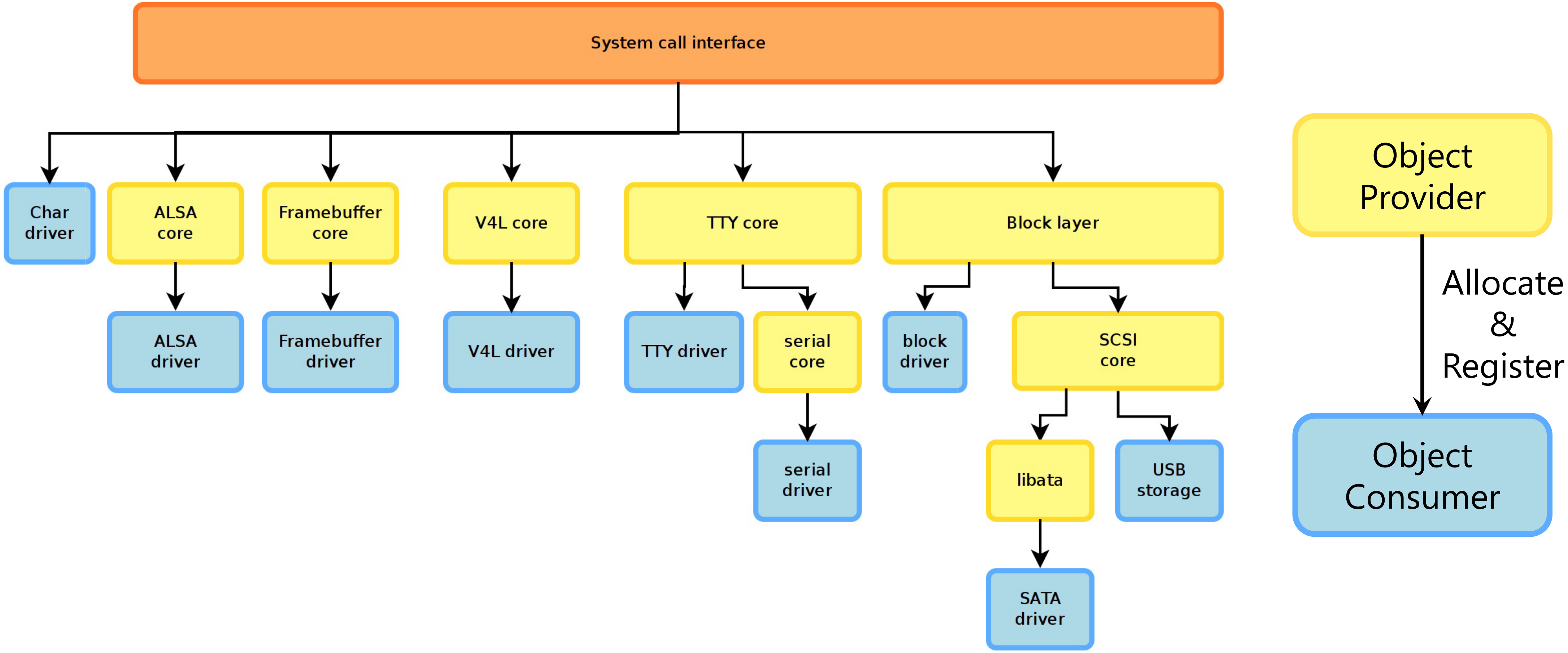
Efficiency

Specialization

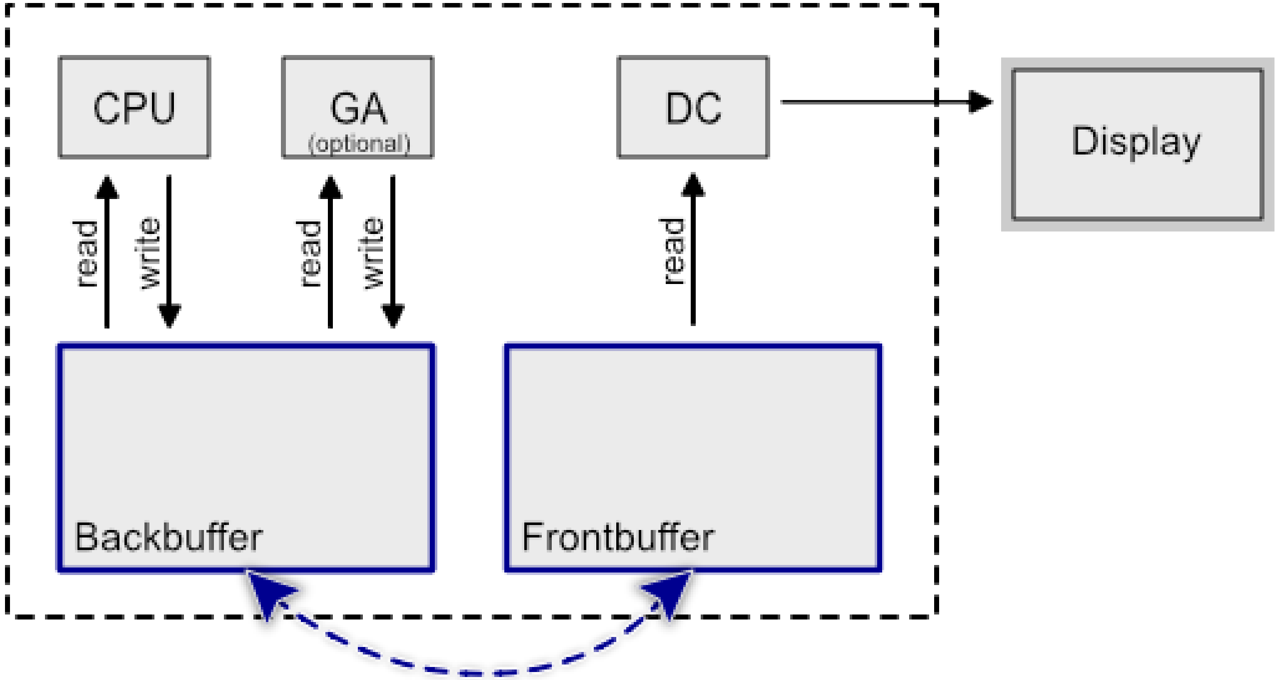
FRAMEWORK AND DRIVERS

- Most device drivers are not directly implemented as character devices or block devices
- They are implemented under a framework, specic to a device type (framebuer, V4L, serial, etc.)
  - The framework allows to factorize the common parts of drivers for the same type of devices
  - From userspace, they are still seen as normal character devices
  - The framework allows to provide a coherent userspace interface (ioctl numbering and semantic, etc.) for every type of device, regardless of the driver

