Embedded Linux kernel and driver development

Sebastien Jan
Michael Opdenacker
Thomas Petazzoni
Free Electrons

Free Electrons

Embedded Linux Developers

Latest update: Mar 2, 2011,
Document sources, updates and translations:
http://free-electrons.com/docs/kernel
Corrections, suggestions, contributions and translations are welcome!

Contents

Driver development

- Loadable kernel modules
- Memory management
- I/O memory and ports
- Character drivers
- Processes and scheduling
- Sleeping, Interrupt management
- Handling concurrency
- Debugging
- mmap
- Device and driver model

Driver development

Loadable kernel modules

hello module

```
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
    printk(KERN_ALERT "Good morrow");
    printk(KERN_ALERT "to this fair assembly.\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_ALERT "Alas, poor world, what treasure");
    printk(KERN_ALERT "hast thou lost!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR("William Shakespeare");
```

`__init`:
removed after initialization (static kernel or module).

`__exit`: discarded when module compiled statically into the kernel.

Example available on http://free-electrons.com/doc/c/hello.c

Hello module explanations

- Headers specific to the Linux kernel: `<linux/xxx.h>`
  - No access to the usual C library, we're doing kernel programming
- An initialization function
  - Called when the module is loaded, returns an error code (0 on success, negative value on failure)
  - Declared by the `module_init()` macro: the name of the function doesn't matter, even though `modulename_init()` is a convention.
- A cleanup function
  - Called when the module is unloaded
  - Declared by the `module_exit()` macro.
- Metadata informations declared using `MODULE_LICENSE()`, `MODULE_DESCRIPTION()` and `MODULE_AUTHOR()`

Symbols exported to modules

- From a kernel module,
only a limited number of kernel functions can be called
- Functions and variables have to be explicitly *exported* by the kernel to be visible from a kernel module
- Two macros are used in the kernel to export functions and variables:
  - `EXPORT_SYMBOL(symbolname)`, which exports a function or variable to all modules
  - `EXPORT_SYMBOL_GPL(symbolname)`, which exports a function or variable only to GPL modules
- A normal driver should not need any non-exported function.

Symbols exported to modules (2)

kernel

```
void func1() { … }

void func2() { … }
EXPORT_SYMBOL(func2);

void func3() { … }
EXPORT_SYMBOL_GPL(func3);
```

func1();	OK
func2();	OK
func3();	OK
func4();	NOK

GPL module A

```
void func4() { … }
EXPORT_SYMBOL_GPL(func4);
```

func1();	NOK
func2();	OK
func3();	OK
func4();	OK

non-GPL module B

func1();	NOK
func2();	OK
func3();	NOK
func4();	NOK

GPL module C

func1();	NOK
func2();	OK
func3();	OK
func4();	OK

Module license

- Several usages
  - Used to restrict the kernel functions that the module can use if it isn't a GPL-licensed module
    - Difference between `EXPORT_SYMBOL()` and `EXPORT_SYMBOL_GPL()`
  - Used by kernel developers to identify issues coming from proprietary drivers, which they can't do anything about (“Tainted” kernel notice in kernel crashes and oopses).
  - Useful for users to check that their system is 100% free (check `/proc/sys/kernel/tainted`)
- Values
  - GPL, GPL v2, GPL and additional rights, Dual MIT/GPL, Dual BSD/GPL, Dual MPL/GPL, Proprietary

Compiling a module

- Two solutions
  - « Out of tree »
    - When the code is outside of the kernel source tree, in a different directory
    - Advantage: Might be easier to handle than modifications to the kernel itself
    - Drawbacks: Not integrated to the kernel configuration/compilation process, needs to be built separately, the driver cannot be built statically
  - Inside the kernel tree
    - Well integrated into the kernel configuration/compilation process
    - Driver can be built statically if needed

Compiling a out-of-tree module

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

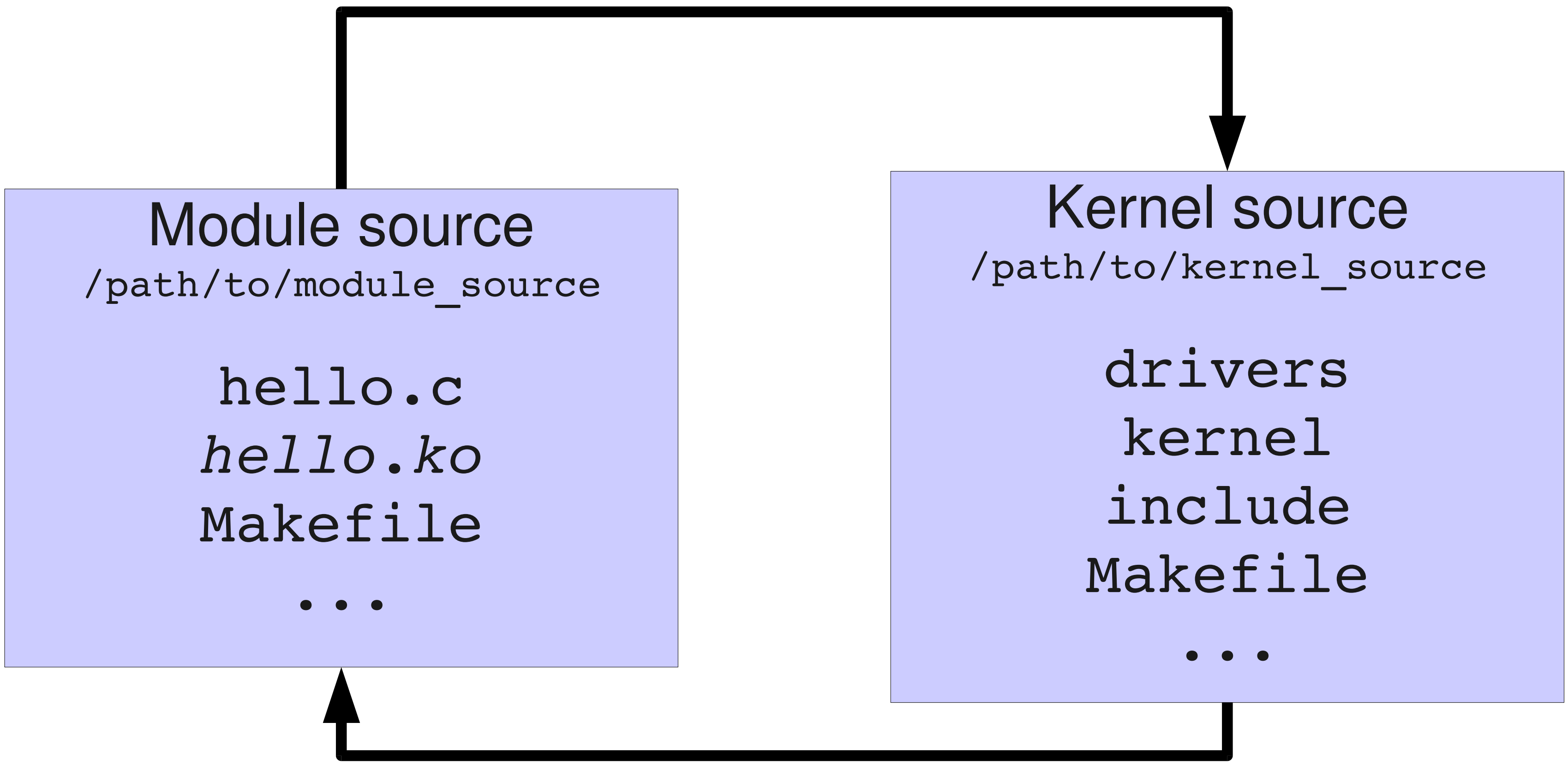
```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources
all:
	$(MAKE) -C $(KDIR) M=`pwd` modules
endif
```

`[Tab]`!
(no spaces)

Either
- full kernel source directory (configured and compiled)
- or just kernel headers directory (minimum needed)

Compiling an out-of-tree module (2)

Step 1: the module `Makefile` is interpreted with KERNELRELEASE undefined, so it calls the kernel `Makefile`, passing the module directory in the `M` variable



Step 2: the kernel `Makefile` knows how to compile a module, and thanks to the `M` variable, knows where the `Makefile` for our module is. The module `Makefile` is interpreted with `KERNELRELEASE` defined, so the kernel sees the `obj-m` definition.

Modules and kernel version

- To be compiled, a kernel module needs access to the kernel headers, containing the functions, types and constants definitions
- Two solutions
  - Full kernel sources
  - Only kernel headers (`linux-headers-*` packages in Debian/Ubuntu distributions)
- The sources or headers must be configured
  - Many macros or functions depend on the configuration
- A kernel module compiled against version X of kernel headers will **not** load in kernel version Y
  - `modprobe/insmod` will say « Invalid module format »

New driver in kernel sources (1)

To add a new driver to the kernel sources:

- Add your new source file to the appropriate source directory.
  Example: `drivers/usb/serial/navman.c`
- Single file drivers in the common case, even if the file is several thousand lines of code. Only really big drivers are split in several files or have their own directory.
- Describe the configuration interface for your new driver
  by adding the following lines to the `Kconfig` file in this directory:

```
config USB_SERIAL_NAVMAN
        tristate "USB Navman GPS device"
        depends on USB_SERIAL
        help
          To compile this driver as a module, choose M here: the
          module will be called navman.
```

New driver in kernel sources (2)

- Add a line in the `Makefile` file based on the `Kconfig` setting:

```
obj-$(CONFIG_USB_SERIAL_NAVMAN) += navman.o
```

It tells the kernel build system to build `navman.c` when the `USB_SERIAL_NAVMAN` option is enabled. It works both if compiled statically or as a module.

- Run `make xconfig` and see your new options!
- Run `make` and your new files are compiled!
- See `Documentation/kbuild/` for details and more elaborate examples like drivers with several source files, or drivers in their own subdirectory, etc.

How to create Linux patches

- The old school way
  - Before making your changes, make sure you have two kernel trees
    `cp -a linux-2.6.37/ linux-2.6.37-patch/`
  - Make your changes in `linux-2.6.37-patch/`
  - Run `make distclean` to keep only source files.
  - Create a patch file:
    ```
    diff -Nur linux-2.6.37/ \
    linux-2.6.37-patch/ > patchfile
    ```
  - Not practical, does not scale to multiple patches

- The new school ways
  - Use `quilt` (tool to manage a stack of patches)
  - Use `git` (revision control system used by the Linux kernel developers)

Thanks to Nicolas Rougier (Copyright 2003, http://webloria.loria.fr/~rougier/) for the Tux image

hello module with parameters

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

MODULE_LICENSE("GPL");

/* A couple of parameters that can be passed in: how many times we say
   hello, and to whom */

static char *whom = "world";
module_param(whom, charp, 0);

static int howmany = 1;
module_param(howmany, int, 0);

static int __init hello_init(void)
{
    int i;
    for (i = 0; i < howmany; i++)
     printk(KERN_ALERT "(%d) Hello, %s\n", i, whom);
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel %s\n", whom);
}

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

Thanks to Jonathan Corbet for the example!

Example available on http://free-electrons.com/doc/c/hello_param.c

Declaring a module parameter

```
#include <linux/moduleparam.h>

module_param(
    name,     /* name of an already defined variable */
    type,     /* either byte, short, ushort, int, uint, long,
                 ulong, charp, or bool.
                 (checked at compile time!) */
    perm      /* for /sys/module/<module_name>/parameters/<param>
                 0: no such module parameter value file */
);
```

Example

```
int irq=5;
module_param(irq, int, S_IRUGO);
```

Modules parameter arrays are also possible with `module_param_array()`, but they are less common.

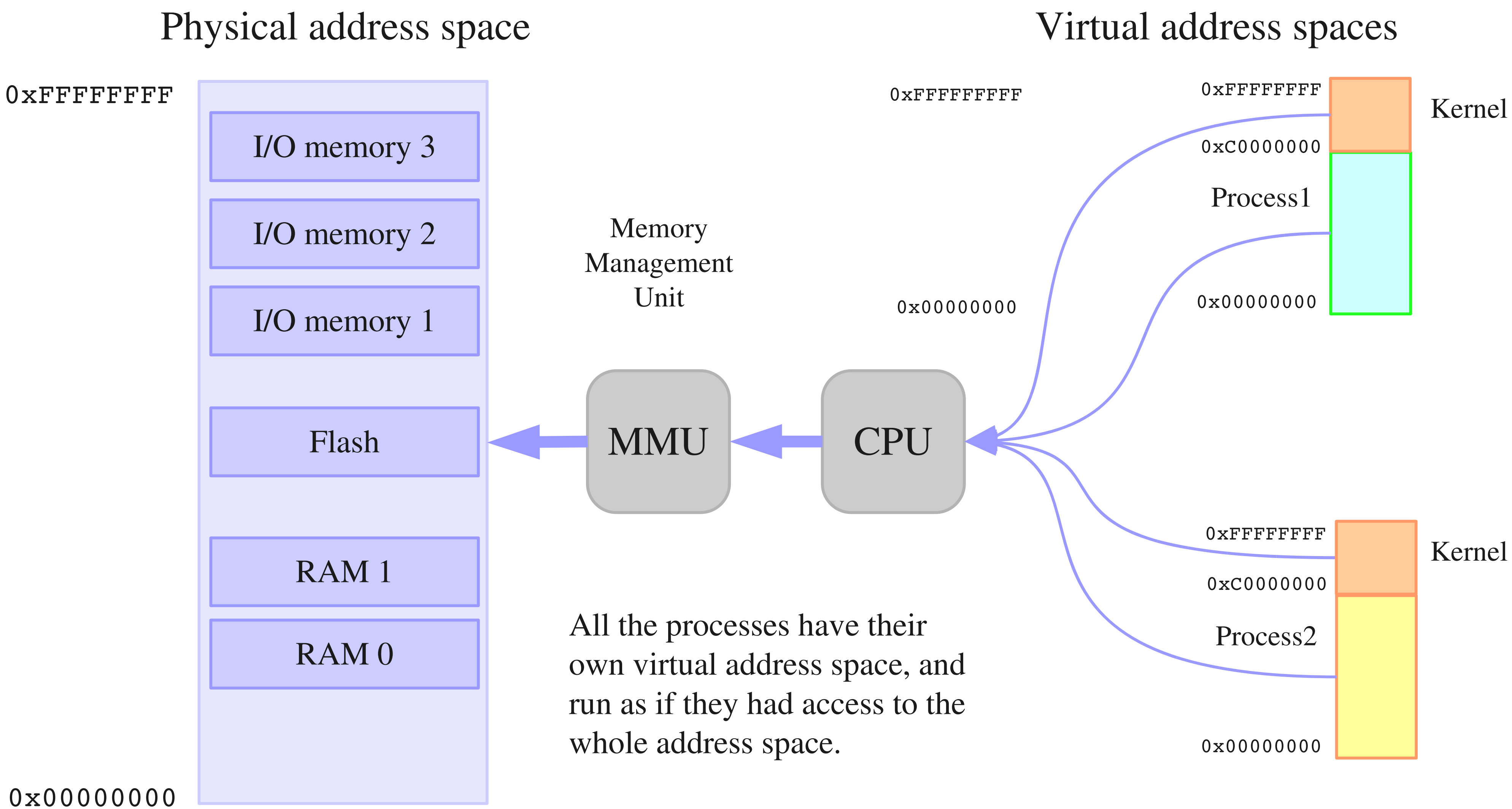
Practical lab – Writing modules

- Write a kernel module with several capabilities, including module parameters.
- Access kernel internals from your module.
- Setup the environment to compile it

Driver development

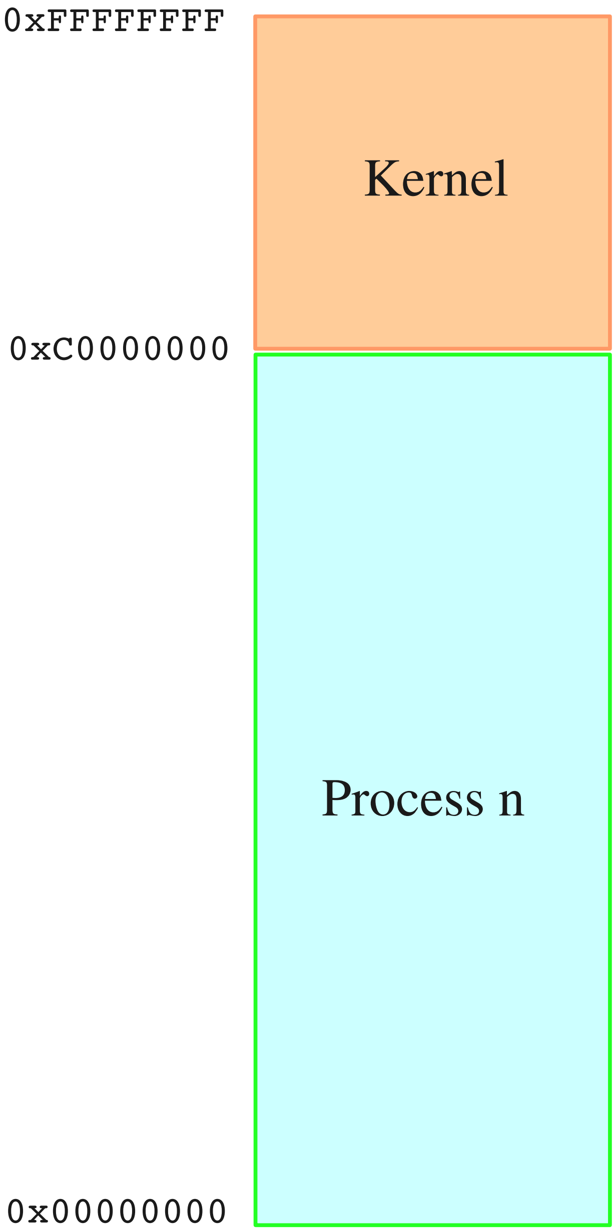
Memory management

Physical and virtual memory



All the processes have their own virtual address space, and run as if they had access to the whole address space.

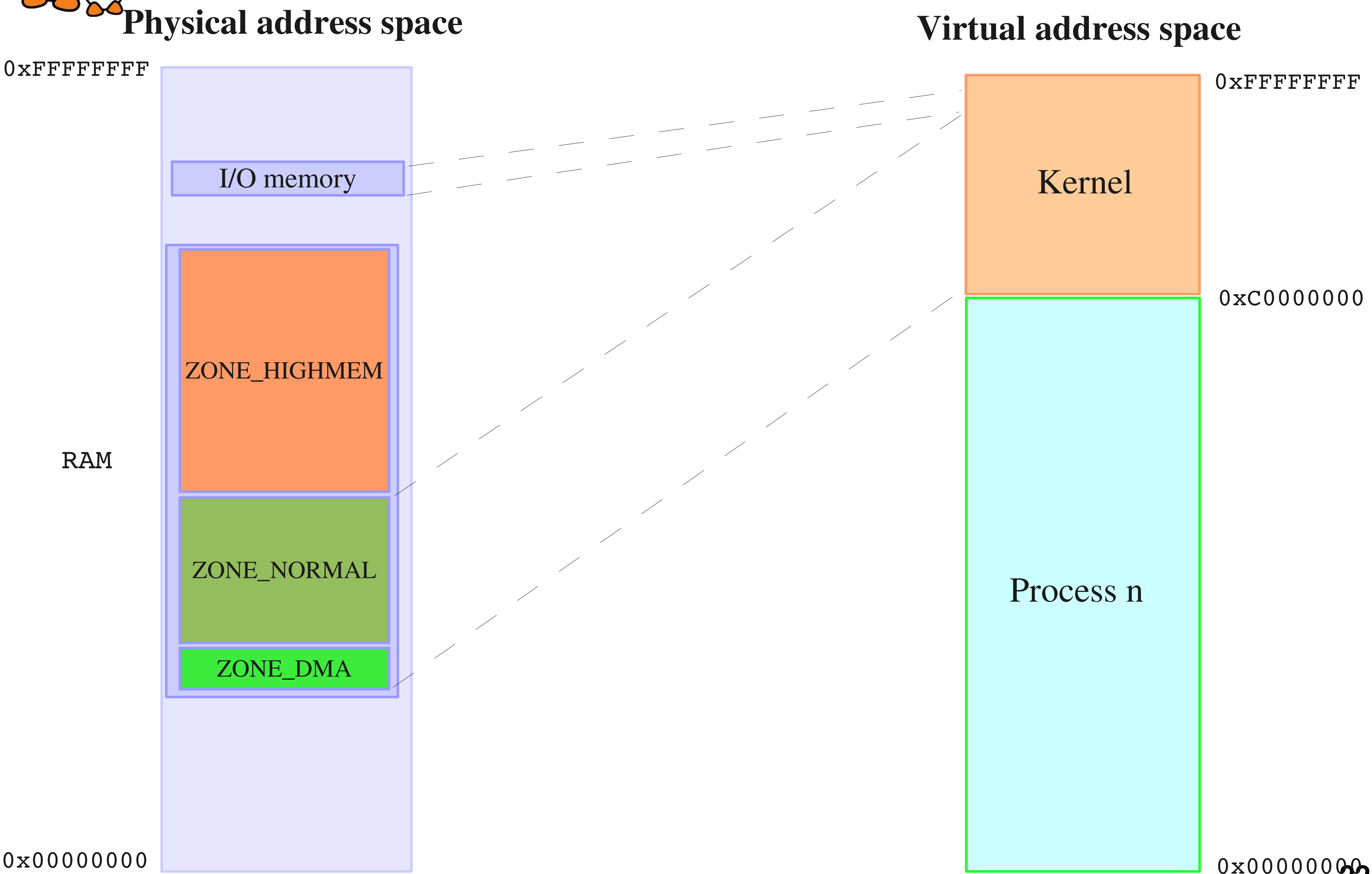
Virtual memory organization: 1GB / 3GB



- 1GB reserved for kernel-space
- Contains kernel code and core data structures, identical in all address spaces
- Most memory can be a direct mapping of physical memory at a fixed offset

- Complete 3GB exclusive mapping available for each user-space process
- Process code and data (program, stack, …)
- Memory-mapped files
- Not necessarily mapped to physical memory (demand fault paging used for dynamic mapping to physical memory pages)
- Differs from one address space to the other

Physical / virtual memory mapping



Accessing more physical memory

- Only less than 1GB memory address-able directly through kernel virtual address space
- If more physical memory is present on the platform:
  - Part of the memory will not be access-able by kernel space, but can be used by user-space
  - To allow kernel to access to more physical memory:
    - Change 1GB/3GB memory split (2GB/2GB) ? => but reduces total memory available for each process
    - Change for a 64bits architecture ;-)
    - Activate the 'highmem' support if available for your architecture:
      - Allows kernel to map parts of its non-directly access-able memory
      - Mapping must be requested explicitly
      - Limited addresses ranges reserved for this usage

Accessing even more physical memory!

- If your 32bits platform hosts more than 4GB, they just cannot be mapped
- The PAE (Physical Address Expansion) may be supported by your architecture
- Adds some address extension bits used to index memory areas
- Allows accessing up to 64GB of physical memory by 4GB pages
- Note that each user-space process is still limited to a 3GB memory space

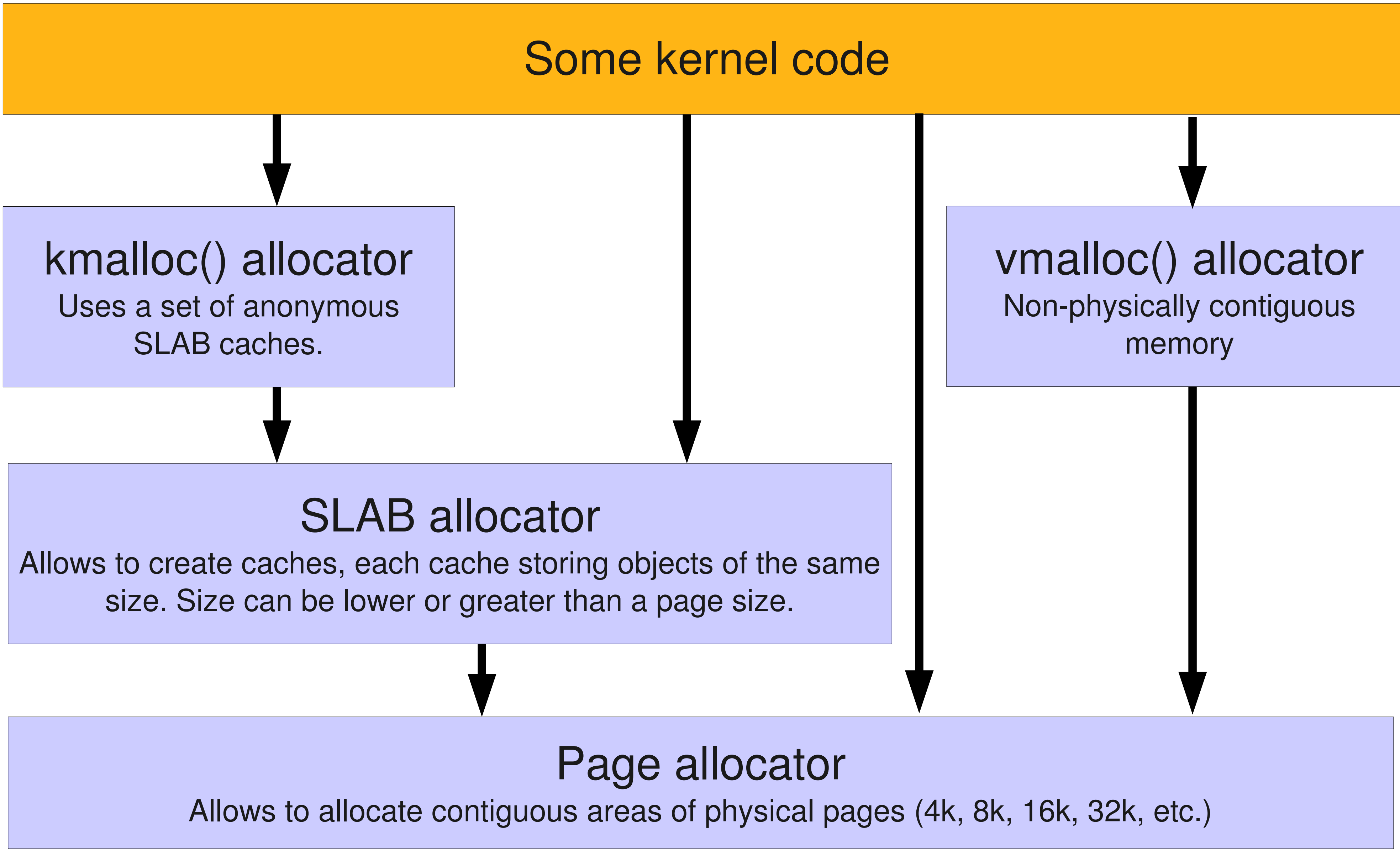
Notes on user-space memory

- New user-space memory is allocated either from the already allocated process memory, or using the mmap system call
- Note that memory allocated may not be physically allocated:
  - Kernel uses demand fault paging to allocate the physical page (the physical page is allocated when access to the virtual address generates a page fault)
  - ... or may have been swapped out, which also induces a page fault
- User space memory allocation is allowed to over-commit memory (more than available physical memory) => can lead to out of memory
- OOM killer enters in action and selects a process to kill to retrieve some memory

Back to kernel memory

- Kernel memory allocators (see following slides) allocate physical pages, and kernel allocated memory cannot be swapped out, so no fault handling required for kernel memory
- Most kernel memory allocation functions also return a kernel virtual address to be used within the kernel space
- Kernel memory low-level allocator manages pages. This is the finest granularity (usually 4kB, architecture dependent)
- However, the kernel memory management handles smaller memory allocations through its allocator (see slabs / SLUB allocator – used by kmalloc)

Allocators in the kernel



Page allocator

- Appropriate for large allocations
- A page is usually 4K, but can be made greater in some architectures (`sh`, `mips`: 4, 8, 16 or 64K, but not configurable in `i386` or `arm`).
- Buddy allocator strategy, so only allocations of power of two number of pages are possible: 1 page, 2 pages, 4 pages, 8 pages, 16 pages, etc.
- Typical maximum size is 8192 KB, but it might depend on the kernel configuration.
- The allocated area is virtually contiguous (of course), but also physically contiguous. It is allocated in the identity-mapped part of the kernel memory space.
  - This means that large areas may not be available or hard to retrieve due to physical memory fragmentation.

Page allocator API

- `unsigned long get_zeroed_page(int flags);`
  Returns the virtual address of a free page, initialized to zero
- `unsigned long __get_free_page(int flags);`
  Same, but doesn't initialize the contents
- ```
  unsigned long __get_free_pages(int flags,
                                 unsigned int order);
  ```
  Returns the starting virtual address of an area of several contiguous pages in physical RAM, with `order` being `log2(<number_of_pages>)`.Can be computed from the size with the `get_order()` function.
- `void free_page(unsigned long addr);`
  Frees one page.
- ```
  void free_pages(unsigned long addr,
                  unsigned int order);
  ```
  Frees multiple pages. Need to use the same `order` as in allocation.

Page allocator flags

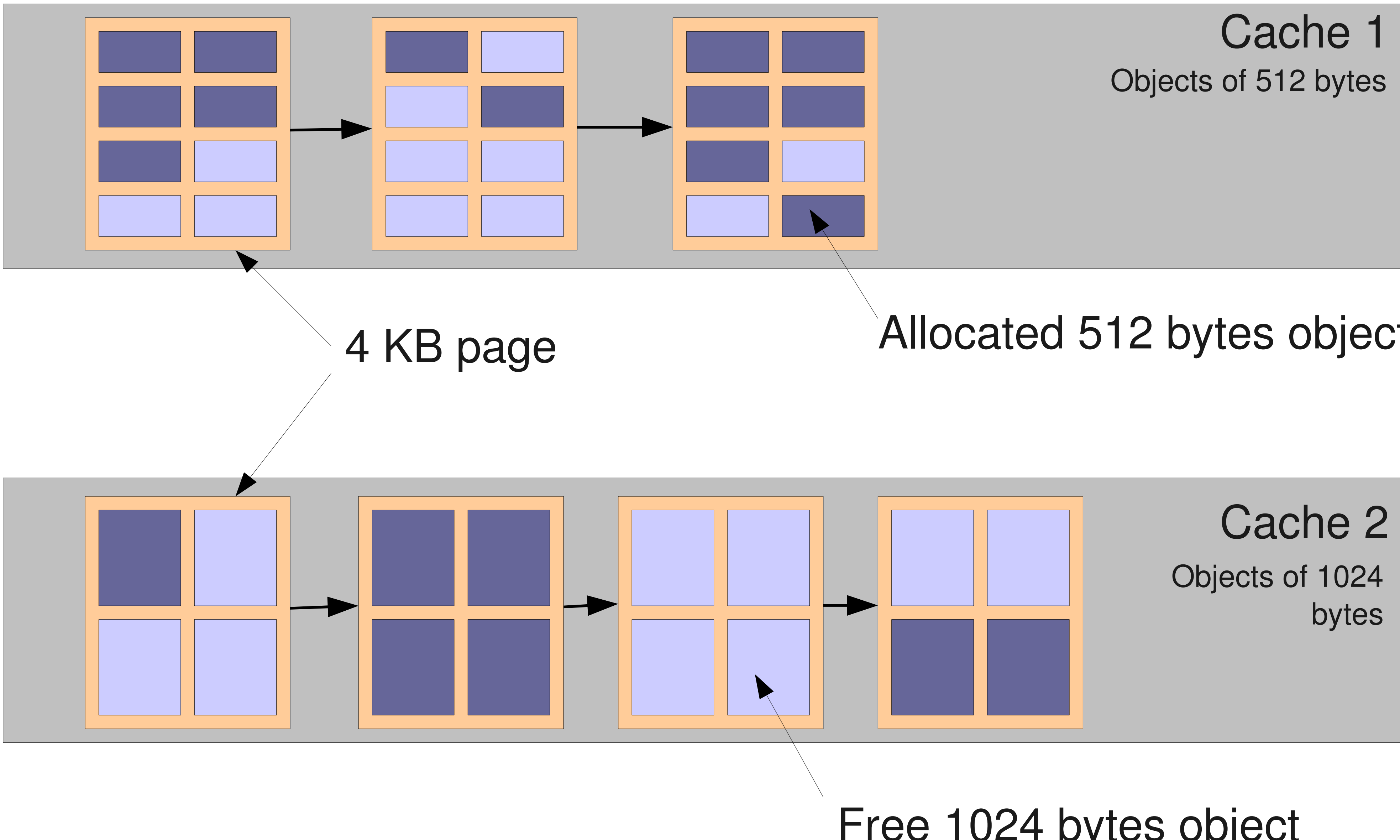
The most common ones are:

- `GFP_KERNEL`
  Standard kernel memory allocation. The allocation may block in order to find enough available memory. Fine for most needs, except in interrupt handler context.
- `GFP_ATOMIC`
  RAM allocated from code which is not allowed to block (interrupt handlers or critical sections). Never blocks, allows to access emegency pools, but can fail if no free memory is readily available.
- `GFP_DMA`
  Allocates memory in an area of the physical memory usable for DMA transfers.
- Others are defined in `include/linux/gfp.h` (GFP: `__get_free_pages`).

SLAB allocator

- The SLAB allocator allows to create caches, which contains a set of objects of the same size
- The object size can be smaller or greater than the page size
- The SLAB allocator takes care of growing or reducing the size of the cache as needed, depending on the number of allocated objects. It uses the page allocator to allocate and free pages.
- SLAB caches are used for data structures that are present in many many instances in the kernel: directory entries, file objects, network packet descriptors, process descriptors, etc.
  - See `/proc/slabinfo`
- They are rarely used for individual drivers.
- See `include/linux/slab.h` for the API

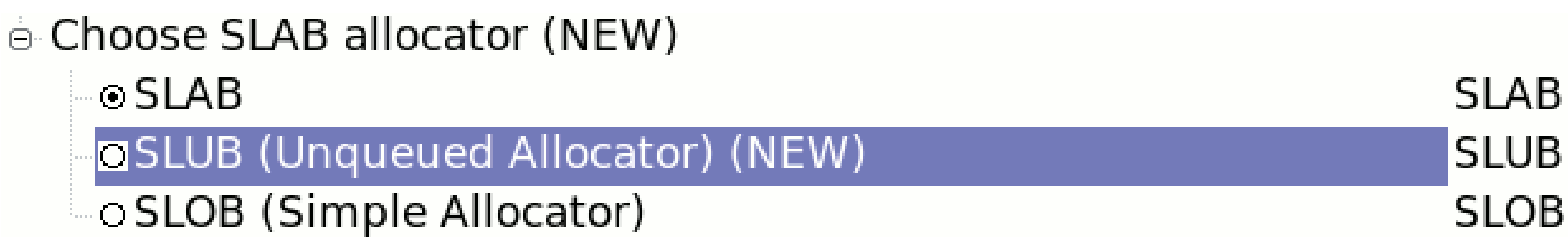
SLAB allocator (2)



Different SLAB allocators

There are three different, but API compatible, implementations of a SLAB allocator in the Linux kernel. A particular implementation is choosen at configuration time.

- SLAB: original, well proven allocator in Linux 2.6.
- SLOB: much simpler. More space efficient but doesn't scale well. Saves a few hundreds of KB in small systems (depends on `CONFIG_EMBEDDED`)
- SLUB: the new default allocator since 2.6.23, simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation.



kmalloc allocator

- The kmalloc allocator is the general purpose memory allocator in the Linux kernel, for objects from 8 bytes to 128 KB
- For small sizes, it relies on generic SLAB caches, named `kmalloc-XXX` in `/proc/slabinfo`
- For larger sizes, it relies on the page allocator
- The allocated area is guaranteed to be physically contiguous
- The allocated area size is rounded up to the next power of two size (while using the SLAB allocator directly allows to have more flexibility)
- It uses the same flags as the page allocator (GFP_KERNEL, GFP_ATOMIC, GFP_DMA, etc.) with the same semantics.
- Should be used as the primary allocator unless there is a strong reason to use another one.

kmalloc API

- `#include <linux/slab.h>`
- `void *kmalloc(size_t size, int flags);`
  Allocate size bytes, and return a pointer to the area (virtual address)
  - `size`: number of bytes to allocate
  - `flags`: same flags as the page allocator
- `void kfree (const void *objp);`
  Free an allocated area
- Example: (`drivers/infiniband/core/cache.c`)

```
struct ib_update_work *work;
work = kmalloc(sizeof *work, GFP_ATOMIC);
...
kfree(work);
```

kmalloc API (2)

- `void *kzalloc(size_t size, gfp_t flags);`
  Allocates a zero-initialized buffer
- ```
  void *kcalloc(size_t n, size_t size,
                gfp_t flags);
  ```
  Allocates memory for an array of `n` elements of size `size`, and zeroes its contents.
- ```
  void *krealloc(const void *p, size_t new_size,
                 gfp_t flags);
  ```
  Changes the size of the buffer pointed by `p` to `new_size`, by reallocating a new buffer and copying the data, unless the `new_size` fits within the alignment of the existing buffer.

vmalloc allocator

- The `vmalloc` allocator can be used to obtain virtually contiguous memory zones, but not physically contiguous. The requested memory size is rounded up to the next page.
- The allocated area is in the kernel space part of the address space, but outside of the identically-mapped area
- Allocations of fairly large areas is possible, since physical memory fragmentation is not an issue, but areas cannot be used for DMA, as DMA usually requires physically contiguous buffers.
- API in `<linux/vmalloc.h>`
  - `void *vmalloc(unsigned long size);`
    Returns a virtual address
  - `void vfree(void *addr);`

Kernel memory debugging

Debugging features available since 2.6.31

- Kmemcheck
  Dynamic checker for access to uninitialized memory.
  Only available on x86 so far, but will help to improve architecture independent code anyway.
  See Documentation/kmemcheck.txt for details.
- Kmemleak
  Dynamic checker for memory leaks
  This feature is available for all architectures.
  See Documentation/kmemleak.txt for details.

Both have a significant overhead. Only use them in development!

Driver development

Useful general-purpose kernel APIs

Memory/string utilities

- In `<linux/string.h>`
  - Memory-related: `memset, memcpy, memmove, memscan, memcmp, memchr`
  - String-related: `strcpy, strcat, strcmp, strchr, strrchr, strlen and variants`
  - Allocate and copy a string: `kstrdup, kstrndup`
  - Allocate and copy a memory area: `kmemdup`
- In `<linux/kernel.h>`
  - String to int conversion: `simple_strtoul, simple_strtol, simple_strtoull, simple_strtoll`
  - Other string functions: `sprintf, sscanf`

Linked lists

- Convenient linked-list facility in `<linux/list.h>`
  - Used in thousands of places in the kernel
- Add a `struct list_head` member to the structure whose instances will be part of the linked list. It is usually named `node` when each instance needs to only be part of a single list.
- Define the list with the `LIST_HEAD` macro for a global list, or define a `struct list_head` element and initialize it with `INIT_LIST_HEAD` for lists embedded in a structure.
- Then use the `list_*()` API to manipulate the list
  - Add elements: `list_add()`, `list_add_tail()`
  - Remove, move or replace elements: `list_del()`, `list_move()`, `list_move_tail()`, `list_replace()`
  - Test the list: `list_empty()`
  - Iterate over the list: `list_for_each_*()` family of macros

Linked lists example

From include/linux/atmel_tc.h

```
struct atmel_tc {
    /* some members */
    struct list_head node;
};
```

Definition of a list element, with a `struct list_head` member

From drivers/misc/atmel_tclib.c

```
static LIST_HEAD(tc_list);

struct atmel_tc *atmel_tc_alloc(unsigned block, const char *name) {
    struct atmel_tc *tc;
    list_for_each_entry(tc, &tc_list, node) {
        /* Do something with tc */
    }
    [...]
}

static int __init tc_probe(struct platform_device *pdev) {
    struct atmel_tc *tc;
    tc = kzalloc(sizeof(struct atmel_tc), GFP_KERNEL);
    list_add_tail(&tc->node, &tc_list);
}
```

The global list

Iterate over the list elements

Add an element to the list

Driver development

I/O memory and ports

Port I/O vs. Memory-Mapped I/O

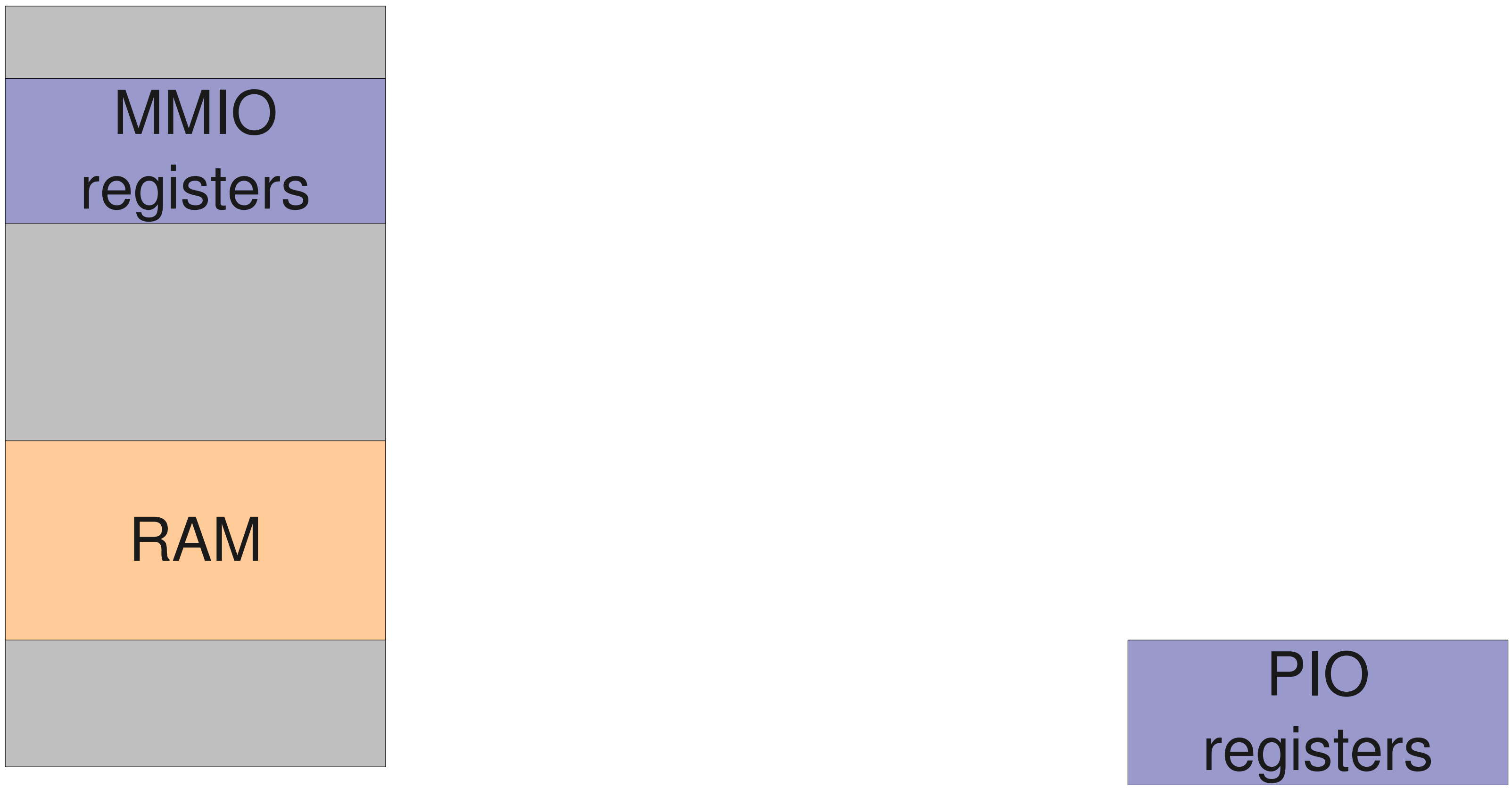
MMIO

- Same address bus to address memory and I/O devices
- Access to the I/O devices using regular instructions
- Most widely used I/O method across the different architectures supported by Linux