EXAMPLE OF FRAMEWORK



EXAMPLE OF THE FRAMEBUFFER



- Kernel option CONFIG FB
- Implemented in drivers/video/
  - fb.c, fbmem.c, fbmon.c, fbcmap.c, fbsysfs.c, modedb.c, fbcvt.c
- Implements a single character driver (through file operations), registers the major number and allocates minors, defines and implements the user/kernel API
  - First part of include/linux/fb.h
- Defines the set of operations a framebuffer driver must implement and helper functions for the drivers
  - struct fb ops
  - Second part of include/linux/fb.h

FRAMEBUFFER SKELETON EXAMPLE

```
static int xxx_open(struct fb_info *info, int user) {}
static int xxx_release(struct fb_info *info, int user) {}
static int xxx_check_var(struct fb_var_screeninfo *var, struct fb_info *info) {}
static int xxx_set_par(struct fb_info *info) {}

static struct fb_ops xxx_ops = {
        .owner          = THIS_MODULE,
        .fb_open        = xxxfb_open,
        .fb_release     = xxxfb_release,
        .fb_check_var   = xxxfb_check_var,
        .fb_set_par     = xxxfb_set_par,
        [...]
};

init()
{
    struct fb_info *info;
    info = framebuffer_alloc(sizeof(struct xxx_par), device);
    info->fbops = &xxxfb_ops;
    [...]
    register_framebuffer(info);
}
```

FOPS

Alloc

Register

DEVICE AND DRIVER MODEL

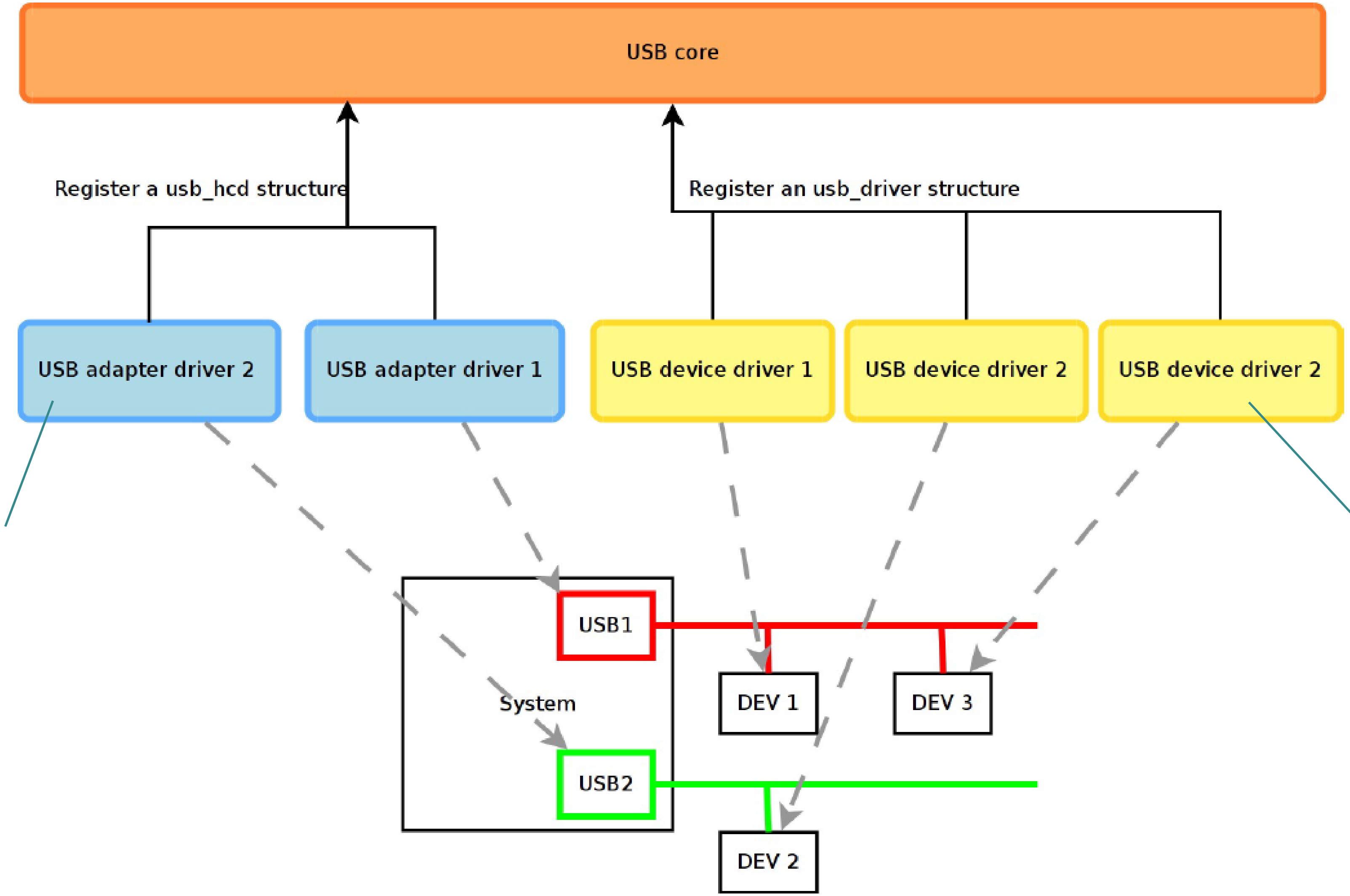
Kernel 2.6 model

- One of the features that came with the 2.6 kernel is a unified device and driver model
- Instead of different ad-hoc mechanisms in each subsystem, the device model unifies the vision of the devices, drivers, their organization and relationships
- Allows to minimize code duplication, provide common facilities, more coherency in the code organization
- Base structure types: `struct device, struct driver, struct bus_type`
- Is visible in userspace through the `sysfs` filesystem, traditionally mounted under `/sys`

BUS DRIVER

- Core element of the device model
- A single bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.
- This driver is responsible for:
  - Registering the bus type (bus type structure)
  - Allow the registration of adapter/interface drivers (USB controllers, I2C controllers, SPI controllers). These are the hardware devices capable of detecting and providing access to the devices connected to the bus
  - Allow the registration of device drivers (USB devices, I2C devices, SPI devices). These are the hardware devices connected to the different buses.
  - Matching the device drivers against the detected devices