PIO

- Different address spaces for memory and I/O devices
- Uses a special class of CPU instructions to access I/O devices
- Example on x86: IN and OUT instructions

MMIO vs PIO



Physical memory address space, accessed with normal load/store instructions

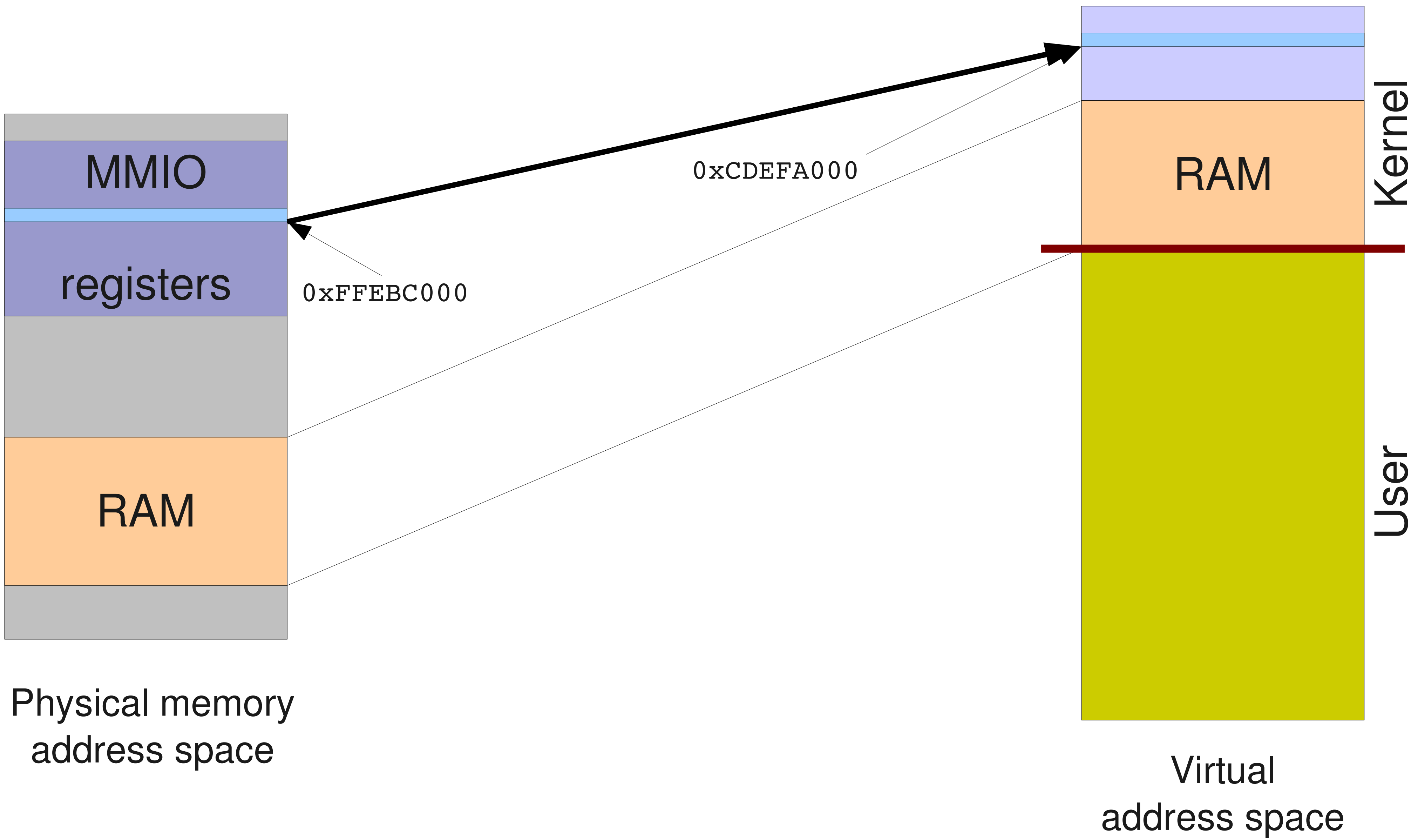
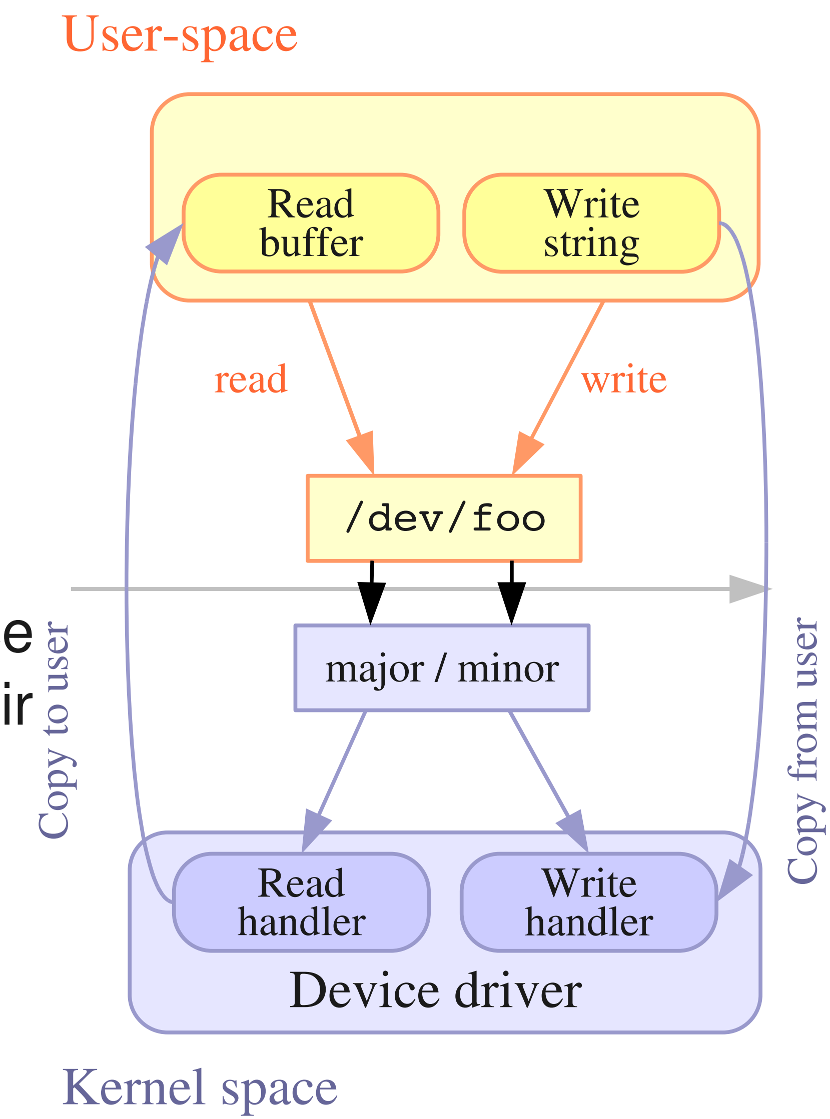
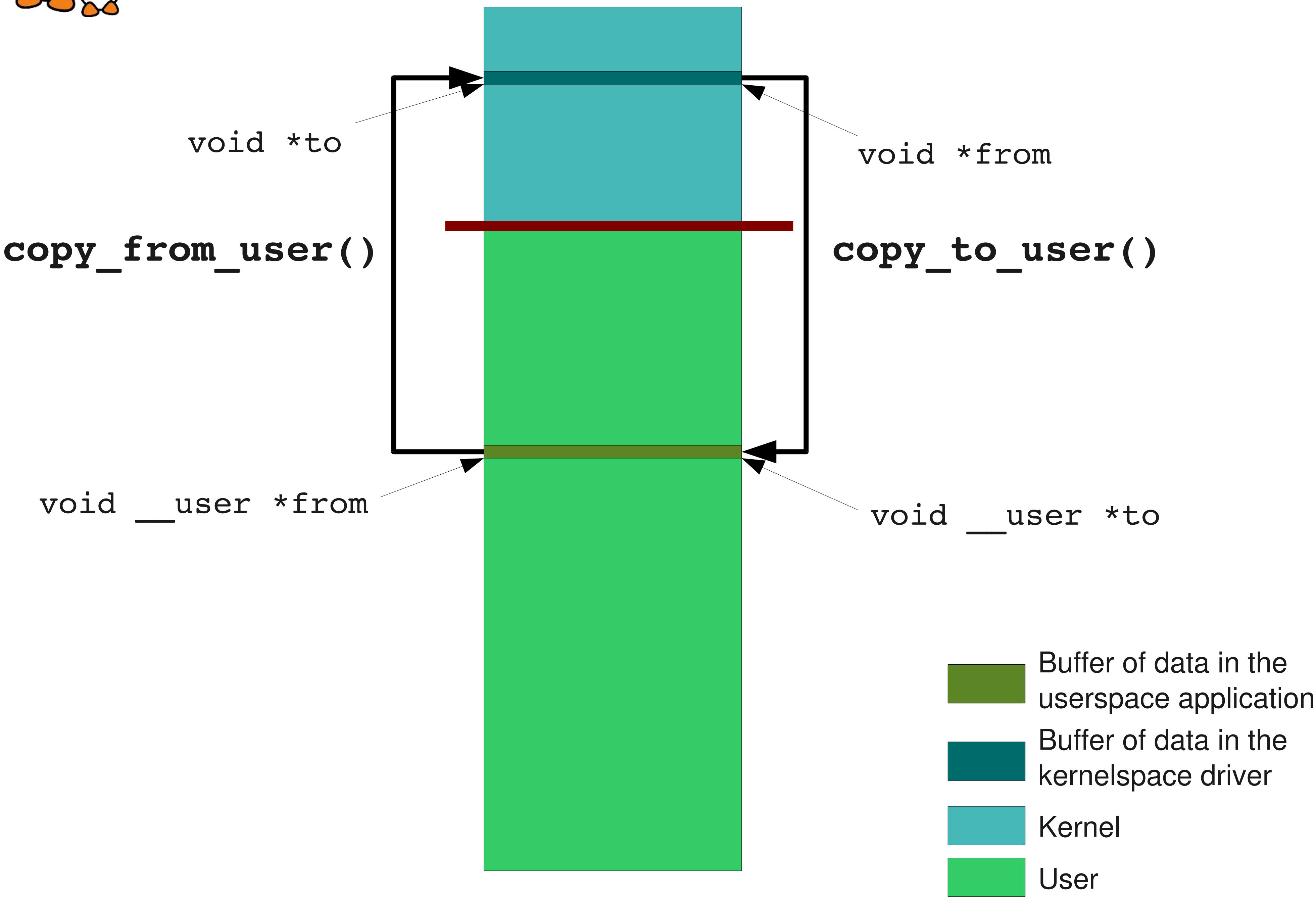
Separate I/O address space, accessed with specific CPU instructions

Requesting I/O ports

`/proc/ioports` example (x86)

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0100-013f : pcmcia_socket0
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
0378-037a : parport0
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial
0800-087f : 0000:00:1f.0
0800-0803 : PM1a_EVT_BLK
0804-0805 : PM1a_CNT_BLK
0808-080b : PM_TMR
0820-0820 : PM2_CNT_BLK
0828-082f : GPE0_BLK
...
```

- Tells the kernel which driver is using which I/O ports
- Allows to prevent other drivers from using the same I/O ports, but is purely voluntary.
- `struct resource *request_region(`
  ```
  struct resource *request_region(
      unsigned long start,
      unsigned long len,
      char *name);
  ```
  Tries to reserve the given region and returns `NULL` if unsuccessful.
  ```
  request_region(0x0170, 8, "ide1");
  ```
- ```
  void release_region(
      unsigned long start,
      unsigned long len);
  ```
  ```

Accessing I/O ports

- Functions to read/write bytes (`b`), word (`w`) and longs (`l`) to I/O ports:

```
unsigned in[bwl](unsigned long *addr);
void out[bwl](unsigned port, unsigned long *addr);
```

- And the strings variants: often more efficient than the corresponding C loop, if the processor supports such operations!

```
void ins[bwl](unsigned port, void *addr,
               unsigned long count);
void outs[bwl](unsigned port, void *addr,
               unsigned long count);
```

- Examples
  - read 8 bits

    ```
    oldlcr = inb(baseio + UART_LCR);
    ```

  - write 8 bits

    ```
    outb(MOXA_MUST_ENTER_ENCHANCE, baseio + UART_LCR);
    ```

Requesting I/O memory

/proc/iomem example

```
00000000-0009efff : System RAM
0009f000-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000cffff : Video ROM
000f0000-000fffff : System ROM
00100000-3ffadfff : System RAM
  00100000-0030afff : Kernel code
  0030b000-003b4bff : Kernel data
3ffae000-3fffffff : reserved
40000000-400003ff : 0000:00:1f.1
40001000-40001fff : 0000:02:01.0
  40001000-40001fff : yenta_socket
40002000-40002fff : 0000:02:01.1
  40002000-40002fff : yenta_socket
40400000-407fffff : PCI CardBus #03
40800000-40bfffff : PCI CardBus #03
40c00000-40ffffff : PCI CardBus #07
41000000-413fffff : PCI CardBus #07
a0000000-a0000fff : pcmcia_socket0
a0001000-a0001fff : pcmcia_socket1
e0000000-e7ffffff : 0000:00:00.0
e8000000-efffffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
...
```

- Functions equivalent to `request_region()` and `release_region()`, but for I/O memory.
- ```
  struct resource * request_mem_region(
     unsigned long start,
     unsigned long len,
     char *name);
  ```
- ```
  void release_mem_region(
     unsigned long start,
     unsigned long len);
  ```

Mapping I/O memory in virtual memory

- Load/store instructions work with virtual addresses
- To access I/O memory, drivers need to have a virtual address that the processor can handle, because I/O memory is not mapped by default in virtual memory.
- The `ioremap` functions satisfy this need:

```
#include <asm/io.h>;

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

- Caution: check that `ioremap` doesn't return a `NULL` address!

ioremap()

`ioremap(0xFFEBC00, 4096) = 0xCDEFA000`



Accessing MMIO devices

- Directly reading from or writing to addresses returned by `ioremap` ("pointer dereferencing") may not work on some architectures.
- To do PCI-style, little-endian accesses, conversion being done automatically

```
unsigned read[bwl](void *addr);
void write[bwl](unsigned val, void *addr);
```

- To do raw access, without endianess conversion

```
unsigned __raw_read[bwl](void *addr);
void __raw_write[bwl](unsigned val, void *addr);
```

- Example
  - 32 bits write

```
__raw_writel(1 << KS8695_IRQ_UART_TX,
               membase + KS8695_INTST);
```

New API for mixed accesses

- A new API allows to write drivers that can work on either devices accessed over PIO or MMIO. A few drivers use it, but there doesn't seem to be a consensus in the kernel community around it.
- Mapping
  - For PIO: `ioport_map()` and `ioport_unmap()`. They don't really map, but they return a special cookie.
  - For MMIO: `ioremap()` and `iounmap()`. As usual.
- Access, works both on addresses returned by `ioport_map()` and `ioremap()`
  - `ioread[8/16/32]()` and `iowrite[8/16/32]` for single access
  - `ioread_rep[8/16/32]()` and `iowrite_rep[8/16/32]()` for repeated accesses

Avoiding I/O access issues

- Caching on I/O ports or memory already disabled
- Use the macros, they do the right thing for your architecture
- The compiler and/or CPU can reorder memory accesses, which might cause troubles for your devices is they expect one register to be read/written before another one.
  - Memory barriers are available to prevent this reordering
  - `rmb()` is a read memory barrier, prevents reads to cross the barrier
  - `wmb()` is a write memory barrier
  - `mb()` is a read-write memory barrier
- Starts to be a problem with CPU that reorder instructions and SMP.
- See `Documentation/memory-barriers.txt` for details

/dev/mem

- Used to provide user-space applications with direct access to physical addresses.
- Usage: open `/dev/mem` and read or write at given offset. What you read or write is the value at the corresponding physical address.
- Used by applications such as the X server to write directly to device memory.
- On `x86`, `arm` and `tile`: `CONFIG_STRICT_DEVMEM` option to restrict `/dev/mem` non-RAM addresses, for security reasons (2.6.37-rc2 status).

Practical lab – I/O memory and ports

- Make a remote connection to your board through ssh.
- Access the system console through the network.
- Reserve the I/O memory addresses used by the serial port.
- Read device registers and write data to them, to send characters on the serial port.

Driver development

Character drivers

Usefulness of character drivers

- Except for storage device drivers, most drivers for devices with input and output flows are implemented as character drivers.
- So, most drivers you will face will be character drivers
  You will regret if you sleep during this part!

User-space needs

- The name of a device file in `/dev` to interact with the device driver through regular file operations (open, read, write, close...)

The kernel needs

- To know which driver is in charge of device files with a given major / minor number pair
- For a given driver, to have handlers ("*file operations*") to execute when user-space opens, reads, writes or closes the device file.



Implementing a character driver

- Four major steps
  - Implement operations corresponding to the system calls an application can apply to a file: file operations
  - Define a `file_operations` structure associating function pointers to their implementation in your driver
  - Reserve a set of major and minors for your driver
  - Tell the kernel to associate the reserved major and minor to your file operations
- This is a very common design scheme in the Linux kernel
  - A common kernel infrastructure defines a set of operations to be implemented by a driver and functions to register your driver
  - Your driver only needs to implement this set of well-defined operations

File operations

- Before registering character devices, you have to define `file_operations` (called *fops*) for the device files.
- The file_operations structure is generic to all files handled by the Linux kernel. It contains many operations that aren't needed for character drivers.
- Here are the most important operations for a character driver. All of them are optional.

```
struct file_operations {
  [...]
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*release) (struct inode *, struct file *);
  [...]
};
```

open() and release()

- `int foo_open (struct inode *i, struct file *f)`
  - Called when user-space opens the device file.
  - `inode` is a structure that uniquely represent a file in the system (be it a regular file, a directory, a symbolic link, a character or block device)
  - `file` is a structure created every time a file is opened. Several file structures can point to the same `inode` structure.
    - Contains informations like the current position, the opening mode, etc.
    - Has a `void *private_data` pointer that one can freely use.
    - A pointer to the file structure is passed to all other operations
- `int foo_release(struct inode *i, struct file *f)`
  - Called when user-space closes the file.

read()

- `ssize_t foo_read( struct file *f, __user char *buf, size_t sz, loff_t *off)`
  - Called when user-space uses the `read()` system call on the device.
  - Must read data from the device, write at most `sz` bytes in the *user-space* buffer `buf`, and update the current position in the file `off`. `f` is a pointer to the same file structure that was passed in the `open()` operation
  - Must return the number of bytes read.
  - On Unix, read() operations typically block when there isn't enough data to read from the device

write()

- `ssize_t foo_write( struct file *f, __user const char *buf, size_t sz ,loff_t *off)`
  - Called when user-space uses the `write()` system call on the device
  - The opposite of `read`, must read at most `sz` bytes from `buf`, write it to the device, update `off` and return the number of bytes written.

Exchanging data with user-space (1)

- Kernel code isn't allowed to directly access user-space memory, using `memcpy` or direct pointer dereferencing
  - Doing so does not work on some architectures
  - If the address passed by the application was invalid, the application would segfault
- To keep the kernel code portable and have proper error handling, your driver must use special kernel functions to exchange data with user-space

Exchanging data with user-space (2)

- A single value
  - `get_user(v, p);`
    The kernel variable v gets the value pointer by the user-space pointer p
  - `put_user(v, p);`
    The value pointed by the user-space pointer p is set to the contents of the kernel variable v.
- A buffer
  - ```
    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);
    ```
- The return value must be checked. Zero on success, non-zero on failure. If non-zero, the convention is to return `-EFAULT`.