ADAPTER, BUS AND DEVICE DRIVERS

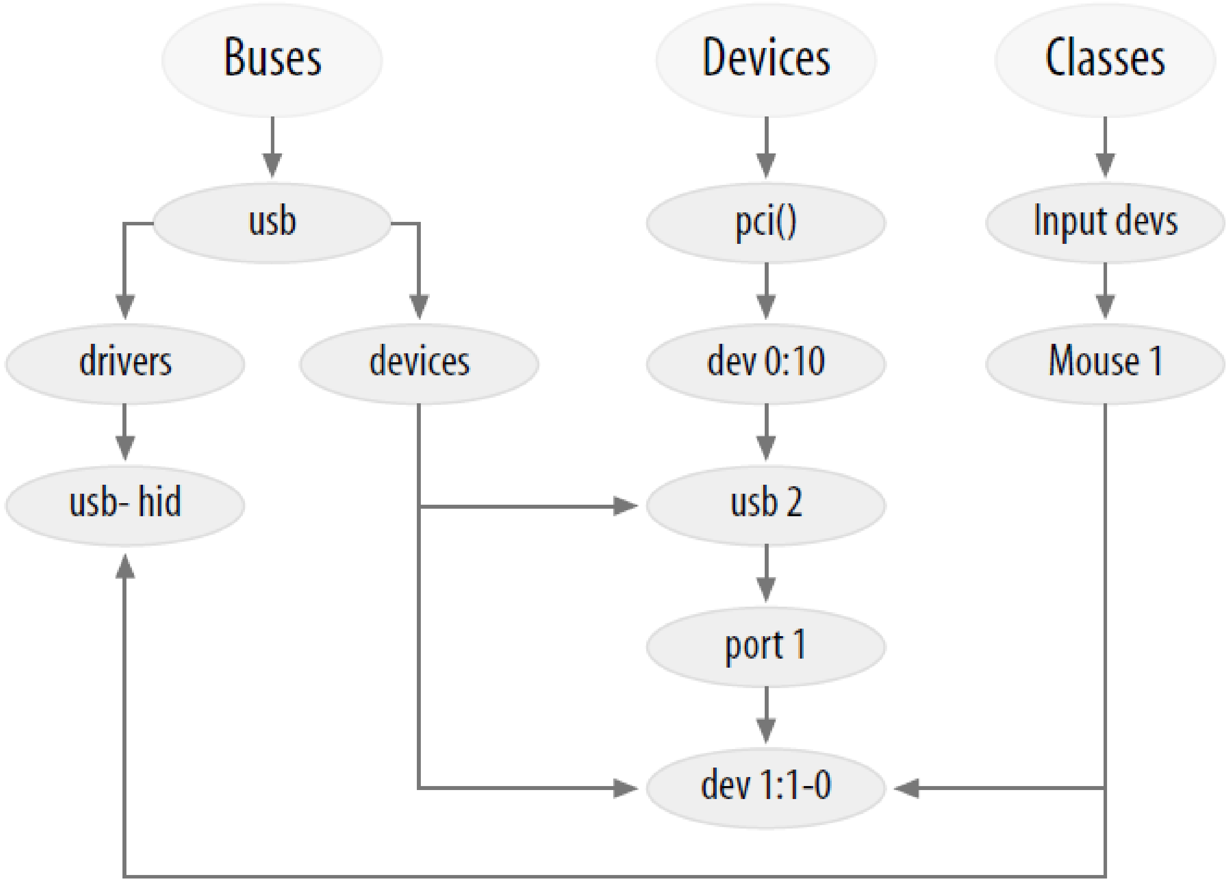


Detection & Access

Registration of HW devices

BUSES, DEVICES, AND DRIVERS

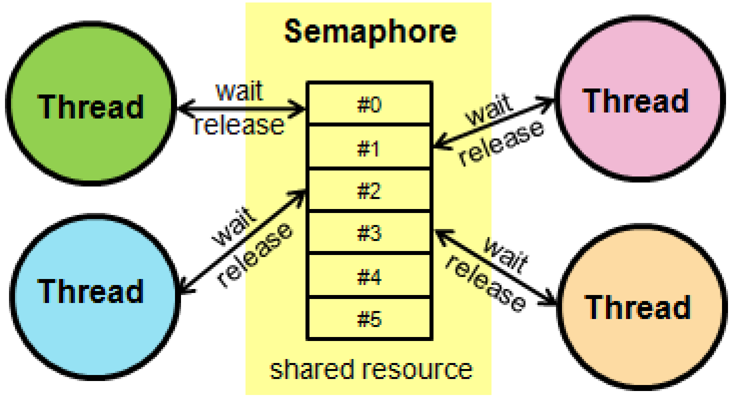
- The core "devices" tree shows how the mouse is connected to the system
- The "bus" tree tracks what is connected to each bus
- The under "classes" concerns itself with the functions provided by the devices, regardless of how they are connected.



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

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);
```

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);
/* for long read period */
void downgrade_write(struct rw_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);

/* For ISR: disable interrupt, the interrupt state is stored in flags */
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
/* enable back the IRQ when the spin is released /!\ if allowed */
void spin_lock_irq(spinlock_t *lock);

/* For tasklet: 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

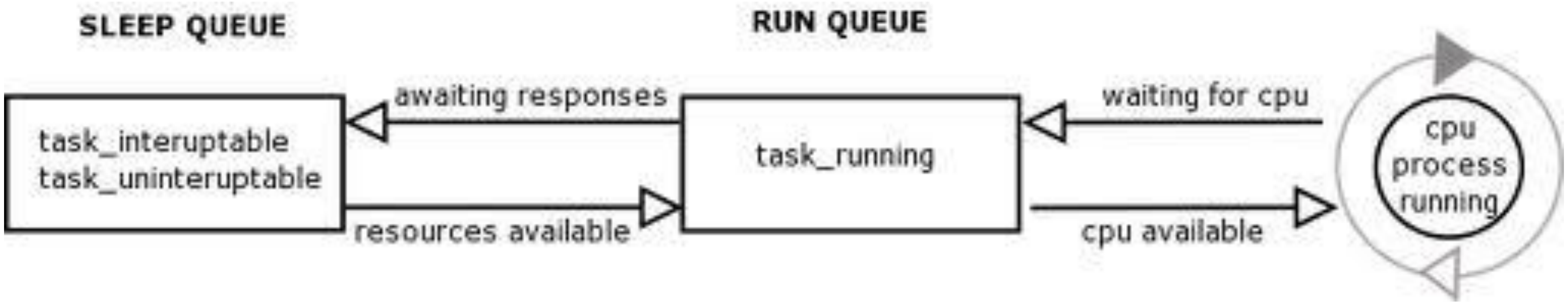
```
void set/clear/change_bit(nr, void *addr);
test_bit(nr, void *addr);
int test_and_set/clear/change_bit(nr, void *addr);
```

- CPU optimized with assembly implementation

ADVANCED CHAR DRIVER OPERATIONS

Sleeping & wait queue

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.

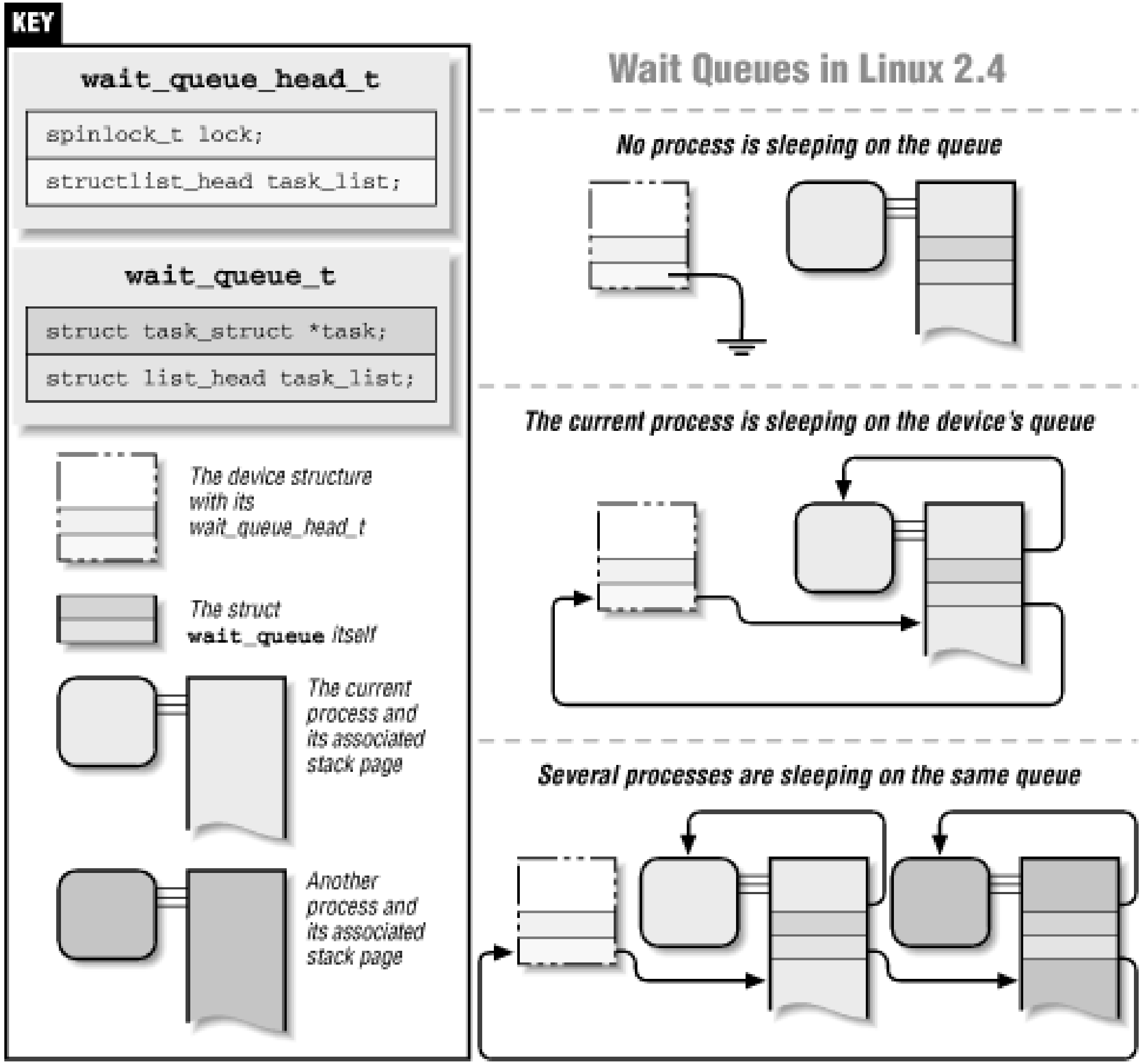
Statical declaration:

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

Dynamic declaration:

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

WAIT 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 */
}
```

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);