Exchanging data with user-space (3)



read operation example

```
static ssize_t
acme_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
   /* The acme_buf address corresponds to a device I/O memory area */
   /* of size acme_bufsize, obtained with ioremap() */
   int remaining_size, transfer_size;

   remaining_size = acme_bufsize - (int) (*ppos); // bytes left to transfer
   if (remaining_size == 0) { /* All read, returning 0 (End Of File) */
      return 0;
   }

   /* Size of this transfer */
   transfer_size = min(remaining_size, (int) count);

   if (copy_to_user(buf /* to */, acme_buf + *ppos /* from */, transfer_size)) {
      return -EFAULT;
   } else { /* Increase the position in the open file */
      *ppos += transfer_size;
      return transfer_size;
   }
}
```

Read method

Piece of code available in
http://free-electrons.com/doc/c/acme.c

write operation example

```
static ssize_t
acme_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
{
   int remaining_bytes;

   /* Number of bytes not written yet in the device */
   remaining_bytes = acme_bufsize - (*ppos);

   if (count > remaining_bytes) {
      /* Can't write beyond the end of the device */
      return -EIO;
   }

   if (copy_from_user(acme_buf + *ppos /* to */, buf /* from */, count)) {
      return -EFAULT;
   } else {
      /* Increase the position in the open file */
      *ppos += count;
      return count;
   }
}
```

Write method

Piece of code available in
http://free-electrons.com/doc/c/acme.c

unlocked_ioctl()

```
long unlocked_ioctl(struct file *f,
          unsigned int cmd, unsigned long arg)
```

- Associated to the `ioctl()` system call
  Called `unlocked` because it doesn't hold the Big Kernel Lock.
- Allows to extend the driver capabilities beyond the limited read/write API
- For example: changing the speed of a serial port, setting video output format, querying a device serial number...
- `cmd` is a number identifying the operation to perform
- `arg` is the optional argument passed as third argument of the `ioctl`() system call. Can be an integer, an address, etc.
- The semantic of `cmd` and `arg` is driver-specific.

ioctl() example: kernel side

```
static long phantom_ioctl(struct file *file, unsigned int cmd,
                          unsigned long arg)
{
        struct phm_reg r;
        void __user *argp = (void __user *)arg;

        switch (cmd) {
        case PHN_SET_REG:
                if (copy_from_user(&r, argp, sizeof(r)))
                        return -EFAULT;
                /* Do something */
                break;
        case PHN_GET_REG:
                if (copy_to_user(argp, &r, sizeof(r)))
                        return -EFAULT;

                /* Do something */
                break;
        default:
                return -ENOTTY;
        }

        return 0;
}
```

Selected excerpt from drivers/misc/phantom.c

Ioctl() example: application side

```
int main(void)
{
   int fd, ret;
   struct phm_reg reg;

   fd = open("/dev/phantom");
   assert(fd > 0);

   reg.field1 = 42;
   reg.field2 = 67;

   ret = ioctl(fd, PHN_SET_REG, & reg);
   assert(ret == 0);

   return 0;
}
```

file operations definition example (3)

Defining a `file_operations` structure:

```
#include <linux/fs.h>

static struct file_operations acme_fops =
{
   .owner = THIS_MODULE,
   .read = acme_read,
   .write = acme_write,
};
```

You just need to supply the functions you implemented! Defaults for other functions (such as `open`, `release`...) are fine if you do not implement anything special.

dev_t data type

Kernel data type to represent a major / minor number pair

- Also called a *device number*.
- Defined in `<linux/kdev_t.h>`
  Linux 2.6: 32 bit size (major: 12 bits, minor: 20 bits)
- Macro to compose the device number:
  ```
  MKDEV(int major, int minor);
  ```
- Macro to extract the minor and major numbers:
  ```
  MAJOR(dev_t dev);
  MINOR(dev_t dev);
  ```

Registering device numbers (1)

```
#include <linux/fs.h>

int register_chrdev_region(
     dev_t from,              /* Starting device number */
     unsigned count,          /* Number of device numbers */
     const char *name);       /* Registered name */
```

Returns `0` if the allocation was successful.

Example

```
static dev_t acme_dev = MKDEV(202, 128);


if (register_chrdev_region(acme_dev, acme_count, “acme”)) {
printk(KERN_ERR “Failed to allocate device number\n”);
...
```

Registering device numbers (2)

If you don't have fixed device numbers assigned to your driver

- Better not to choose arbitrary ones.
  There could be conflicts with other drivers.
- The kernel API offers a `alloc_chrdev_region` function
  to have the kernel allocate free ones for you. You can find the allocated major number in `/proc/devices`.

Information on registered devices

Registered devices are visible in `/proc/devices`:

```
Character devices:        Block devices:
  1 mem                     1 ramdisk
  4 /dev/vc/0               3 ide0
  4 tty                     8 sd
  4 ttyS                    9 md
  5 /dev/tty               22 ide1
  5 /dev/console           65 sd
  5 /dev/ptmx              66 sd
  6 lp                     67 sd
 10 misc                   68 sd
 13 input
 14 sound
 ...
```

Major number — Registered name

Character device registration (1)

- The kernel represents character drivers with a `cdev` structure
- Declare this structure globally (within your module):

```
#include <linux/cdev.h>
static struct cdev acme_cdev;
```

- In the init function, initialize the structure:

```
cdev_init(&acme_cdev, &acme_fops);
```

Character device registration (2)

- Then, now that your structure is ready, add it to the system:

```
int cdev_add(
     struct cdev *p,      /* Character device structure */
     dev_t dev,           /* Starting device major / minor number */
     unsigned count);     /* Number of devices */
```

- After this function call, the kernel knows the association between the major/minor numbers and the file operations. Your device is ready to be used!
- Example (continued):

```
if (cdev_add(&acme_cdev, acme_dev, acme_count)) {
   printk (KERN_ERR "Char driver registration failed\n");
...
```

Character device unregistration

- First delete your character device:
```
void cdev_del(struct cdev *p);
```
- Then, and only then, free the device number:
```
void unregister_chrdev_region(dev_t from,
unsigned count);
```
- Example (continued):
```
cdev_del(&acme_cdev);
unregister_chrdev_region(acme_dev, acme_count);
```

Linux error codes

- The kernel convention for error management is
  - Return 0 on success
    `return 0;`
  - Return a negative error code on failure
    `return -EFAULT;`
- Error codes
  - `include/asm-generic/errno-base.h`
  - `include/asm-generic/errno.h`

Char driver example summary (1)

```
static void *acme_buf;
static int acme_bufsize=8192;

static int acme_count=1;
static dev_t acme_dev = MKDEV(202,128);

static struct cdev acme_cdev;

static ssize_t acme_write(...) {...}

static ssize_t acme_read(...) {...}

static struct file_operations acme_fops =
{
    .owner = THIS_MODULE,
    .read = acme_read,
    .write = acme_write
};
```

Char driver example summary (2)

Shows how to handle errors and deallocate resources in the right order!

```
static int __init acme_init(void)
{
    int err;
    acme_buf = ioremap (ACME_PHYS,
                        acme_bufsize);

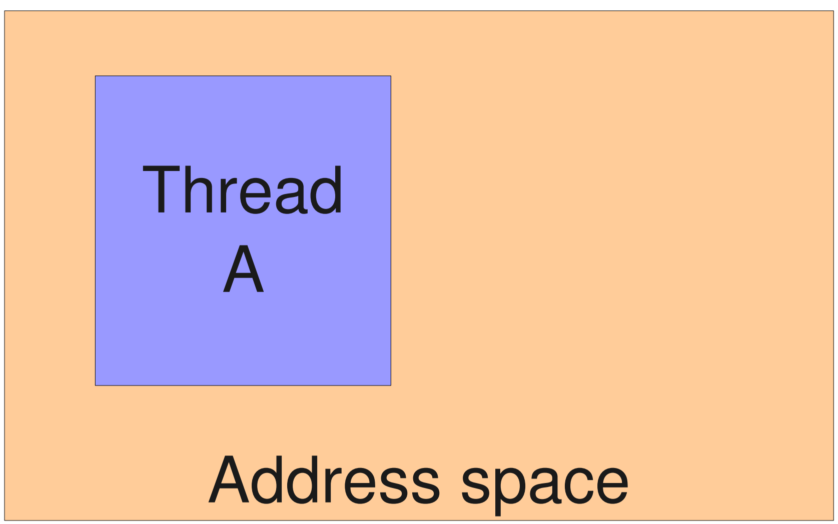
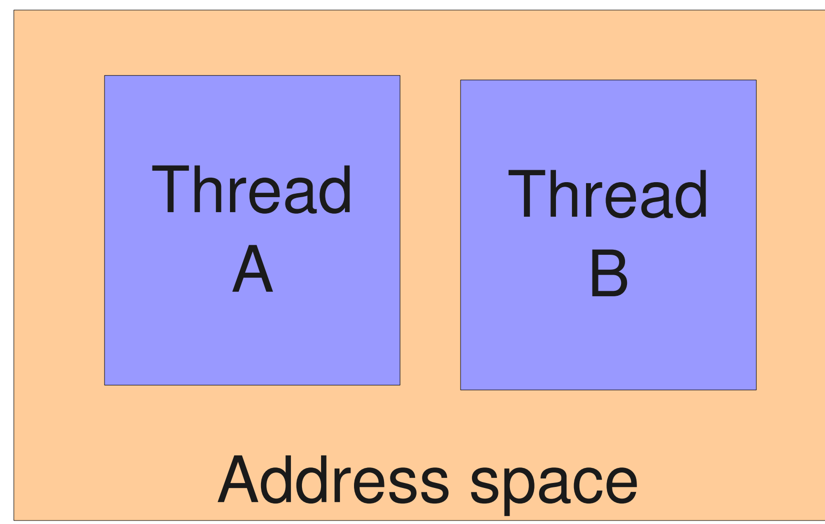
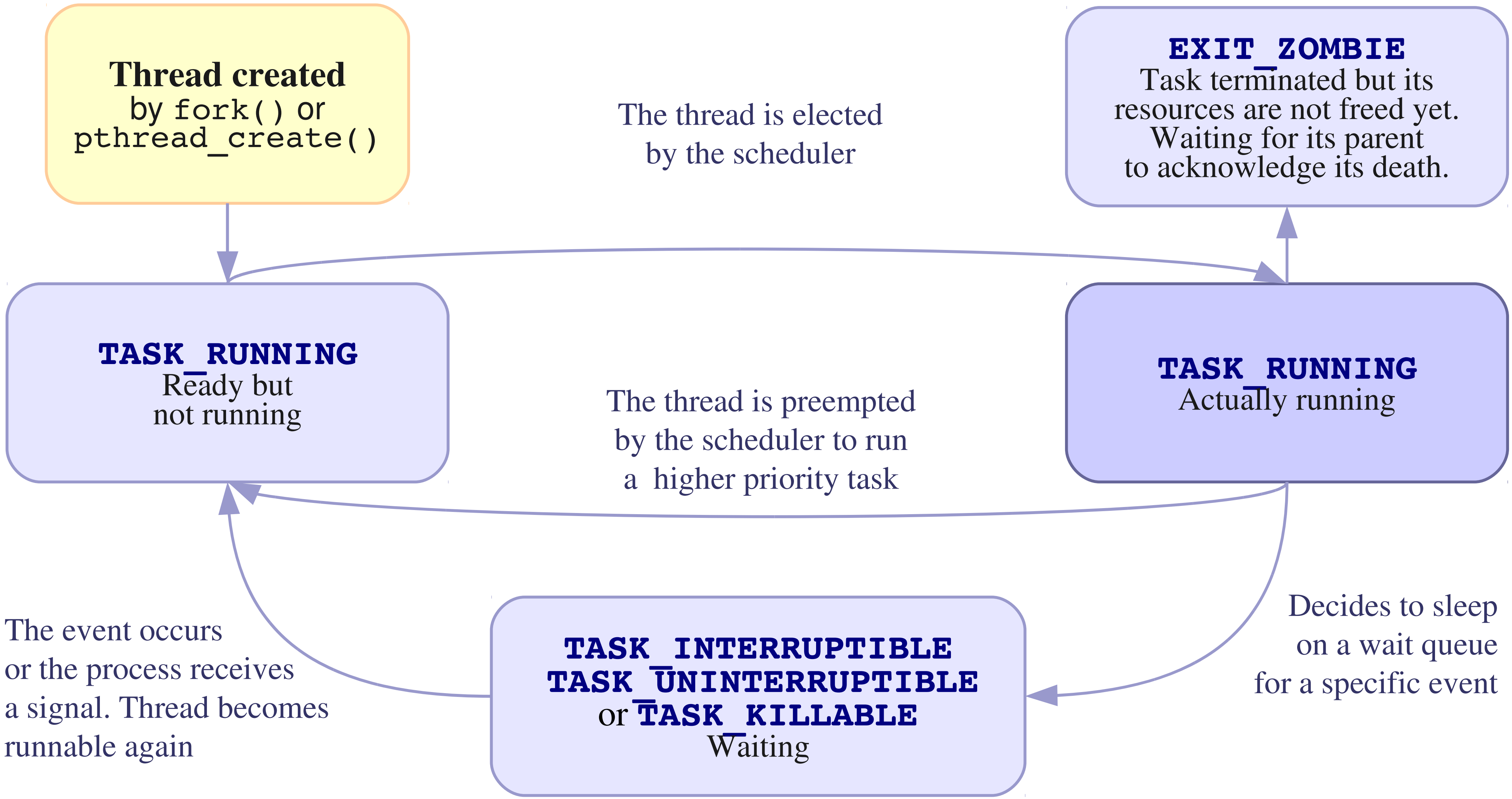
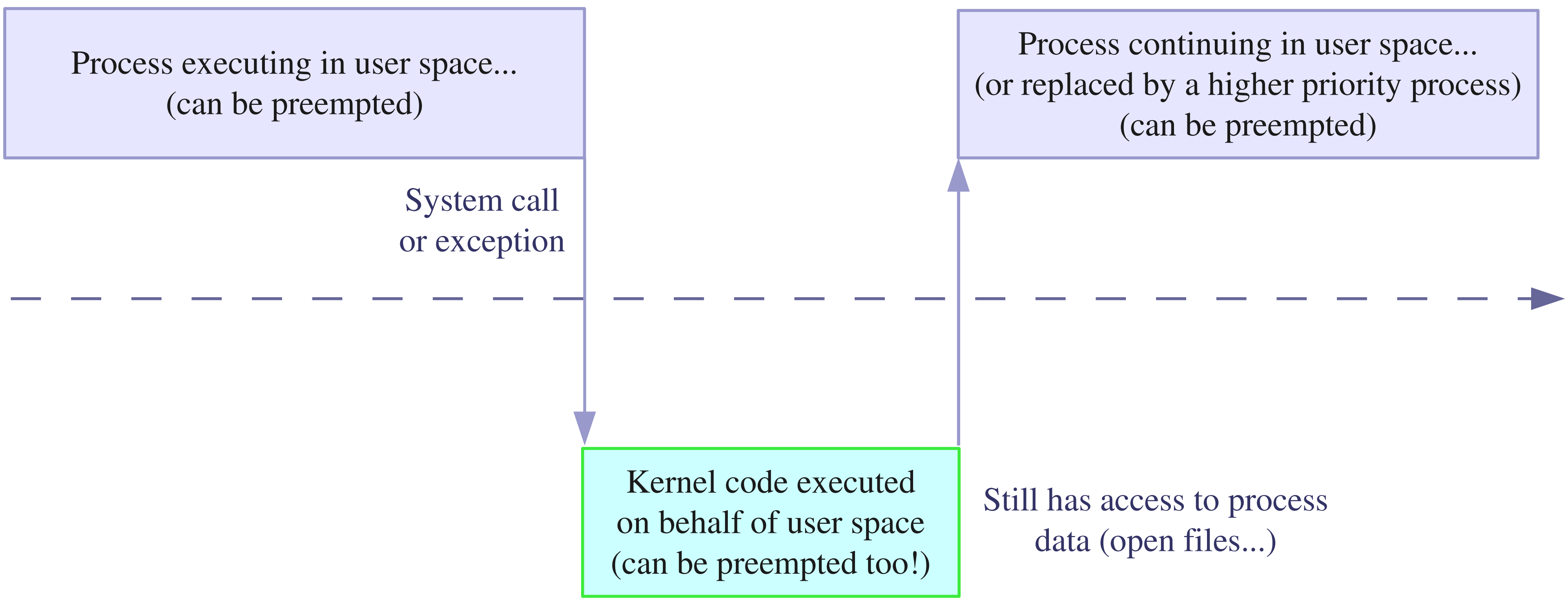
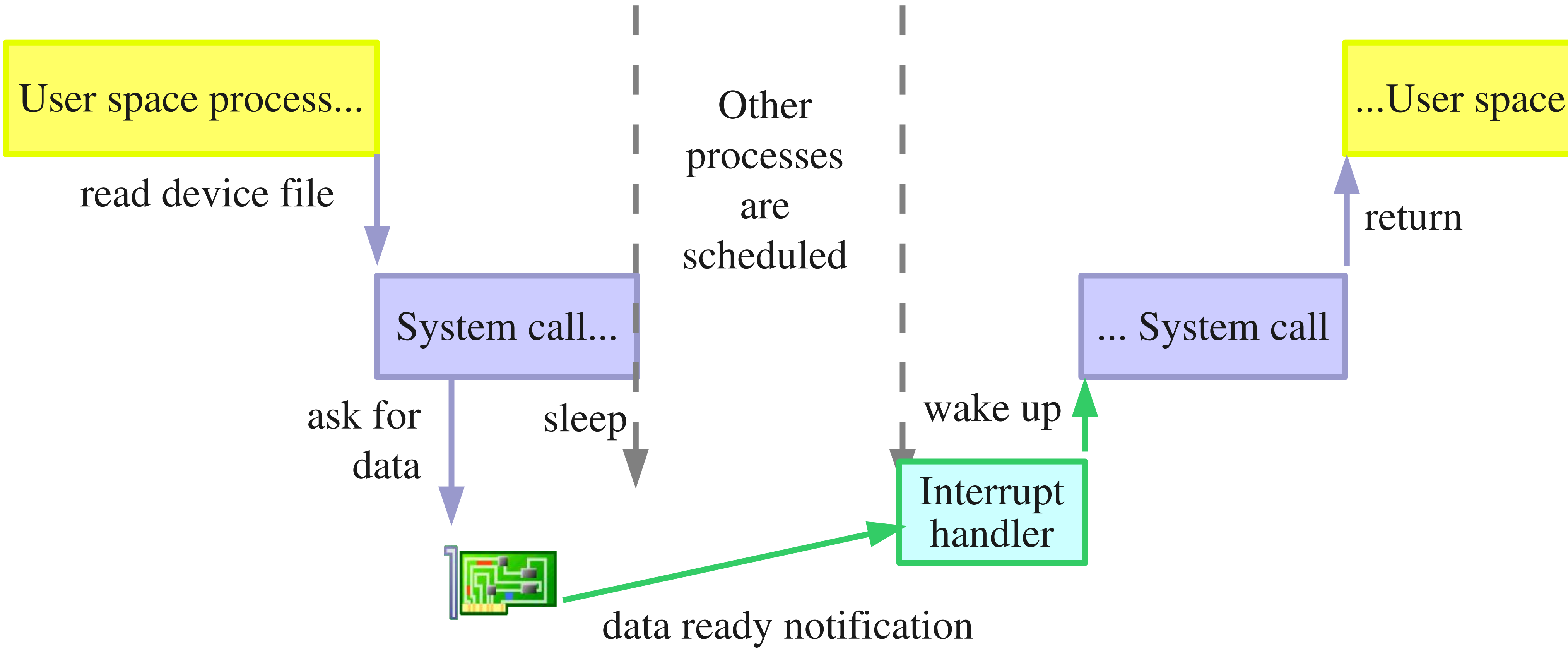
    if (!acme_buf) {
       err = -ENOMEM;
       goto err_exit;
    }

    if (register_chrdev_region(acme_dev,
                     acme_count, "acme")) {
       err=-ENODEV;
       goto err_free_buf;
    }

    cdev_init(&acme_cdev, &acme_fops);

    if (cdev_add(&acme_cdev, acme_dev,
                 acme_count)) {
       err=-ENODEV;
       goto err_dev_unregister;
    }
```

```
    return 0;

    err_dev_unregister:
        unregister_chrdev_region(
           acme_dev, acme_count);
    err_free_buf:
        iounmap(acme_buf);
    err_exit:
        return err;
}

static void __exit acme_exit(void)
{
    cdev_del(&acme_cdev);
    unregister_chrdev_region(acme_dev,
                       acme_count);
    iounmap(acme_buf);
}
```

Complete example code available on http://free-electrons.com/doc/c/acme.c

Character driver summary

Character driver writer - Define the file operations callbacks for the device file: `read`, `write`, `ioctl`... - In the module init function, reserve major and minor numbers with `register_chrdev_region()`, init a `cdev` structure with your file operations and add it to the system with `cdev_add()`. - In the module exit function, call `cdev_del()` and `unregister_chrdev_region()`	Kernel
System administration - Load the character driver module - Create device files with matching major and minor numbers if needed The device file is ready to use!	User-space
System user - Open the device file, read, write, or send ioctl's to it.	User-space
Kernel - Executes the corresponding file operations	Kernel

Practical lab – Character drivers

- Writing a simple character driver, to write data to the serial port.
- On your workstation, checking that transmitted data is received correctly.
- Exchanging data between userspace and kernel space.
- Practicing with the character device driver API.
- Using kernel standard error codes.

Driver development

Processes and scheduling

Process, thread?

- Confusion about the terms «process», «thread» and «task»
- In Unix, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - One thread, that starts executing the `main()` function.
  - Upon creation, a process contains one thread
- Additional threads can be created inside an existing process, using `pthread_create()`
  - They run in the same address space as the initial thread of the process
  - They start executing a function passed as argument to `pthread_create()`

Process, thread: kernel point of view

- The kernel represents each thread running in the system by a structure of type task_struct
- From a scheduling point of view, it makes no difference between the initial thread of a process and all additional threads created dynamically using pthread_create()



Process after `fork()`



Same process after `pthread_create()`

A thread life



Execution of system calls

The execution of system calls takes place in the context of the thread requesting them.



Driver development

Sleeping

Sleeping

Sleeping is needed when a process (user space or kernel space) is waiting for data.



How to sleep (1)

Must declare a wait queue

A wait queue will be used to store the list of threads waiting for an event.

- Static queue declaration
  useful to declare as a global variable

```
DECLARE_WAIT_QUEUE_HEAD (module_queue);
```

- Or dynamic queue declaration
  useful to embed the wait queue inside another data structure

```
wait_queue_head_t queue;
init_waitqueue_head(&queue);
```

How to sleep (2)

Several ways to make a kernel process sleep

- `wait_event(queue, condition);`
  Sleeps until the task is woken up and the given C expression is true.
  Caution: can't be interrupted (can't kill the user-space process!)
- `int wait_event_killable(queue, condition);` (Since Linux 2.6.25)
  Can be interrupted, but only by a “fatal” signal (`SIGKILL`). Returns -ERESTARSYS if interrupted.
- `int wait_event_interruptible(queue, condition);`
  Can be interrupted by any signal. Returns -ERESTARTSYS if interrupted.
- `int wait_event_timeout(queue, condition, timeout);`
  Also stops sleeping when the task is woken up and the timeout expired. Returns 0 if the timeout elapsed, non-zero if the condition was met.
- `int wait_event_interruptible_timeout(queue, condition, timeout);`
  Same as above, interruptible. Returns 0 if the timeout elapsed, -ERESTARTSYS if interrupted, positive value if the condition was met

How to sleep - Example

```
ret = wait_event_interruptible
        (sonypi_device.fifo_proc_list,
         kfifo_len(sonypi_device.fifo) != 0);

if (ret)
   return ret;
```

Waking up!

Typically done by interrupt handlers when data sleeping processes are waiting for becomes available.

- `wake_up(&queue);`
  Wakes up all processes in the wait queue
- `wake_up_interruptible(&queue);`
  Wakes up all processes waiting in an interruptible sleep on the given queue

Exclusive vs. non-exclusive

- `wait_event_interruptible()` puts a task in a non-exclusive wait
  - All non-exclusive tasks are woken up by `wake_up() / wake_up_interruptible()`
- `wait_event_interruptible_exclusive()` puts a task in an exclusive wait
  - `wake_up() / wake_up_interruptible()` wakes up all non-exclusive tasks and only one exclusive task
  - `wake_up_all() / wake_up_interruptible_all()` wakes up all non-exclusive and all exclusive tasks
- Exclusive sleeps are useful to avoid waking up multiple tasks when only one will be able to “consume” the event
- Non-exclusive sleeps are useful when the event can “benefit” to multiple tasks

Sleeping and waking up - implementation

The scheduler doesn't keep evaluating the sleeping condition!

```
#define __wait_event(wq, condition)                                    \
   do {                                                               \
         DEFINE_WAIT(__wait);                                         \
                                                                      \
         for (;;) {                                                   \
               prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);   \
               if (condition)                                         \
                     break;                                           \
               schedule();                                            \
         }                                                            \
         finish_wait(&wq, &__wait);                                   \
   } while (0)
```

- `wait_event_interruptible(&queue, condition);`
  The process is put in the `TASK_INTERRUPTIBLE` state.
- `wake_up_interruptible(&queue);`
  All processes waiting in `queue` are woken up, so they get scheduled later and have the opportunity to reavalute the `condition.`

Driver development

Interrupt management

Registering an interrupt handler (1)

Defined in `include/linux/interrupt.h`

- ```
  int request_irq(                    Returns 0 if successful
      unsigned int irq,               Requested irq channel
      irq_handler_t handler,          Interrupt handler
      unsigned long irq_flags,        Option mask (see next page)
      const char * devname,           Registered name
      void *dev_id);                  Pointer to some handler data
  ```
  Cannot be NULL and must be unique for shared irqs!
- `void free_irq( unsigned int irq, void *dev_id);`
- `dev_id` cannot be NULL and must be unique for shared irqs.
  Otherwise, on a shared interrupt line,
  `free_irq` wouldn't know which handler to free.
  ```

Registering an interrupt handler (2)

`irq_flags` bit values (can be combined, none is fine too)

- `IRQF_DISABLED`
  "Quick" interrupt handler. Run with all interrupts disabled on the current cpu (instead of just the current line). For latency reasons, should only be used when needed!
- `IRQF_SHARED`
  Run with interrupts disabled only on the current irq line and on the local cpu. The interrupt channel can be shared by several devices. Requires a hardware status register telling whether an IRQ was raised or not.

Interrupt handler constraints

- No guarantee on which address space the system will be in when the interrupt occurs: can't transfer data to and from user space
- Interrupt handler execution is managed by the CPU, not by the scheduler. Handlers can't run actions that may sleep, because there is nothing to resume their execution. In particular, need to allocate memory with `GFP_ATOMIC`.
- Have to complete their job quickly enough: they shouldn't block their interrupt line for too long.

Information on installed handlers

IGEPv2 (OMAP3 ARM)
example on Linux 2.6.33

`/proc/interrupts`

```
           CPU0
  7:          2        INTC  TWL4030-PIH
 11:          0        INTC  prcm
 12:    6946564        INTC  DMA
 25:          2        INTC  OMAP DSS
 37:   50993360        INTC  gp timer
 56:        598        INTC  i2c_omap
 61:          0        INTC  i2c_omap
 72:          1        INTC  serial idle
 73:          1        INTC  serial idle
 74:         35        INTC  serial idle, serial
 77:    8792082        INTC  ehci_hcd:usb1
 83:    5421922        INTC  mmc0
 86:        126        INTC  mmc1
 92:          1        INTC  musb_hdrc
 93:          0        INTC  musb_hdrc
336:   11781580        GPIO  eth0
376:          0     twl4030  twl4030_pwrbutton
378:          2     twl4030  twl4030_usb
379:          0     twl4030  rtc0
384:          0     twl4030  mmc0
Err:          0
```

Registered name

Spurious interrupt count

Interrupt handler prototype

```
irqreturn_t foo_interrupt
        (int irq, void *dev_id)
```

Arguments

- `irq`, the IRQ number
- `dev_id`, the opaque pointer passed at `request_irq()`

Return value

- `IRQ_HANDLED`: recognized and handled interrupt
- `IRQ_NONE`: not on a device managed by the module. Useful to share interrupt channels and/or report spurious interrupts to the kernel.

The interrupt handler's job

- Acknowledge the interrupt to the device (otherwise no more interrupts will be generated, or the interrupt will keep firing over and over again)
- Read/write data from/to the device
- Wake up any waiting process waiting for the completion of this read/write operation:
  `wake_up_interruptible(&module_queue);`

Top half and bottom half processing (1)

Splitting the execution of interrupt handlers in 2 parts

- *Top half*: the interrupt handler must complete as quickly as possible. Once it acknowledged the interrupt, it just schedules the lengthy rest of the job taking care of the data, for a later execution.
- *Bottom half*: completing the rest of the interrupt handler job. Handles data, and then wakes up any waiting user process.
  Can be implemented using *tasklets* or *workqueues.*

Tasklets

- Declare the tasklet in the module source file:

```
DECLARE_TASKLET (module_tasklet,      /* name */
                 module_do_tasklet,   /* function */
                 data                 /* params */
);
```

- Schedule the tasklet in the top half part (interrupt handler):

```
tasklet_schedule(&module_tasklet);
```

- Note that a `tasklet_hi_schedule` function is available to define high priority tasklets to run before ordinary ones.
- Tasklets are executed with all interrupts enabled, but in interrupt context, so sleeping is not allowed.

Interrupt management summary

Device driver

- When the device file is first opened, register an interrupt handler for the device's interrupt channel.

Interrupt handler

- Called when an interrupt is raised.
- Acknowledge the interrupt
- If needed, schedule a tasklet taking care of handling data. Otherwise, wake up processes waiting for the data.

Tasklet

- Process the data
- Wake up processes waiting for the data

Device driver

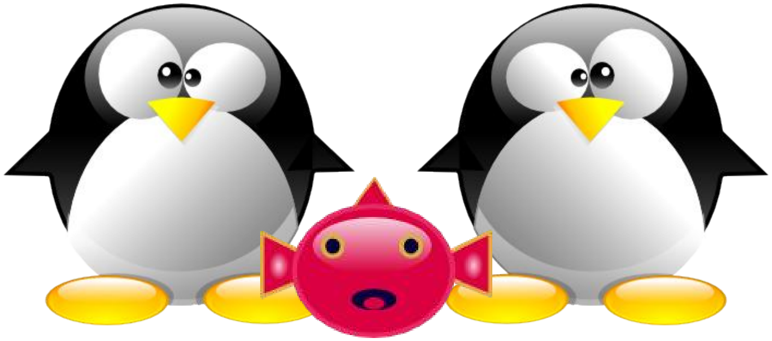
- When the device is no longer opened by any process, unregister the interrupt handler.

Practical lab – Interrupts

- Adding read capability to the character driver developed earlier.
- Register an interrupt handler.
- Waiting for data to be available in the read file operation.
- Waking up the code when data is available from the device.

Driver development

Concurrent access to resources



Sources of concurrency issues

The same resources can be accessed by several kernel processes in parallel, causing potential concurrency issues

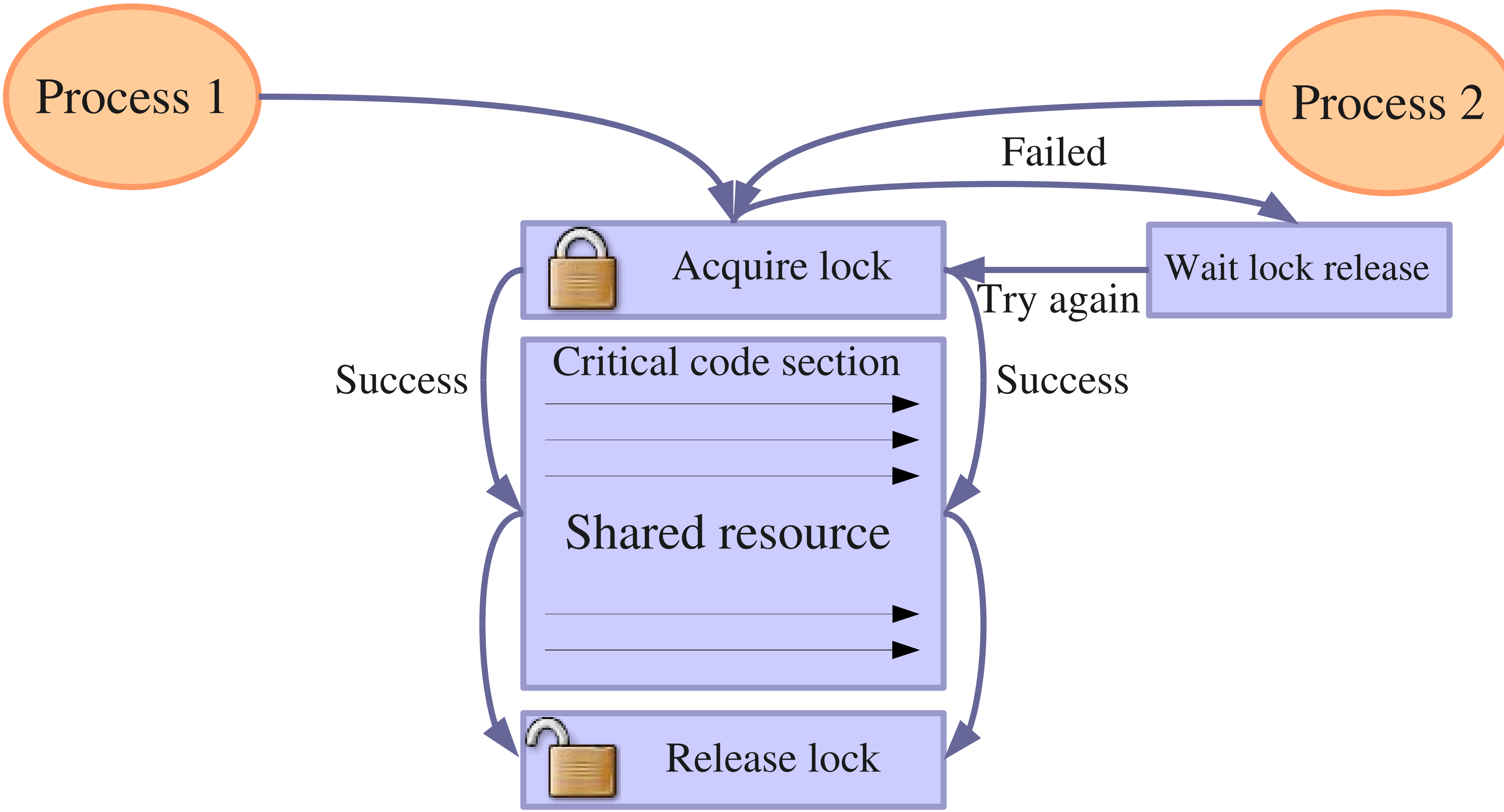
- Several user-space programs accessing the same device data or hardware. Several kernel processes could execute the same code on behalf of user processes running in parallel.
- Multiprocessing: the same driver code can be running on another processor. This can also happen with single CPUs with hyperthreading.
- Kernel preemption, interrupts: kernel code can be interrupted at any time (just a few exceptions), and the same data may be access by another process before the execution continues.

Avoiding concurrency issues

- Avoid using global variables and shared data whenever possible
(cannot be done with hardware resources).
- Use techniques to manage concurrent access to resources.

See Rusty Russell's Unreliable Guide To Locking
`Documentation/DocBook/kernel-locking/`
in the kernel sources.

Concurrency protection with locks



Linux mutexes

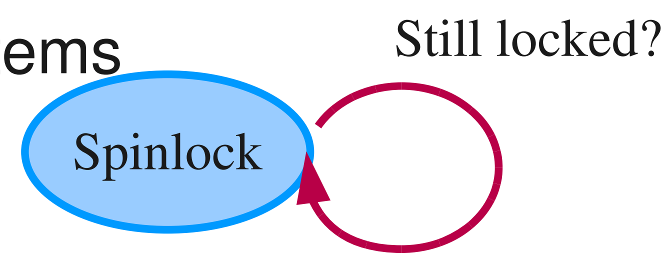
- The main locking primitive since Linux 2.6.16.
- Better than counting semaphores when binary ones are enough.
- The process requesting the lock blocks when the lock is already held. Mutexes can therefore only be used in contexts where sleeping is allowed.
- Mutex definition:
  ```
  #include <linux/mutex.h>
  ```
- Initializing a mutex statically:
  ```
  DEFINE_MUTEX(name);
  ```
- Or initializing a mutex dynamically:
  ```
  void mutex_init(struct mutex *lock);
  ```

locking and unlocking mutexes

- `void mutex_lock (struct mutex *lock);`
  Tries to lock the mutex, sleeps otherwise.
  Caution: can't be interrupted, resulting in processes you cannot kill!
- `int mutex_lock_killable (struct mutex *lock);`
  Same, but can be interrupted by a fatal (`SIGKILL`) signal. If interrupted, returns a non zero value and doesn't hold the lock. Test the return value!!!
- `int mutex_lock_interruptible (struct mutex *lock);`
  Same, but can be interrupted by any signal.
- `int mutex_trylock (struct mutex *lock);`
  Never waits. Returns a non zero value if the mutex is not available.
- `int mutex_is_locked(struct mutex *lock);`
  Just tells whether the mutex is locked or not.
- `void mutex_unlock (struct mutex *lock);`
  Releases the lock. Do it as soon as you leave the critical section.

Spinlocks

- Locks to be used for code that is not allowed to sleep (interrupt handlers), or that doesn't want to sleep (critical sections). Be very careful not to call functions which can sleep!
- Originally intended for multiprocessor systems
- Spinlocks never sleep and keep spinning in a loop until the lock is available.



- Spinlocks cause kernel preemption to be disabled on the CPU executing them.
- The critical section protected by a spinlock is not allowed to sleep.

Initializing spinlocks

- Static
  ```
  spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
  ```
- Dynamic
  ```
  void spin_lock_init (spinlock_t *lock);
  ```

Using spinlocks

Several variants, depending on where the spinlock is called:

- `void spin_[un]lock (spinlock_t *lock);`
  Doesn't disable interrupts. Used for locking in process context (critical sections in which you do not want to sleep).
- `void spin_lock_irqsave / spin_unlock_irqrestore (spinlock_t *lock, unsigned long flags);`
  Disables / restores IRQs on the local CPU.
  Typically used when the lock can be accessed in both process and interrupt context, to prevent preemption by interrupts.
- `void spin_[un]lock_bh (spinlock_t *lock);`
  Disables software interrupts, but not hardware ones.
  Useful to protect shared data accessed in process context and in a soft interrupt (“bottom half”). No need to disable hardware interrupts in this case.

Note that reader / writer spinlocks also exist.

Spinlock example

Spinlock structure embedded into uart_port

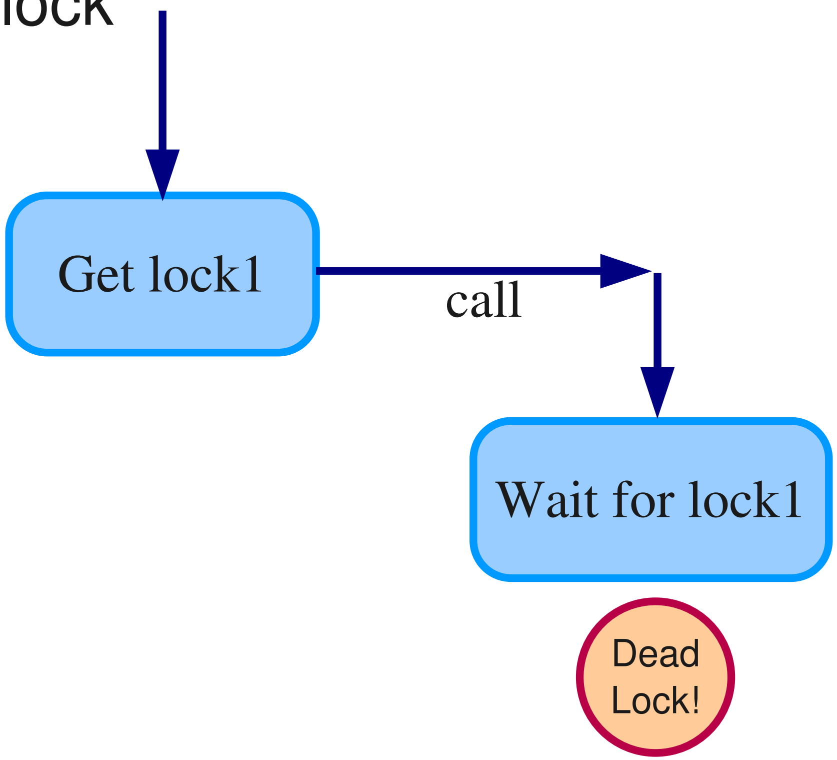
```
struct uart_port {
        spinlock_t lock;
        /* Other fields */
};
```

Spinlock taken/released with protection against interrupts

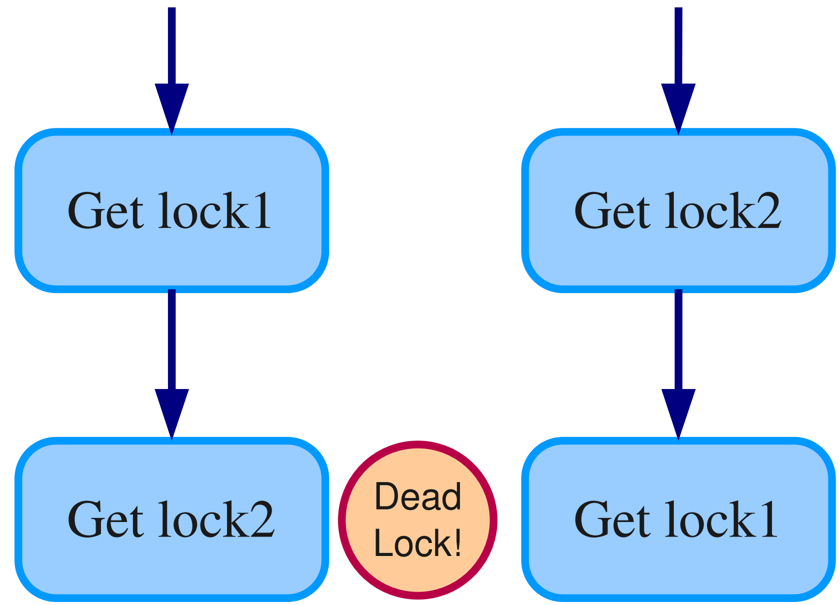
```
static unsigned int ulite_tx_empty(struct uart_port *port)
{
      unsigned long flags;

      spin_lock_irqsave(&port->lock, flags);
      /* Do something */
      spin_unlock_irqrestore(&port->lock, flags);

      […]
}
```

They can lock up your system. Make sure they never happen!

Don't call a function that can try to get access to the same lock



Holding multiple locks is risky!



Kernel lock validator

From Ingo Molnar and Arjan van de Ven

- Adds instrumentation to kernel locking code
- Detect violations of locking rules during system life, such as:
  - Locks acquired in different order
    (keeps track of locking sequences and compares them).
  - Spinlocks acquired in interrupt handlers and also in process context when interrupts are enabled.
- Not suitable for production systems but acceptable overhead in development.

See Documentation/lockdep-design.txt for details

Alternatives to locking

As we have just seen, locking can have a strong negative impact on system performance. In some situations, you could do without it.

- By using lock-free algorithms like Read Copy Update (RCU). RCU API available in the kernel (See http://en.wikipedia.org/wiki/RCU).
- When available, use atomic operations.

Atomic variables

- Useful when the shared resource is an integer value
- Even an instruction like `n++` is not guaranteed to be atomic on all processors!

Header

- `#include <asm/atomic.h>`

Type

- `atomic_t`
  contains a signed integer (at least 24 bits)

Atomic operations (main ones)

- Set or read the counter:
```
atomic_set (atomic_t *v, int i);
int atomic_read (atomic_t *v);
```
- Operations without return value:
```
void atomic_inc (atomic_t *v);
void atomic_dec (atomic_t *v);
void atomic_add (int i, atomic_t *v);
void atomic_sub (int i, atomic_t *v);
```
- Simular functions testing the result:
```
int atomic_inc_and_test (...);
int atomic_dec_and_test (...);
int atomic_sub_and_test (...);
```
- Functions returning the new value:
```
int atomic_inc_and_return (...);
int atomic_dec_and_return (...);
int atomic_add_and_return (...);
int atomic_sub_and_return (...);
```

Atomic bit operations

- Supply very fast, atomic operations
- On most platforms, apply to an `unsigned long` type.
  Apply to a `void` type on a few others.
- Set, clear, toggle a given bit:

```
void set_bit(int nr, unsigned long * addr);
void clear_bit(int nr, unsigned long * addr);
void change_bit(int nr, unsigned long * addr);
```

- Test bit value:

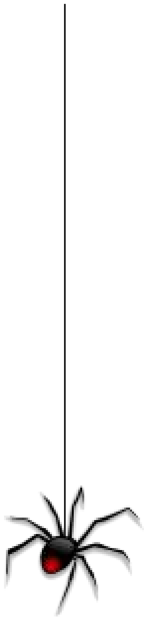
```
int test_bit(int nr, unsigned long *addr);
```

- Test and modify (return the previous value):