tasklet_schedule(name);
```

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

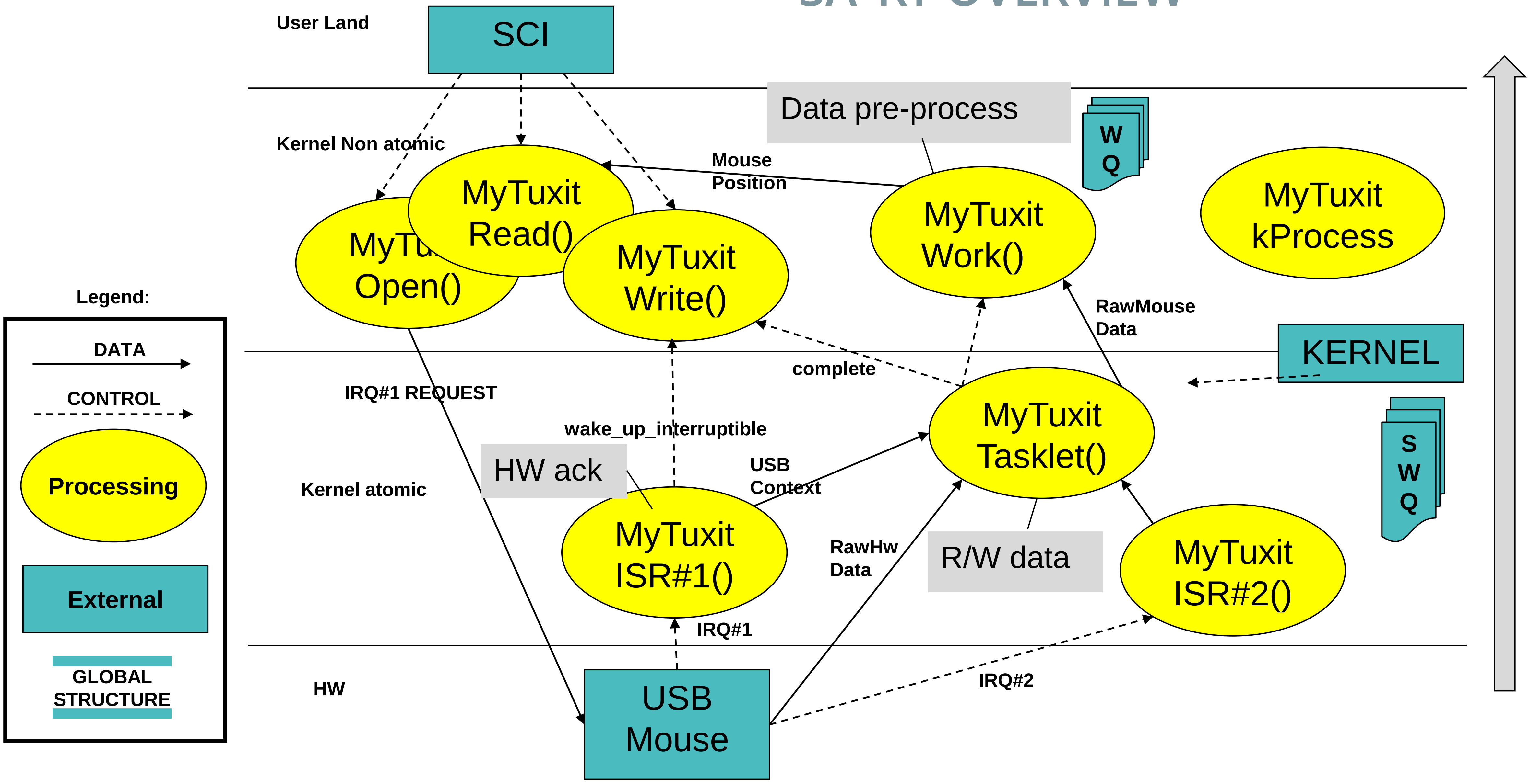
- If you only submit tasks to the queue occasionally, it may be more efficient to simply use the shared, default workqueue that is provided by the kernel.

```
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);
```

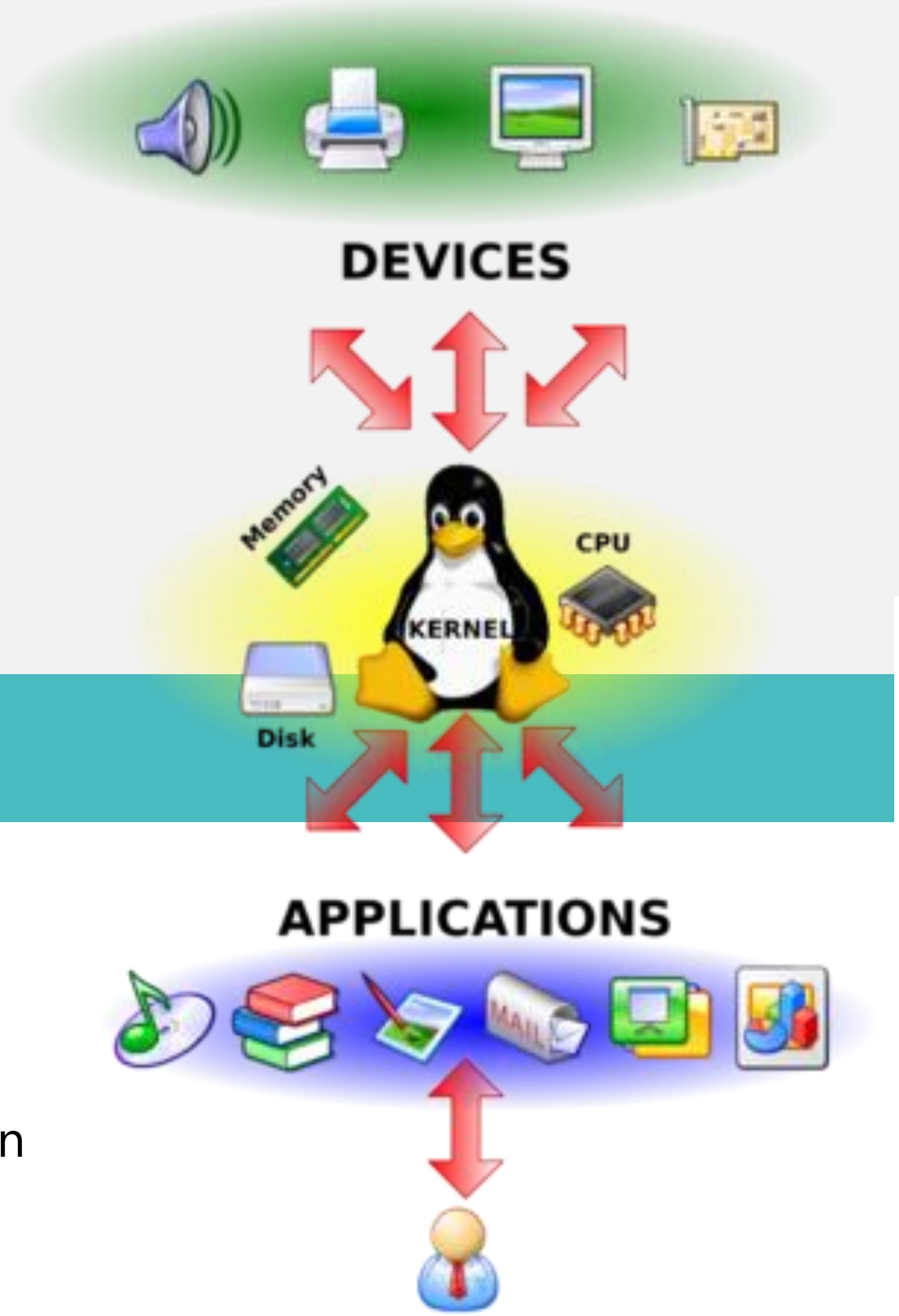
RT LAYER OVERVIEW

Using SA-RT method

SA-RT OVERVIEW



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

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
```

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;
}
```

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;
}
```