```
int test_and_set_bit (...);
int test_and_clear_bit (...);
int test_and_change_bit (...);
```

Practical lab – Locking

- Add locking to the driver to prevent concurrent accesses to shared ressources



Driver development

Debugging and tracing

Debugging with printk

- Universal debugging technique used since the beginning of programming (first found in cavemen drawings)
- Printed or not in the console according to the priority.
  This is controlled by the `loglevel` kernel parameter, or through `/proc/sys/kernel/printk`
  (see `Documentation/sysctl/kernel.txt`)
- Available priorities (`include/linux/kernel.h`):

```
#define KERN_EMERG      "<0>"   /* system is unusable */
#define KERN_ALERT      "<1>"   /* action must be taken immediately */
#define KERN_CRIT       "<2>"   /* critical conditions */
#define KERN_ERR        "<3>"   /* error conditions */
#define KERN_WARNING    "<4>"   /* warning conditions */
#define KERN_NOTICE     "<5>"   /* normal but significant condition */
#define KERN_INFO       "<6>"   /* informational */
#define KERN_DEBUG      "<7>"   /* debug-level messages */
```

Debugging with /proc or /sys

Instead of dumping messages in the kernel log, you can have your drivers make information available to user space

- Through a file in `/proc` or `/sys`, which contents are handled by callbacks defined and registered by your driver.
- Can be used to show any piece of information about your device or driver.
- Can also be used to send data to the driver or to control it.
- Caution: anybody can use these files. You should remove your debugging interface in production!
- Since the arrival of *debugfs*, no longer the preferred debugging mechanism

Debugfs

A virtual filesystem to export debugging information to user-space.

- Kernel configuration: `DEBUG_FS`
  `Kernel hacking -> Debug Filesystem`
- Much simpler to code than an interface in `/proc` or `/sys`.
  The debugging interface disappears when Debugfs is configured out.
- You can mount it as follows:
  `sudo mount -t debugfs none /mnt/debugfs`
- First described on http://lwn.net/Articles/115405/
- API documented in the Linux Kernel Filesystem API:
  http://free-electrons.com/kerneldoc/latest/DocBook/filesystems/index.html

Simple debugfs example

```
#include <linux/debugfs.h>

static char *acme_buf;                                    // module buffer
static unsigned long acme_bufsize;
static struct debugfs_blob_wrapper acme_blob;
static struct dentry *acme_buf_dentry;

static u32 acme_state;                                    // module variable
static struct dentry *acme_state_dentry;

/* Module init */
acme_blob.data = acme_buf;
acme_blob.size = acme_bufsize;
acme_buf_dentry = debugfs_create_blob("acme_buf", S_IRUGO,      // Create
                                      NULL, &acme_blob);        // new files
acme_state_dentry = debugfs_create_bool("acme_state", S_IRUGO,  // in debugfs
                                        NULL, &acme_state);

/* Module exit */
debugfs_remove (acme_buf_dentry);                         // removing the files from debugfs
debugfs_remove (acme_state_dentry);
```

Debugging with ioctl

- Can use the `ioctl()` system call to query information about your driver (or device) or send commands to it.
- This calls the `unlocked_ioctl` file operation that you can register in your driver.
- Advantage: your debugging interface is not public.
  You could even leave it when your system (or its driver) is in the hands of its users.

Using Magic SysRq

- Allows to run multiple debug / rescue commands even when the kernel seems to be in deep trouble
  - On PC: Alt + SysRq + <character>
  - On embedded: break character on the serial line + <character>
- . Example commands:
  - `n`: makes RT processes nice-able.
  - `t`: shows the kernel stack of all sleeping processes
  - `w`: shows the kernel stack of all running processes
  - `b`: reboot the system
  - You can even register your own!
- Detailed in `Documentation/sysrq.txt`

kgdb - A kernel debugger

- The execution of the kernel is fully controlled by `gdb` from another machine, connected through a serial line.
- Can do almost everything, including inserting breakpoints in interrupt handlers.
- Feature included in standard Linux since 2.6.26 (`x86` and `sparc`). `arm`, `mips` and `ppc` support merged in 2.6.27.

Using kgdb

- Details available in the kernel documentation: http://free-electrons.com/kerneldoc/latest/DocBook/kgdb/
- Recommended to turn on `CONFIG_FRAME_POINTER` to aid in producing more reliable stack backtraces in gdb.
- You must include a kgdb I/O driver. One of them is kgdb over serial console (`kgdboc`: *kgdb over console*, enabled by `CONFIG_KGDB_SERIAL_CONSOLE`)
- Configure `kgdboc` at boot time by passing to the kernel: `kgdboc=<tty-device>,[baud]`. For example: `kgdboc=ttyS0,115200`

Using kgdb (2)

- Then also pass `kgdbwait` to the kernel:
  it makes `kgdb` wait for a debugger connection.
- Boot your kernel, and when the console is initialized, interrupt the kernel with `[Alt][SyrRq][g]`.
- On your workstation, start gdb as follows:
  ```
  % gdb ./vmlinux
  (gdb) set remotebaud 115200
  (gdb) target remote /dev/ttyS0
  ```
- Once connected, you can debug a kernel the way you would debug an application program.

Debugging with a JTAG interface

- Two types of JTAG dongles
  - Those offering a gdb compatible interface, over a serial port or an Ethernet connexion. Gdb can directly connect to them.
  - Those not offering a gdb compatible interface are generally supported by OpenOCD (Open On Chip Debugger)
    - OpenOCD is the bridge between the gdb debugging language and the JTAG-dongle specific language
    - http://openocd.berlios.de/web/
    - See the very complete documentation: http://openocd.berlios.de/doc/
    - For each board, you'll need an OpenOCD configuration file (ask your supplier)
- See very useful details on using Eclipse / gcc / gdb / OpenOCD on Windows: http://www2.amontec.com/sdk4arm/ext/jlynch-tutorial-20061124.pdf and http://www.yagarto.de/howto/yagarto2/

More kernel debugging tips

- Enable `CONFIG_KALLSYMS_ALL`
  (`General Setup` -> `Configure standard kernel features`)
  to get oops messages with symbol names instead of raw addresses
  (this obsoletes the ksymoops tool).
- If your kernel doesn't boot yet or hangs without any message, you can activate Low Level debugging (Kernel Hacking section, only available on `arm`):
  `CONFIG_DEBUG_LL=y`
- Techniques to locate the C instruction which caused an oops:
  http://kerneltrap.org/node/3648
- More about kernel debugging in the free Linux Device Drivers book:
  http://lwn.net/images/pdf/LDD3/ch04.pdf

Tracing with SystemTap

http://sourceware.org/systemtap/

- Infrastructure to add instrumentation to a running kernel:
  trace functions, read and write variables, follow pointers, gather statistics...
- Eliminates the need to modify the kernel sources to add one's own instrumentation to investigated a functional or performance problem.
- Uses a simple scripting language.
  Several example scripts and probe points are available.
- Based on the Kprobes instrumentation infrastructure.
  See `Documentation/kprobes.txt` in kernel sources.
  Linux 2.6.26: supported on most popular CPUs (`arm` included in 2.6.25).
  However, lack of recent support for `mips` (2.6.16 only!).

SystemTap script example (1)

```
#! /usr/bin/env stap
# Using statistics and maps to examine kernel memory allocations

global kmalloc

probe kernel.function("__kmalloc") {
    kmalloc[execname()] <<< $size
}

# Exit after 10 seconds
probe timer.ms(10000) { exit () }

probe end {
    foreach ([name] in kmalloc) {
        printf("Allocations for %s\n", name)
        printf("Count:   %d allocations\n", @count(kmalloc[name]))
        printf("Sum:     %d Kbytes\n", @sum(kmalloc[name])/1024)
        printf("Average: %d bytes\n", @avg(kmalloc[name]))
        printf("Min:     %d bytes\n", @min(kmalloc[name]))
        printf("Max:     %d bytes\n", @max(kmalloc[name]))
        print("\nAllocations by size in bytes\n")
        print(@hist_log(kmalloc[name]))
        printf("-----------------------------------------------\n\n");
    }
}
```

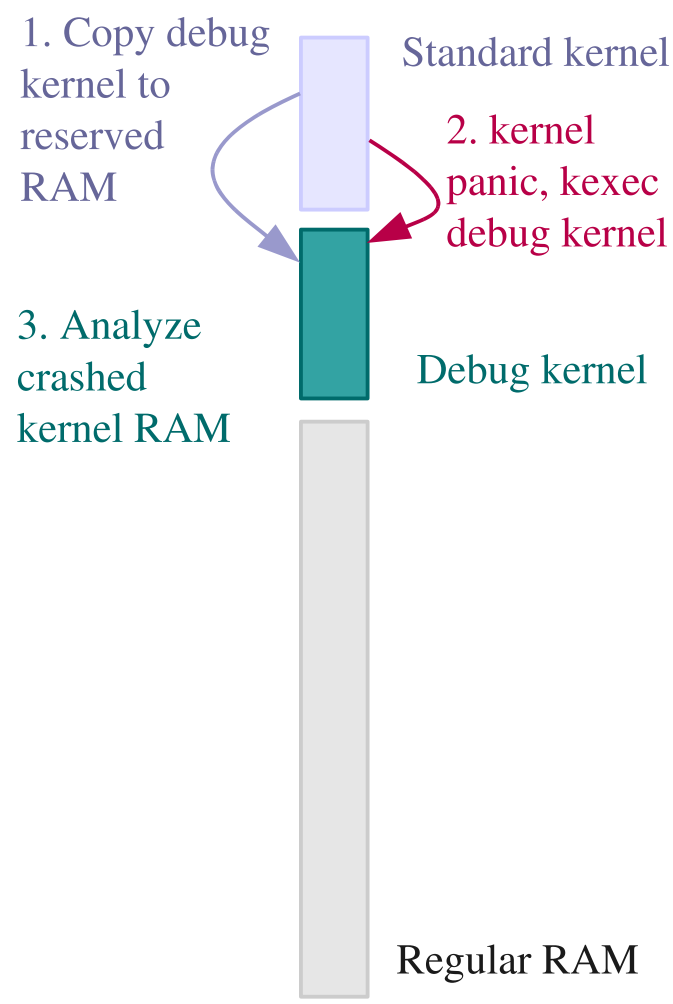
SystemTap script example (2)

```
#! /usr/bin/env stap

# Logs each file read performed by each process

probe kernel.function ("vfs_read")
{
  dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
  inode_nr = $file->f_dentry->d_inode->i_ino
  printf ("%s(%d) %s 0x%x/%d\n",
      execname(), pid(), probefunc(), dev_nr, inode_nr)
}
```

Nice tutorial on http://sources.redhat.com/systemtap/tutorial.pdf

Kernel crash analysis with kexec/kdump

- `kexec` system call: makes it possible to call a new kernel, without rebooting and going through the BIOS / firmware.
- Idea: after a kernel panic, make the kernel automatically execute a new, clean kernel from a reserved location in RAM, to perform post-mortem analysis of the memory of the crashed kernel.
- See Documentation/kdump/kdump.txt in the kernel sources for details.



Kernel markers

- Capability to add static markers to kernel code, merged in Linux 2.6.24 by Matthieu Desnoyers.
- Almost no impact on performance, until the marker is dynamically enabled, by inserting a probe kernel module.
- Useful to insert trace points that won't be impacted by changes in the Linux kernel sources.
- See marker and probe example in `samples/markers` in the kernel sources.

See http://en.wikipedia.org/wiki/Kernel_marker

LTTng

http://lttng.org

- The successor of the Linux Trace Toolkit (LTT)
- Toolkit allowing to collect and analyze tracing information from the kernel, based on kernel markers and kernel tracepoints.
- So far, based on kernel patches, but doing its best to use in-tree solutions, and to be merged in the future.
- Very precise timestamps, very little overhead.
- Useful documentation on http://lttng.org/?q=node/2#manuals

LTTV

Viewer for LTTng traces

- Support for huge traces (tested with 15 GB ones)
- Can combine multiple tracefiles in a single view.
- Graphical or text interface

See http://lttng.org/files/lttv-doc/user_guide/

Practical lab – Kernel debugging

- Load a broken driver and see it crash
- Analyze the error information dumped by the kernel.
- Disassemble the code and locate the exact C instruction which caused the failure.
- Use the JTAG and OpenOCD to remotely control the kernel execution

Driver development
mmap

mmap (1)

Possibility to have parts of the virtual address space of a program mapped to the contents of a file!

`> cat /proc/1/maps` (init process)

```
start             end               perm  offset    major:minor inode       mapped file name
00771000-0077f000 r-xp 00000000 03:05 1165839    /lib/libselinux.so.1
0077f000-00781000 rw-p 0000d000 03:05 1165839    /lib/libselinux.so.1
0097d000-00992000 r-xp 00000000 03:05 1158767    /lib/ld-2.3.3.so
00992000-00993000 r--p 00014000 03:05 1158767    /lib/ld-2.3.3.so
00993000-00994000 rw-p 00015000 03:05 1158767    /lib/ld-2.3.3.so
00996000-00aac000 r-xp 00000000 03:05 1158770    /lib/tls/libc-2.3.3.so
00aac000-00aad000 r--p 00116000 03:05 1158770    /lib/tls/libc-2.3.3.so
00aad000-00ab0000 rw-p 00117000 03:05 1158770    /lib/tls/libc-2.3.3.so
00ab0000-00ab2000 rw-p 00ab0000 00:00 0
08048000-08050000 r-xp 00000000 03:05 571452     /sbin/init (text)
08050000-08051000 rw-p 00008000 03:05 571452     /sbin/init (data, stack)
08b43000-08b64000 rw-p 08b43000 00:00 0
f6fdf000-f6fe0000 rw-p f6fdf000 00:00 0
fefd4000-ff000000 rw-p fefd4000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

mmap (2)

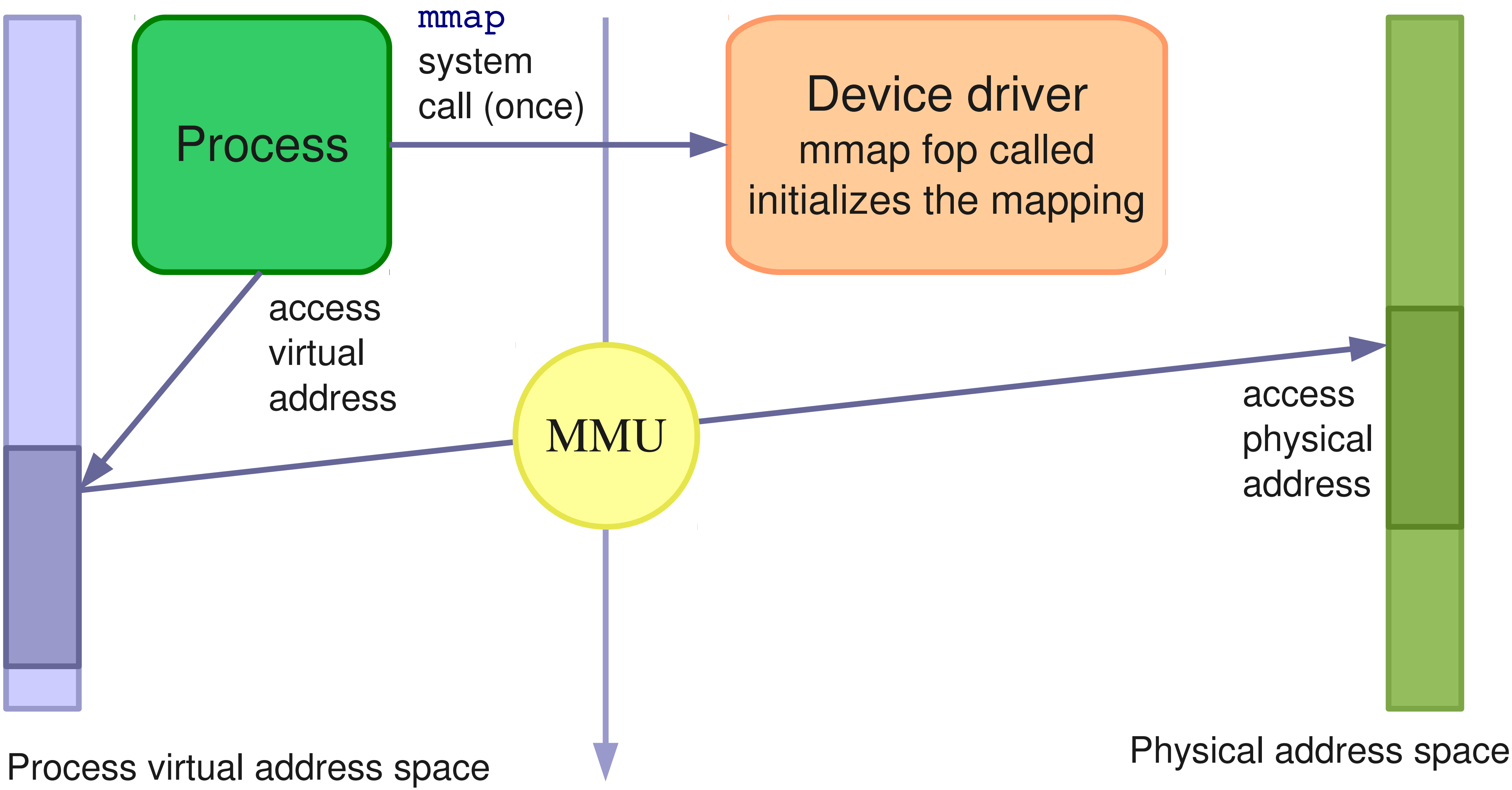
Particularly useful when the file is a device file!
Allows to access device I/O memory and ports without having to go through (expensive) `read`, `write` or `ioctl` calls!

X server example (maps excerpt)

```
start             end               perm  offset    major:minor inode       mapped file name
08047000-081be000 r-xp 00000000 03:05 310295     /usr/X11R6/bin/Xorg
081be000-081f0000 rw-p 00176000 03:05 310295     /usr/X11R6/bin/Xorg
...
f4e08000-f4f09000 rw-s e0000000 03:05 655295     /dev/dri/card0
f4f09000-f4f0b000 rw-s 4281a000 03:05 655295     /dev/dri/card0
f4f0b000-f6f0b000 rw-s e8000000 03:05 652822     /dev/mem
f6f0b000-f6f8b000 rw-s fcff0000 03:05 652822     /dev/mem
```

A more user friendly way to get such information: `pmap <pid>`

mmap overview



How to implement mmap - User space

- Open the device file
- Call the `mmap` system call (see `man mmap` for details):

```
void * mmap(
   void *start,     /* Often 0, preferred starting address */
   size_t length,   /* Length of the mapped area */
   int prot ,       /* Permissions: read, write, execute */
   int flags,       /* Options: shared mapping, private copy...
*/
   int fd,          /* Open file descriptor */
   off_t offset     /* Offset in the file */
);
```

- You get a virtual address you can write to or read from.

How to implement mmap - Kernel space

- Character driver: implement a `mmap` file operation and add it to the driver file operations:

```
int (*mmap) (
   struct file *,                /* Open file structure */
   struct vm_area_struct *       /* Kernel VMA structure */
);
```

- Initialize the mapping.
  Can be done in most cases with the `remap_pfn_range()` function, which takes care of most of the job.

remap_pfn_range()

- *pfn*: page frame number
  The most significant bits of the page address
  (without the bits corresponding to the page size).
- `#include <linux/mm.h>`

```
int remap_pfn_range(
    struct vm_area_struct *,              /* VMA struct */
    unsigned long virt_addr,    /* Starting user virtual address */
    unsigned long pfn,    /* pfn of the starting physical address */
    unsigned long size,                   /* Mapping size */
    pgprot_t                              /* Page permissions */
);
```

Simple mmap implementation

```
static int acme_mmap (
   struct file * file, struct vm_area_struct * vma)
{
   size = vma->vm_end - vma->vm_start;

   if (size > ACME_SIZE)
      return -EINVAL;

   if (remap_pfn_range(vma,
               vma->vm_start,
               ACME_PHYS >> PAGE_SHIFT,
               size,
               vma->vm_page_prot))
      return -EAGAIN;
   return 0;
}
```

devmem2

http://free-electrons.com/pub/mirror/devmem2.c, by Jan-Derk Bakker

Very useful tool to directly peek (read) or poke (write) I/O addresses mapped in physical address space from a shell command line!