TASKLET: BOTTOM HALF

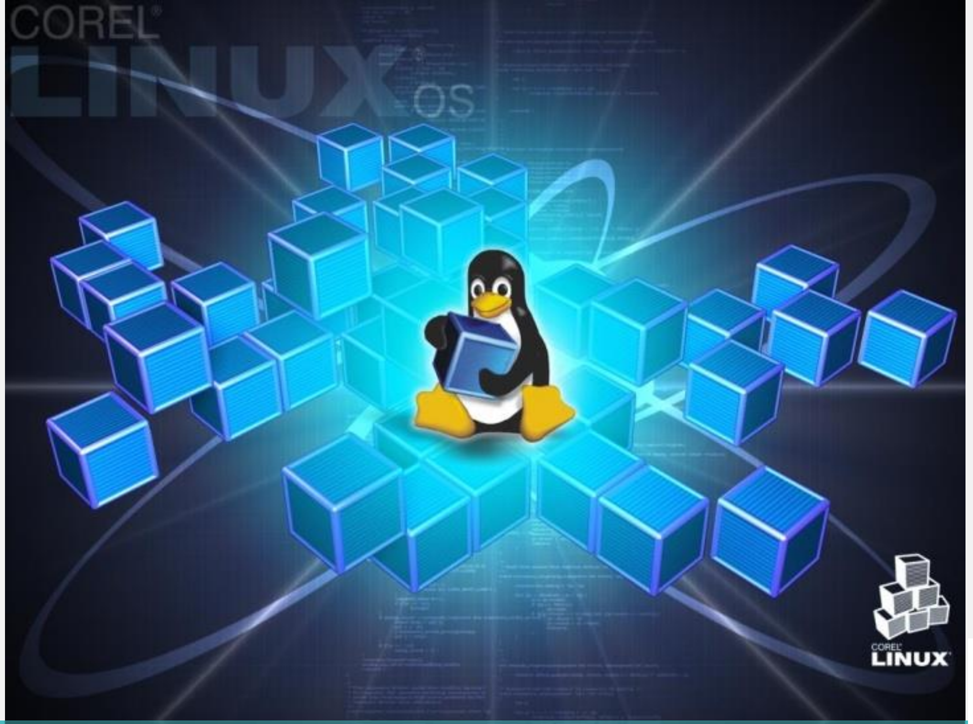
```
void short_do_tasklet (unsigned long unused)
{
  /* awake any reading process */
  wake_up_interruptible(&short_queue);
}
```

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;
}
```

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…)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

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

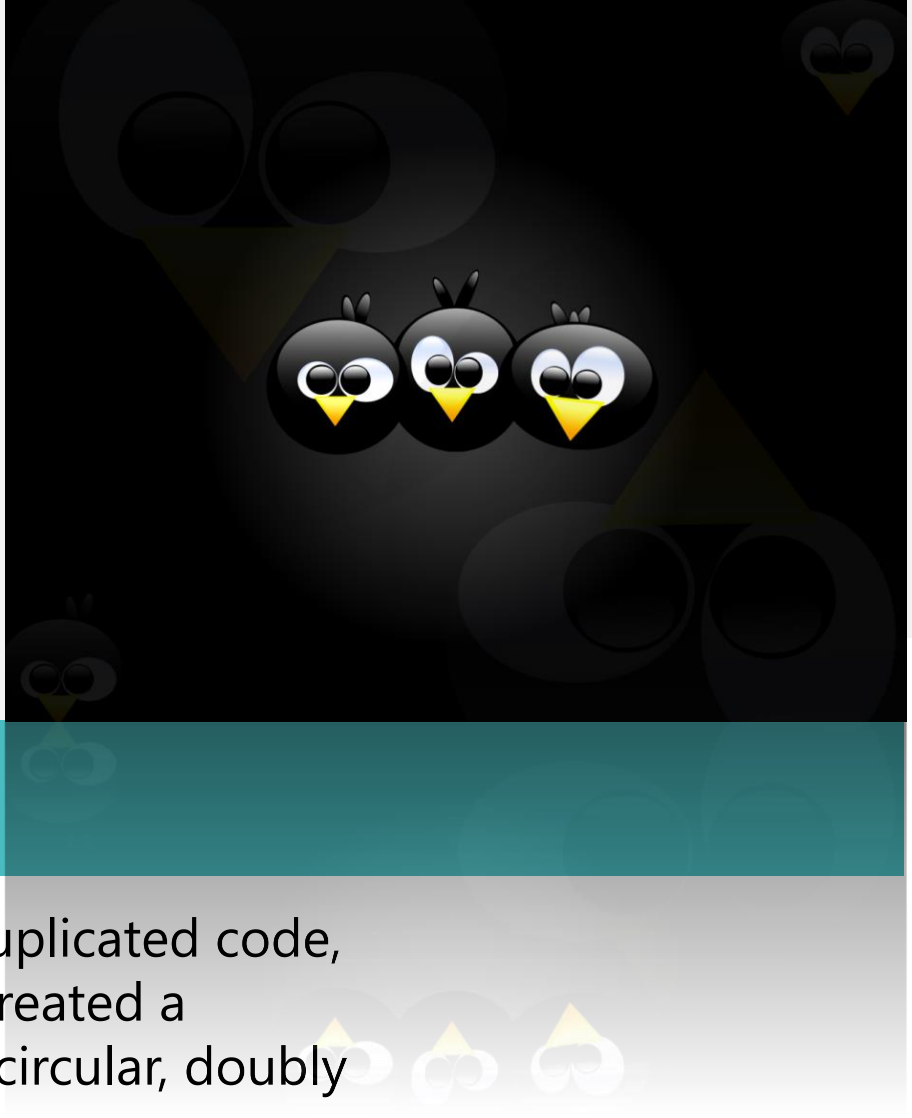
- addr should be an address obtained from *ioremap* (perhaps with an integer offset); the return value is what was read from the given I/O memory.

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

- 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);
```