- Very useful for early interaction experiments with a device, without having to code and compile a driver.
- Uses `mmap` to `/dev/mem`.
- Examples (`b`: byte, `h`: half, `w`: word)
  `devmem2 0x000c0004 h` (reading)
  `devmem2 0x000c0008 w 0xffffffff` (writing)
- `devmem` is now available in BusyBox, making it even easier to use.

mmap summary

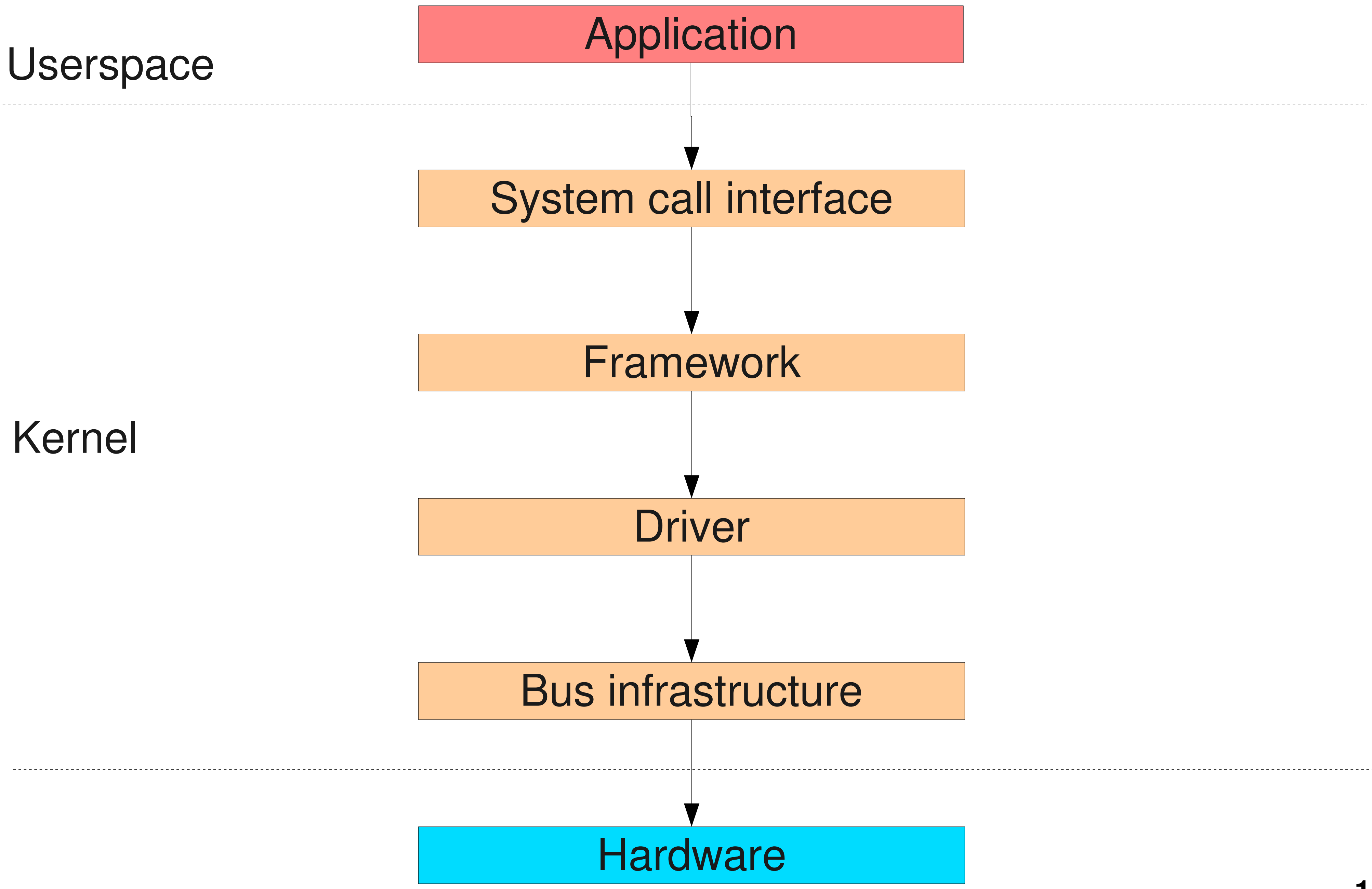
- The device driver is loaded.
  It defines an `mmap` file operation.
- A user space process calls the `mmap` system call.
- The `mmap` file operation is called.
  It initializes the mapping using the device physical address.
- The process gets a starting address to read from and write to (depending on permissions).
- The MMU automatically takes care of converting the process virtual addresses into physical ones.

Direct access to the hardware!
No expensive `read` or `write` system calls!

Driver development

Kernel architecture for device drivers

Kernel and device drivers

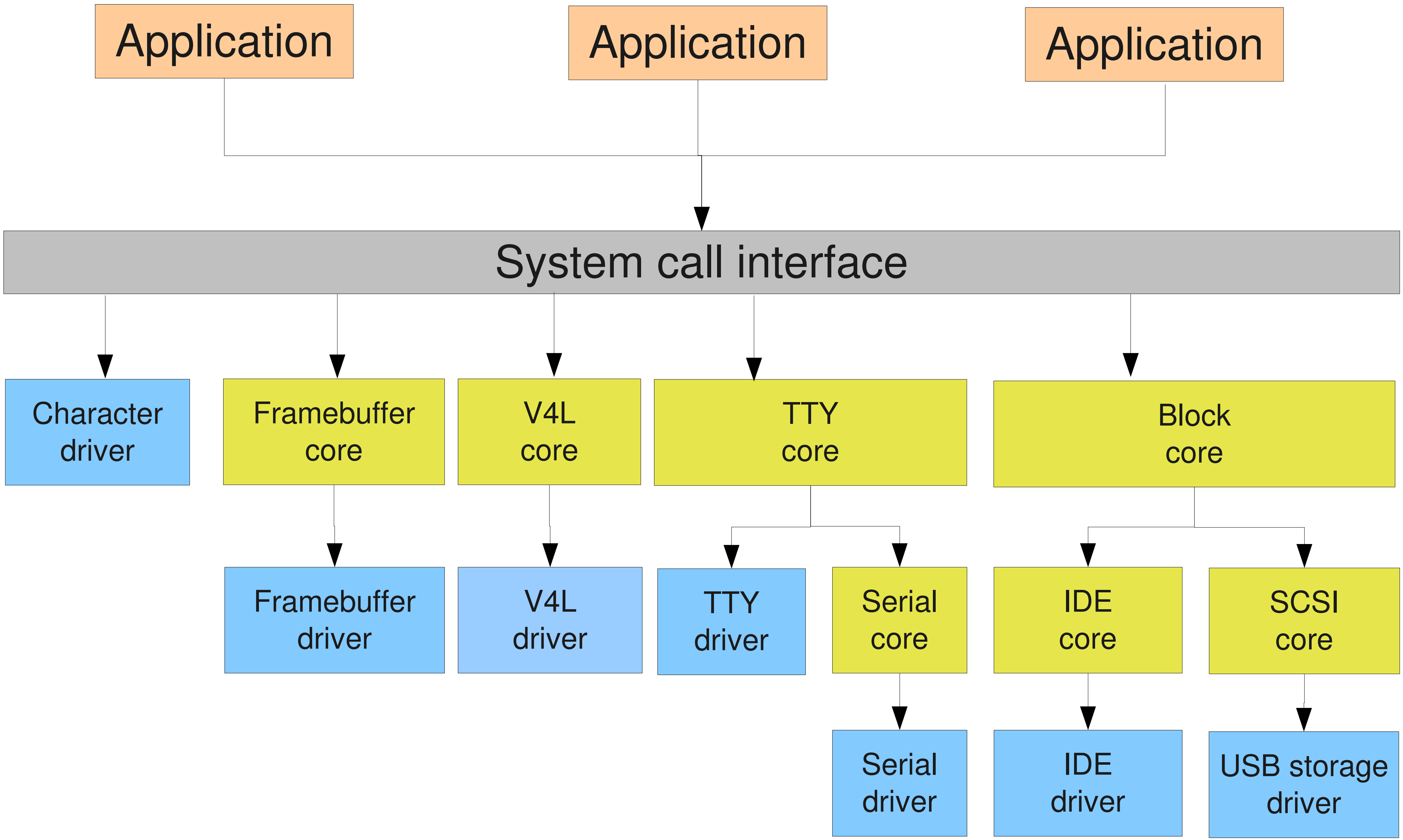


Kernel and device drivers

- Many device drivers are not implemented directly as character drivers
- They are implemented under a « framework », specific to a given device type (framebuffer, 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 character devices by the applications
  - The framework allows to provide a coherent userspace interface (ioctl, etc.) for every type of device, regardless of the driver
- The device drivers rely on the « bus infrastructure » to enumerate the devices and communicate with them.

Kernel frameworks

« Frameworks »


Application
Application
Application
System call interface
Character driver
Framebuffer core
V4L core
TTY core
Block core
Framebuffer driver
V4L driver
TTY driver
Serial core
IDE core
SCSI core
Serial driver
IDE driver
USB storage driver

Example: framebuffer framework

- Kernel option `CONFIG_FB`

```
menuconfig FB
        tristate "Support for frame buffer devices"
```

- Implemented in `drivers/video/`
  - `fb.c, fbmem.c, fbmon.c, fbcmap.c, fbsysfs.c, modedb.c, fbcvt.c`
- Implements a single character driver and defines 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` (in `#ifdef __KERNEL__`)

Framebuffer driver skeleton

- Skeleton driver in `drivers/video/skeletonfb.c`
- Implements the set of framebuffer specific operations defined by the `struct fb_ops` structure
  - `xxxfb_open()`
  - `xxxfb_read()`
  - `xxxfb_write()`
  - `xxxfb_release()`
  - `xxxfb_checkvar()`
  - `xxxfb_setpar()`
  - `xxxfb_setcolreg()`
  - `xxxfb_blank()`
  - `xxxfb_pan_display()`
  - `xxxfb_fillrect()`
  - `xxxfb_copyarea()`
  - `xxxfb_imageblit()`
  - `xxxfb_cursor()`
  - `xxxfb_rotate()`
  - `xxxfb_sync()`
  - `xxxfb_ioctl()`
  - `xxxfb_mmap()`

Framebuffer driver skeleton

- After the implementation of the operations, definition of a `struct fb_ops` structure

```
static struct fb_ops xxxfb_ops = {
        .owner          = THIS_MODULE,
        .fb_open        = xxxfb_open,
        .fb_read        = xxxfb_read,
        .fb_write       = xxxfb_write,
        .fb_release     = xxxfb_release,
        .fb_check_var   = xxxfb_check_var,
        .fb_set_par     = xxxfb_set_par,
        .fb_setcolreg   = xxxfb_setcolreg,
        .fb_blank       = xxxfb_blank,
        .fb_pan_display = xxxfb_pan_display,
        .fb_fillrect    = xxxfb_fillrect,       /* Needed !!! */
        .fb_copyarea    = xxxfb_copyarea,       /* Needed !!! */
        .fb_imageblit   = xxxfb_imageblit,      /* Needed !!! */
        .fb_cursor      = xxxfb_cursor,         /* Optional !!! */
        .fb_rotate      = xxxfb_rotate,
        .fb_sync        = xxxfb_sync,
        .fb_ioctl       = xxxfb_ioctl,
        .fb_mmap        = xxxfb_mmap,
};
```

Framebuffer driver skeleton

- In the `probe()` function, registration of the framebuffer device and operations

```
static int __devinit xxxfb_probe
     (struct pci_dev *dev,
      const struct pci_device_id *ent)
{
     struct fb_info *info;
     [...]
     info = framebuffer_alloc(sizeof(struct xxx_par), device);
     [...]
     info->fbops = &xxxfb_ops;
     [...]
     if (register_framebuffer(info) < 0)
         return -EINVAL;
     [...]
}
```

- `register_framebuffer()` will create the character device that can be used by userspace application with the generic framebuffer API

Device Model and Bus Infrastructure

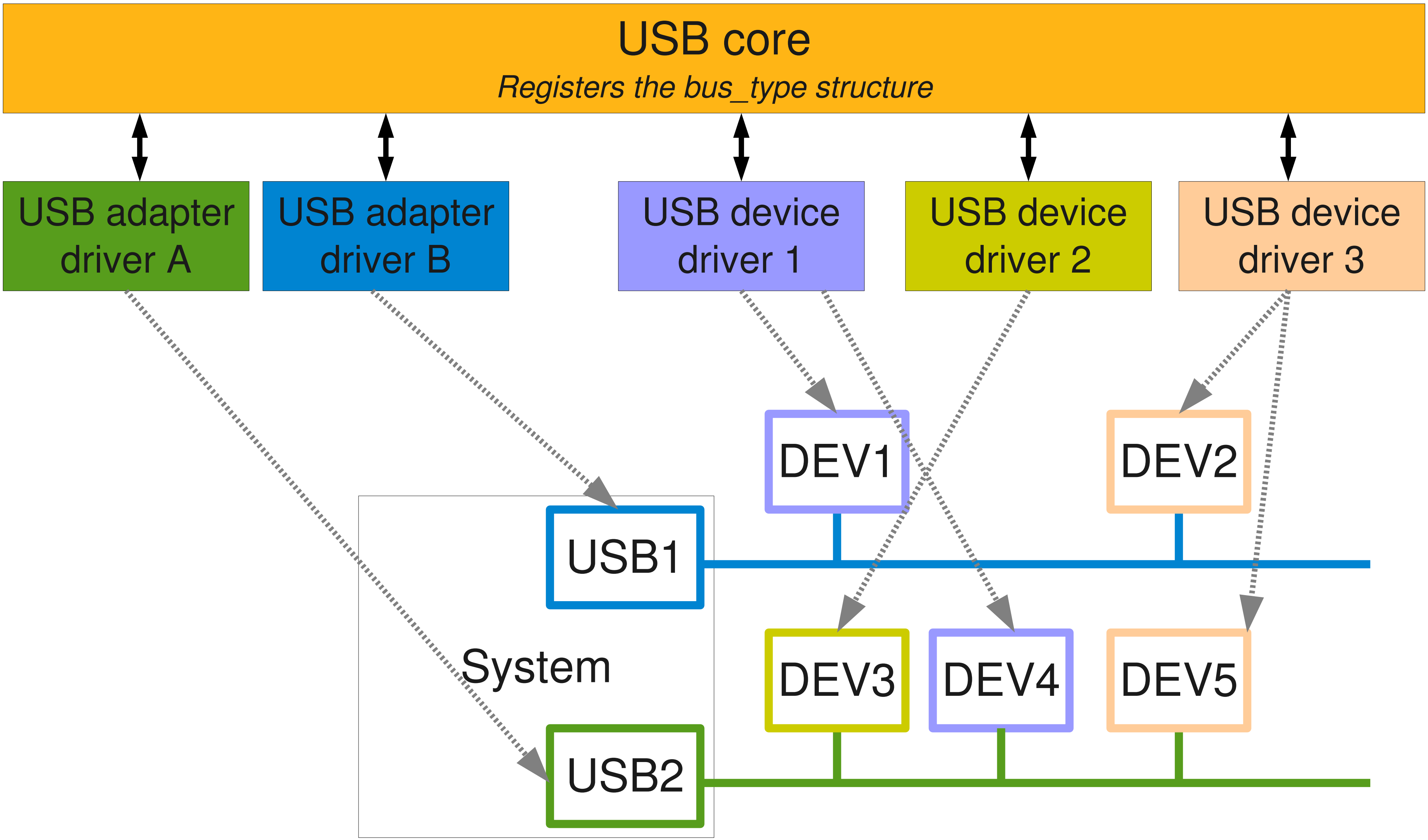
Unified device model

- Th 2.6 kernel included a significant new feature: a unified device model
- Instead of having different ad-hoc mechanisms in the various subsystems, the device model unifies the description of the devices and their topology
  - Minimization of code duplication
  - Common facilities (reference counting, event notification, power management, etc.)
  - Enumerate the devices view their interconnections, link the devices to their buses and drivers, etc.
- Understand the device model is necessary to understand how device drivers fit into the Linux kernel architecture.

Bus drivers

- The first component of the device model is the bus driver
- **One** bus driver for each **type** of bus: USB, PCI, SPI, MMC, I2C, etc.
- It is responsible for
  - Registering the bus type (struct `bus_type`)
  - Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able of detecting the connected devices, and providing a communication mechanism with the devices
  - Allowing the registration of device drivers (USB devices, I2C devices, PCI devices, etc.), managing the devices
  - Matching the device drivers against the devices detected by the adapter drivers.
  - Provides an API to both adapter drivers and device drivers
  - Defining driver and device specific structure, typically `xxx_driver` and `xxx_device`

Example: USB bus



Example: USB bus (2)

- Core infrastructure (bus driver)
  - `drivers/usb/core`
  - The bus_type is defined in `drivers/usb/core/driver.c` and registered in `drivers/usb/core/usb.c`
- Adapter drivers
  - `drivers/usb/host`
  - For EHCI, UHCI, OHCI, XHCI, and their implementations on various systems (Atmel, IXP, Xilinx, OMAP, Samsung, PXA, etc.)
- Device drivers
  - Everywhere in the kernel tree, classified by their type

Example of device driver

- To illustrate how drivers are implemented to work with the device model, we will study the source code of a driver for a USB network card
  - It is USB device, so it has to be a USB device driver
  - It is a network device, so it has to be a network device
  - Most drivers rely on a bus infrastructure (here, USB) and register themselves in a framework (here, network)
- We will only look at the device driver side, and not the adapter driver side
- The driver we will look at is `drivers/net/usb/rtl8150.c`

Device identifiers

- Defines the set of devices that this driver can manage, so that the USB core knows for which devices this driver should be used
- The MODULE_DEVICE_TABLE macro allows depmod to extract at compile time the relation between device identifiers and drivers, so that drivers can be loaded automatically by udev. See /lib/modules/$(uname -r)/modules.{alias,usbmap}

```
static struct usb_device_id rtl8150_table[] = {
    { USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150)     },
    { USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX)        },
    { USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR)    },
    { USB_DEVICE(VENDOR_ID_LONGSHINE, PRODUCT_ID_LCS8138TX) },
    { USB_DEVICE(VENDOR_ID_OQO, PRODUCT_ID_RTL8150)         },
    { USB_DEVICE(VENDOR_ID_ZYXEL, PRODUCT_ID_PRESTIGE)      },
    {}
};

MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Instanciation of usb_driver

- struct usb_driver is a structure defined by the USB core. Each USB device driver must instantiate it, and register itself to the USB core using this structure
- This structure inherits from struct driver, which is defined by the device model.

```
static struct usb_driver rtl8150_driver = {
    .name           = "rtl8150",
    .probe          = rtl8150_probe,
    .disconnect     = rtl8150_disconnect,
    .id_table       = rtl8150_table,
    .suspend        = rtl8150_suspend,
    .resume         = rtl8150_resume
};
```

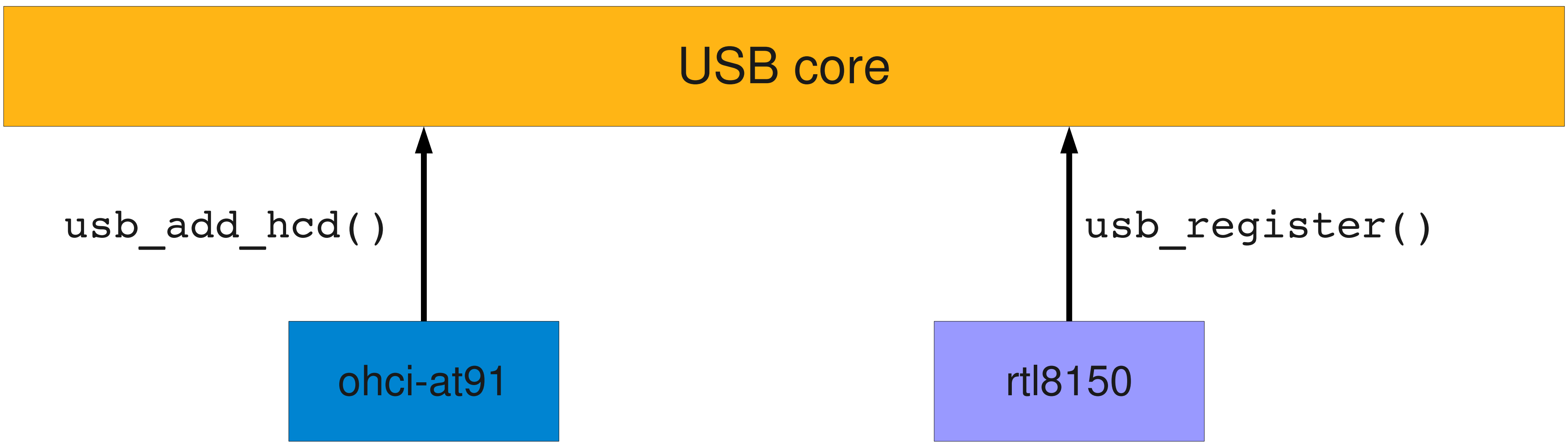
Driver (un)registration

- When the driver is loaded or unloaded, it must register or unregister itself from the USB core
- Done using `usb_register()` and `usb_deregister()`, provided by the USB core.