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

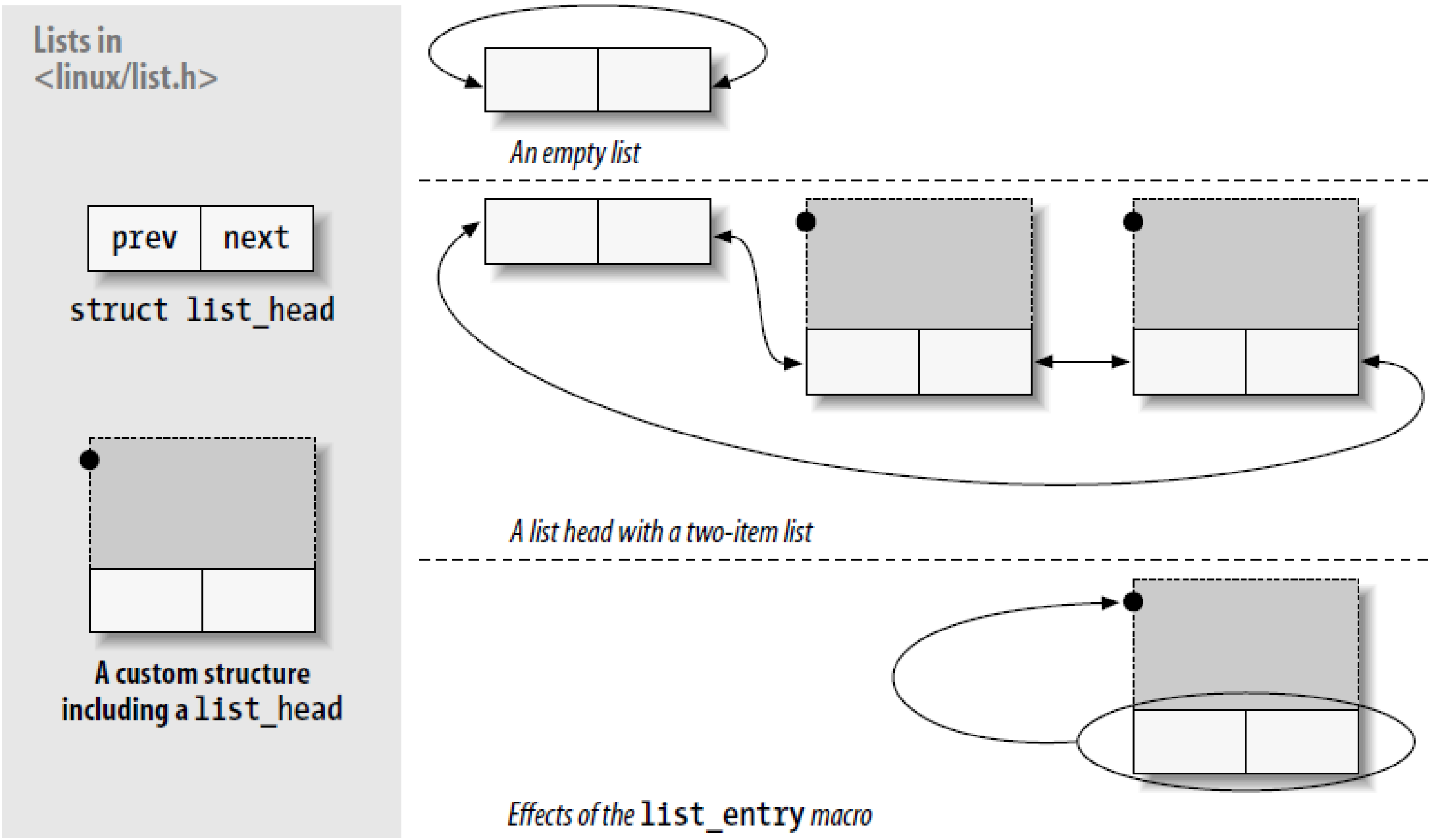
LIST HEAD

```
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



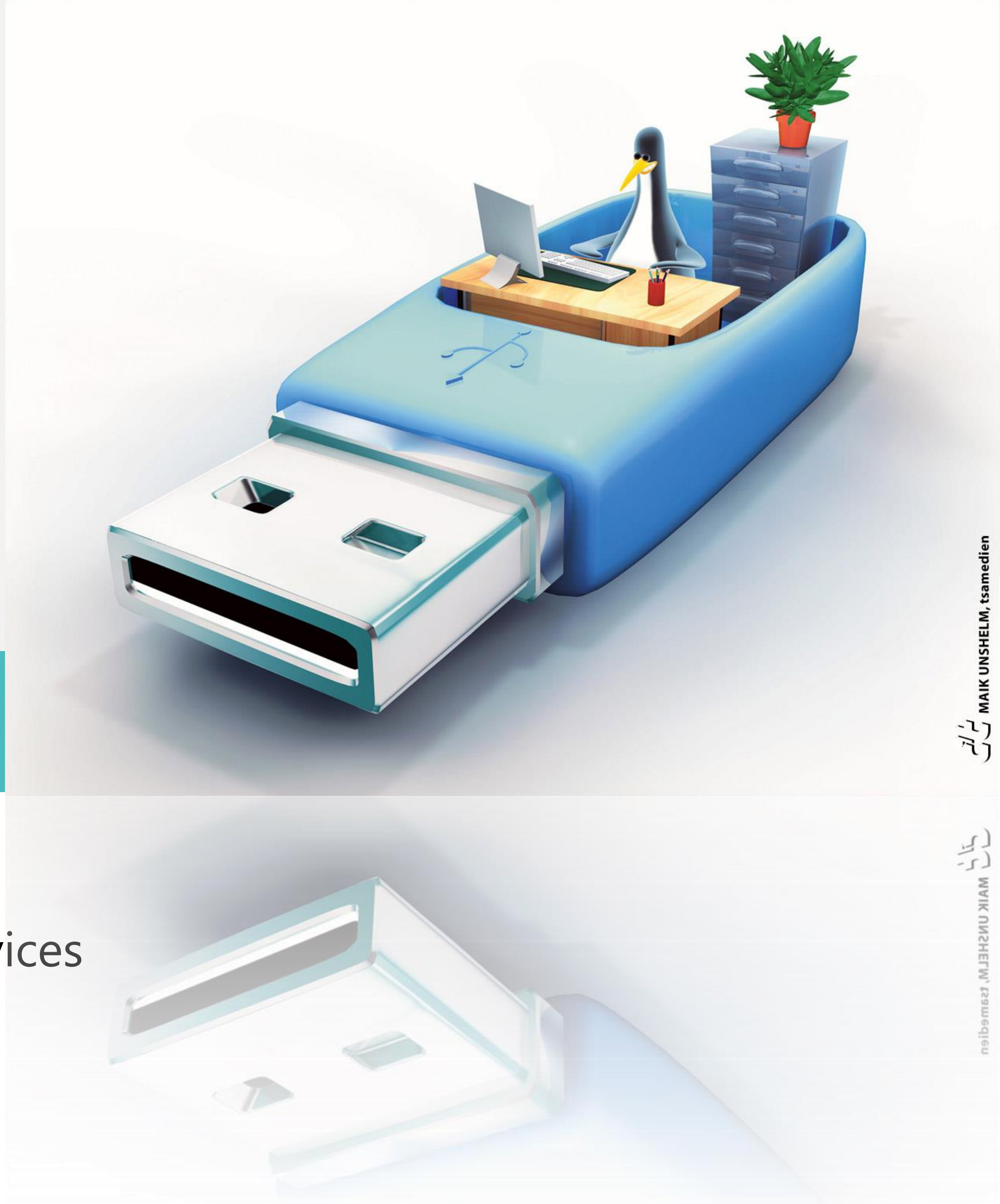
LIST MAKING

```
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);
```

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);
```

THE LINUX DEVICE MODEL



- Device classes
- Hot-pluggable devices
- Object lifecycles

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);`

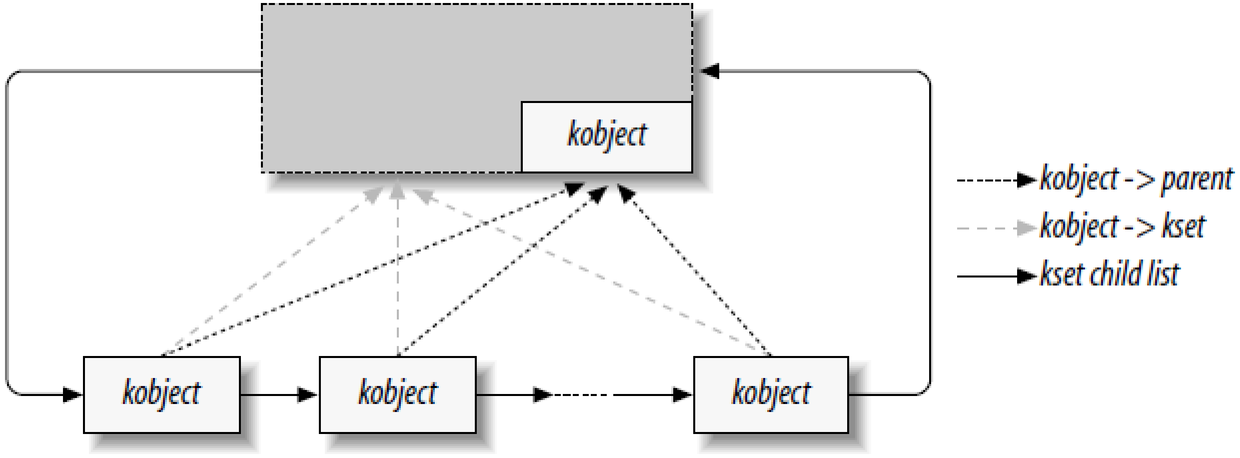
KOBJECT HANDLING

```
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.

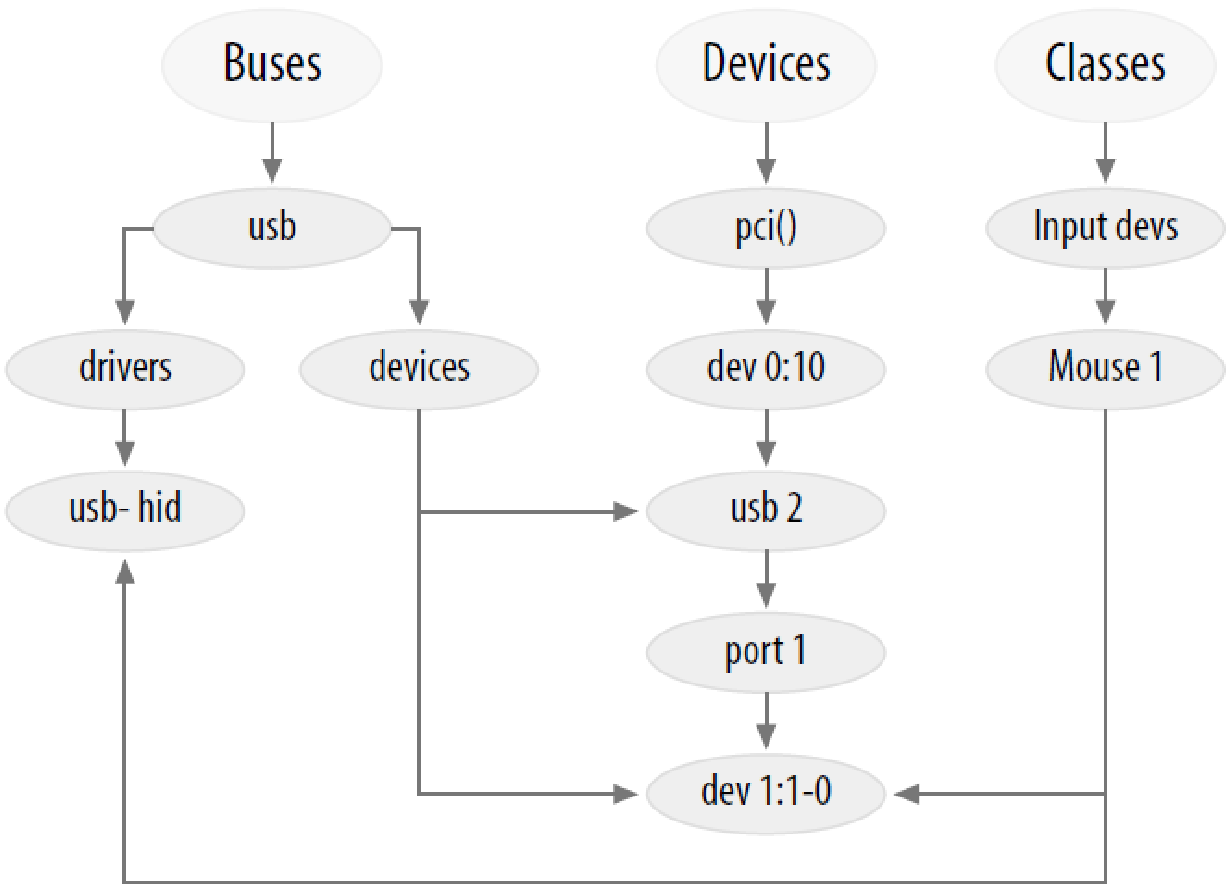
```
int (*hotplug)(struct kset *kset, struct kobject *kobj,
               char **envp, int num_envp, char *buffer,
               int buffer_size);
```

BUSES, DEVICES, AND DRIVERS ANNEX

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

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.



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 identiers 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,
        },
};
```

MERCI

André Lépine

Enseignement Linux Embarqué

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

L'École des INGÉNIEURS Scientifiques