```
static int __init usb_rtl8150_init(void)
{
    return usb_register(&rtl8150_driver);
}
static void __exit usb_rtl8150_exit(void)
{
    usb_deregister(&rtl8150_driver);
}

module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

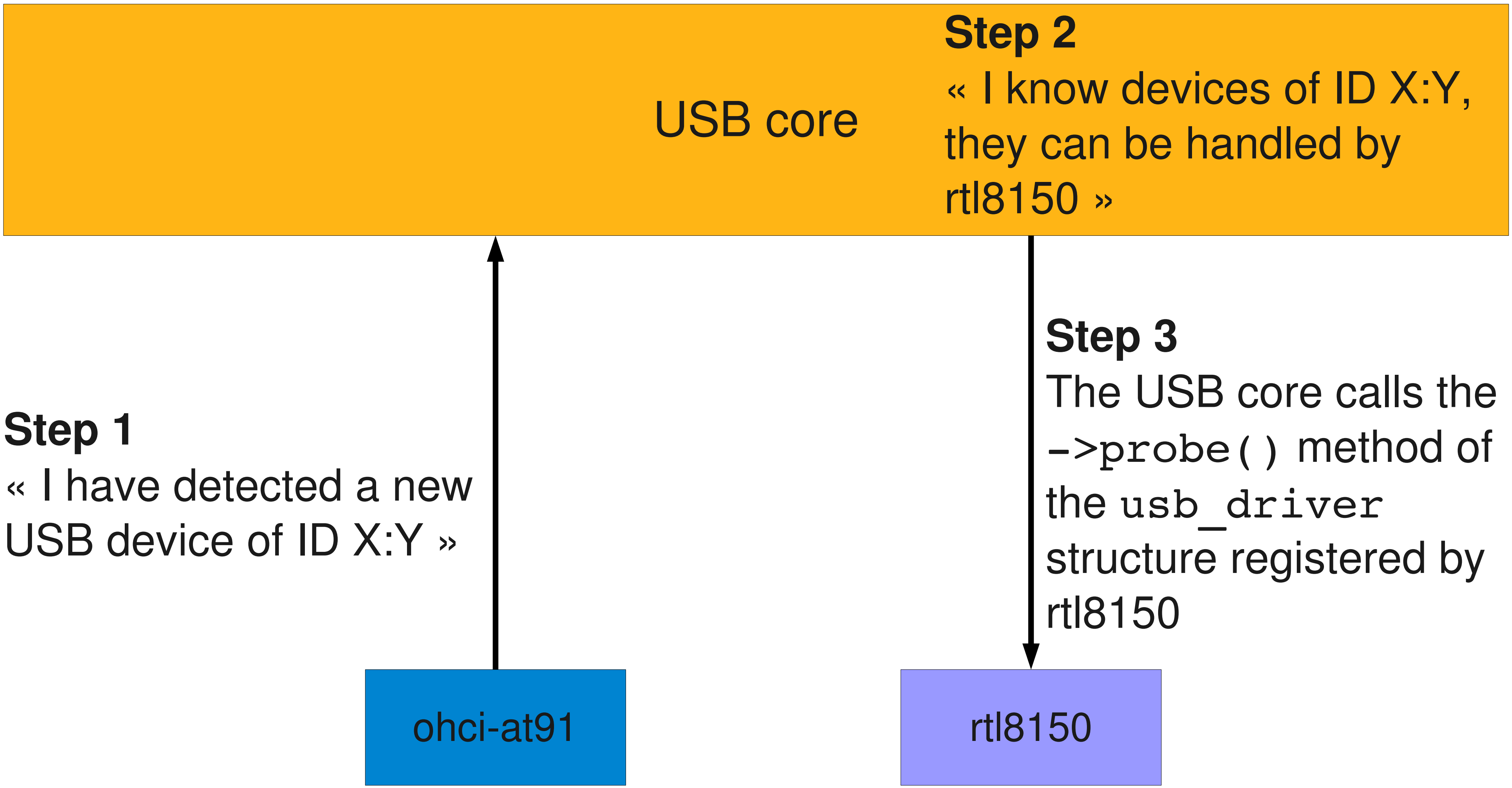
At initialization

- The USB adapter driver that corresponds to the USB controller of the system registers itself to the USB core
- The rtl8150 USB device driver registers itself to the USB core



- The USB core now knows the association between the vendor/product IDs of rtl8150 and the `usb_driver` structure of this driver

When a device is detected



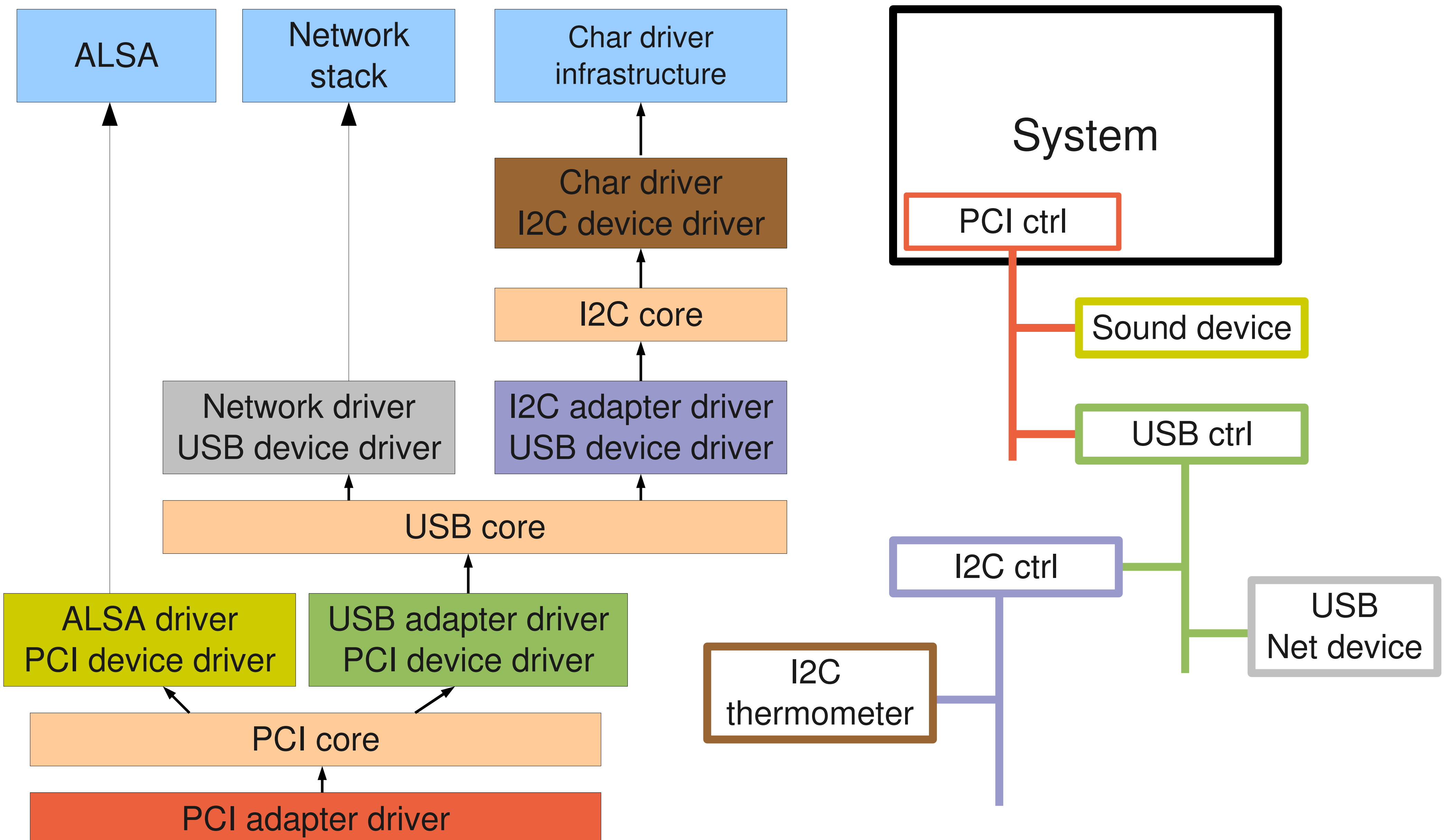
Probe method

- The `probe()` method receives as argument a structure describing the device, usually specialized by the bus infrastructure (`pci_dev`, `usb_interface`, etc.)
- This function is responsible for
  - Initializing the device, mapping I/O memory, registering the interrupt handlers. The bus infrastructure provides methods to get the addresses, interrupts numbers and other device-specific information.
  - Registering the device to the proper kernel framework, for example the network infrastructure.

Probe method example

```
static int rtl8150_probe( struct usb_interface *intf,
                          const struct usb_device_id *id)
{
   rtl8150_t *dev;
   struct net_device *netdev;

   netdev = alloc_etherdev(sizeof(rtl8150_t));
   [...]
   dev = netdev_priv(netdev);
   tasklet_init(&dev->tl, rx_fixup, (unsigned long)dev);
   spin_lock_init(&dev->rx_pool_lock);
   [...]
   netdev->netdev_ops = &rtl8150_netdev_ops;
   alloc_all_urbs(dev);
   [...]
   usb_set_intfdata(intf, dev);
   SET_NETDEV_DEV(netdev, &intf->dev);
   register_netdev(netdev);

   return 0;
}
```

The model is recursive



sysfs

- The bus, device, drivers, etc. structures are internal to the kernel
- The `sysfs` virtual filesystem offers a mechanism to export such information to userspace
- Used for example by `udev` to provide automatic module loading, firmware loading, device file creation, etc.
- `sysfs` is usually mounted in `/sys`
  - `/sys/bus/` contains the list of buses
  - `/sys/devices/` contains the list of devices
  - `/sys/class` enumerates devices by class (`net`, `input`, `block`...), whatever the bus they are connected to. Very useful!
- Take your time to explore `/sys` on your workstation.

Platform devices

- 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.

Implementation of the platform driver

- The driver implements a platform_driver structure (example taken from `drivers/serial/imx.c`)

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

- And registers its driver to the platform driver infrastructure

```
static int __init imx_serial_init(void)
{
        ret = platform_driver_register(&serial_imx_driver);
}
static void __exit imx_serial_cleanup(void)
{
        platform_driver_unregister(&serial_imx_driver);
}
```

Platform device instantiation (1)

- As platform devices cannot be detected dynamically, they are defined statically
  - By direct instantiation of `platform_device` structures, as done on ARM. Definition done in the board-specific or SoC-specific code.
  - By using a device tree, as done on Power PC, from which `platform_device` structures are created
- Example on ARM, where the instantiation is done in `arch/arm/mach-imx/mx1ads.c`

```
static struct platform_device imx_uart1_device = {
        .name           = "imx-uart",
        .id             = 0,
        .num_resources  = ARRAY_SIZE(imx_uart1_resources),
        .resource       = imx_uart1_resources,
        .dev = {
                .platform_data = &uart_pdata,
        }
};
```

Platform device instantiation (2)

- The device is part of a list

```
static struct platform_device *devices[] __initdata = {
        &cs89x0_device,
        &imx_uart1_device,
        &imx_uart2_device,
};
```

- And the list of devices is added to the system during board initialization

```
static void __init mx1ads_init(void)
{
    [...]
    platform_add_devices(devices, ARRAY_SIZE(devices));
}

MACHINE_START(MX1ADS, "Freescale MX1ADS")
    [...]
    .init_machine = mx1ads_init,
MACHINE_END
```

The resource mechanism

- Each device managed by a particular driver typically uses different hardware *resources:* addresses for the I/O registers, DMA channels, IRQ lines, etc.
- These informations can be represented using the `struct resource`, and an array of `struct resource` is associated to a `platform_device`
- Allows a driver to be instantiated for multiple devices functioning similarly, but with different addresses, IRQs, etc.

```
static struct resource imx_uart1_resources[] = {
        [0] = {
                .start  = 0x00206000,
                .end    = 0x002060FF,
                .flags  = IORESOURCE_MEM,
        },
        [1] = {
                .start  = (UART1_MINT_RX),
                .end    = (UART1_MINT_RX),
                .flags  = IORESOURCE_IRQ,
        },
};
```

Using resources

- When a platform_device is added to the system using `platform_add_device()`, the `probe()` method of the platform driver gets called
- This method is responsible for initializing the hardware, registering the device to the proper framework (in our case, the serial driver framework)
- The platform driver has access to the I/O resources:

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = ioremap(res->start, PAGE_SIZE);
sport->rxirq = platform_get_irq(pdev, 0);
```

`platform_data` mechanism

- In addition to the well-defined resources, many drivers require driver-specific informations for each platform device
- These informations can be passed using the `platform_data` field of the `struct device` (from which `struct platform_device` inherits)
- As it is a `void *` pointer, it can be used to pass any type of information.
  - Typically, each driver defines a structure to pass information through platform_data

platform_data example (1)

- The i.MX serial port driver defines the following structure to be passed through `platform_data`

```
struct imxuart_platform_data {
    int (*init)(struct platform_device *pdev);
    void (*exit)(struct platform_device *pdev);
    unsigned int flags;
    void (*irda_enable)(int enable);
    unsigned int irda_inv_rx:1;
    unsigned int irda_inv_tx:1;
    unsigned short transceiver_delay;
};
```

- The MX1ADS board code instantiates such a structure

```
static struct imxuart_platform_data uart1_pdata = {
.flags = IMXUART_HAVE_RTSCTS,
};
```

platform_data example (2)

- The uart_pdata structure is associated to the platform_device in the MX1ADS board file (the real code is slightly more complicated)

```
struct platform_device mx1ads_uart1 = {
    .name = "imx-uart",
    .dev {
        .platform_data = &uart1_pdata,
    },
    .resource = imx_uart1_resources,
    […]
};
```

- The driver can access the platform data:

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imxuart_platform_data *pdata;
    pdata = pdev->dev.platform_data;
    if (pdata && (pdata->flags & IMXUART_HAVE_RTSCTS))
        sport->have_rtscts = 1;
    […]
}
```

Driver-specific data structure

- Each « framework » defines a structure that a device driver must register to be recognized as a device in this framework
  - uart_port for serial port, netdev for network devices, fb_info for framebuffers, etc.
- In addition to this structure, the driver usually needs to store additional informations about its device
- This is typically done
  - By subclassing the « framework » structure
  - Or by storing a reference to the « framework » structure

Driver-specific data structure examples

i.MX serial driver: `imx_port` is a subclass of `uart_port`

```
struct imx_port {
        struct uart_port        port;
        struct timer_list       timer;
        unsigned int            old_status;
        int                     txirq,rxirq,rtsirq;
        unsigned int            have_rtscts:1;
        […]
};
```

rtl8150 network driver: `rtl8150` has a reference to `net_device`

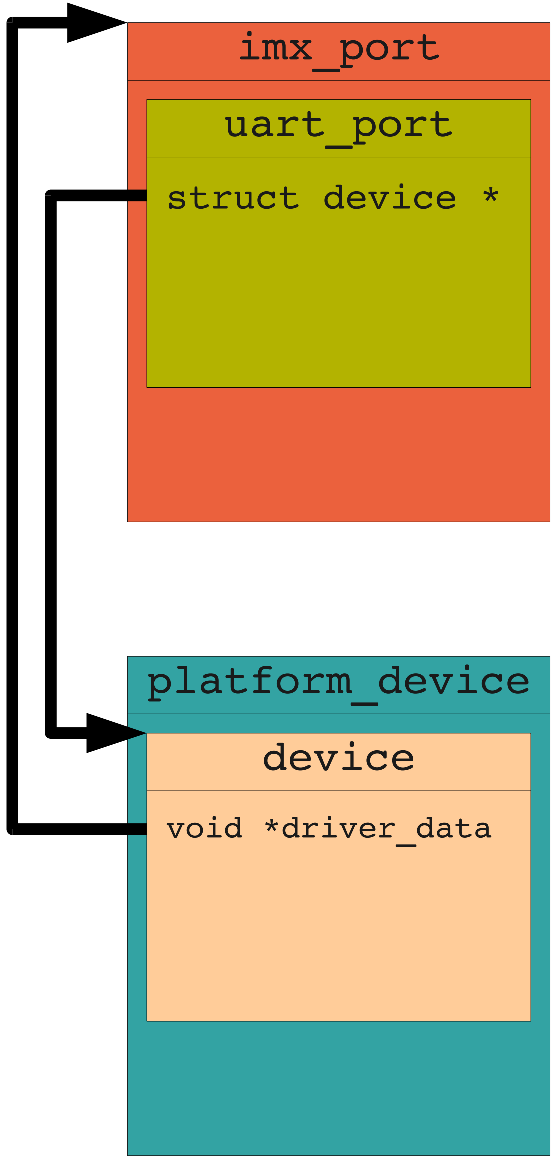
```
struct rtl8150 {
        unsigned long          flags;
        struct usb_device      *udev;
        struct tasklet_struct tl;
        struct net_device      *netdev;
        […]
};
```

Link between structures (1)

- The « framework » typically contains a `struct device *` pointer that the driver must point to the corresponding struct device
  - It's the relation between the logical device (for example a network interface) and the physical device (for example the USB network adapter)
- The device structure also contains a `void *` pointer that the driver can freely use.
  - It's often use to link back the device to the higher-level structure from the framework.
  - It allows, for example, from the `platform_device` structure, to find the structure describing the logical device

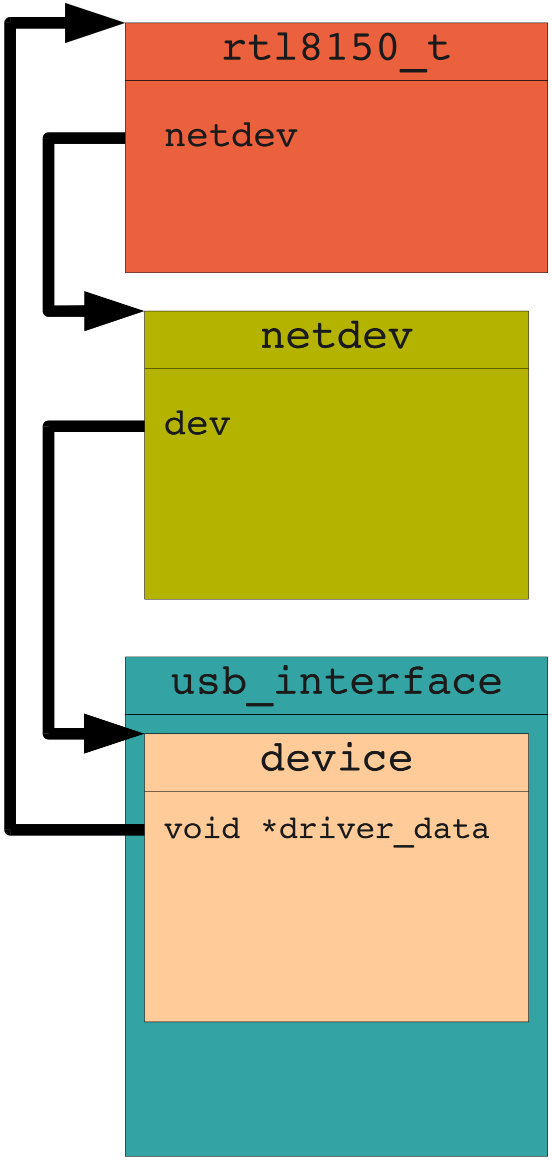
Link between structures (2)

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imx_port *sport;
    […]
    /* setup the link between uart_port and the struct
       device inside the platform_device */
    sport->port.dev = &pdev->dev;
    […]
    /* setup the link between the struct device inside
       the platform device to the imx_port structure */
    platform_set_drvdata(pdev, &sport->port);
    […]
    uart_add_one_port(&imx_reg, &sport->port);
}

static int serial_imx_remove(struct platform_device *pdev)
{
    /* retrive the imx_port from the platform_device */
    struct imx_port *sport = platform_get_drvdata(pdev);
    […]
    uart_remove_one_port(&imx_reg, &sport->port);
    […]
}
```



Link between structures (3)

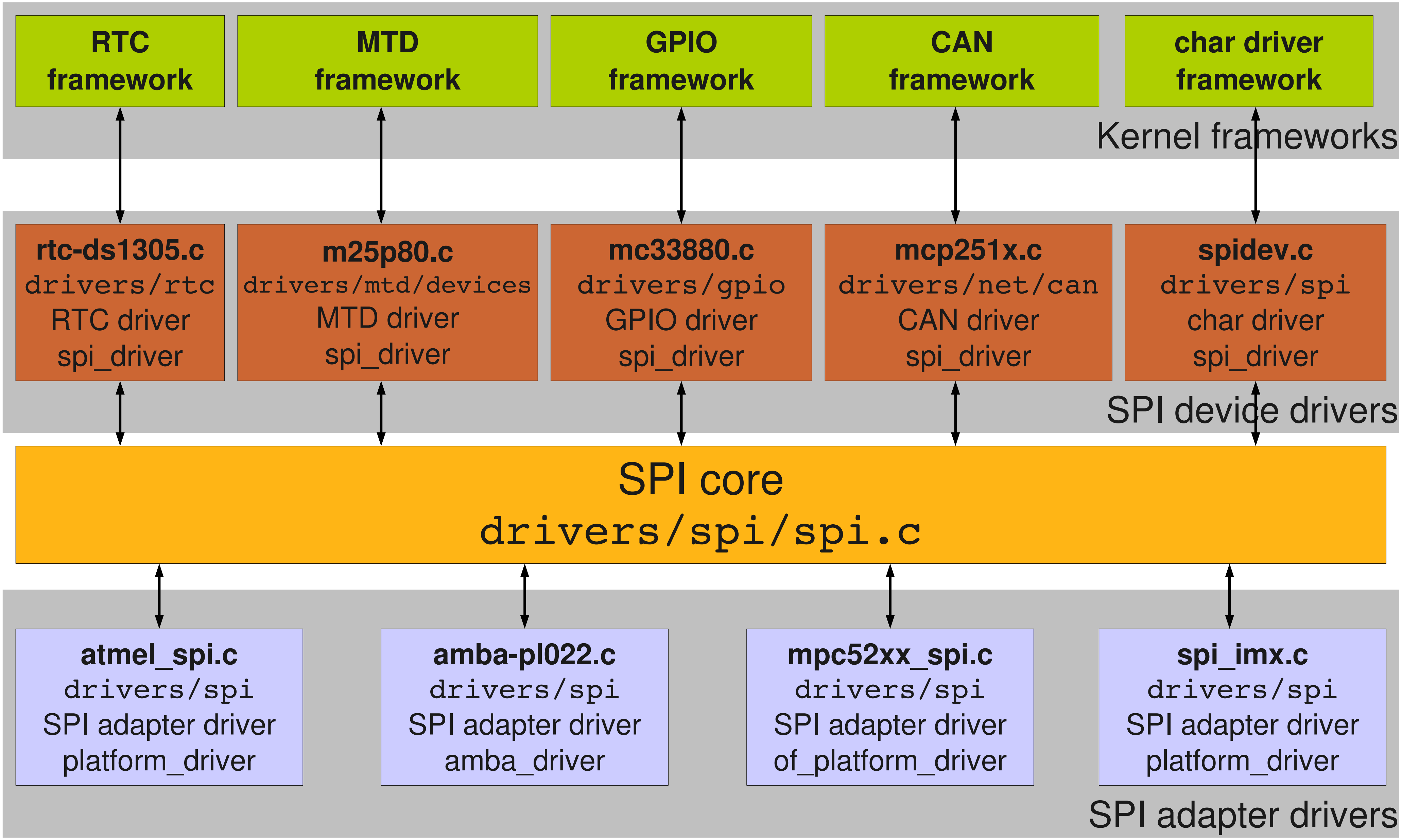
```
static int rtl8150_probe(struct usb_interface *intf,
                         const struct usb_device_id *id)
{
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    dev = netdev_priv(netdev);

    usb_set_intfdata(intf, dev);
    SET_NETDEV_DEV(netdev, &intf->dev);

    […]
}

static void rtl8150_disconnect(struct usb_interface *intf)
{
    rtl8150_t *dev = usb_get_intfdata(intf);

    […]
}
```



Example of another non-dynamic bus: SPI

- SPI is called non-dynamic as it doesn't support runtime enumeration of devices: the system needs to know which devices are on which SPI bus, and at which location
- The SPI infrastructure in the kernel is in drivers/spi
  - `drivers/spi/spi.c` is the core, which implements the struct `bus_type` for spi
    - It allows registration of adapter drivers using `spi_register_master()`, and registration of device drivers using `spi_register_driver()`
  - `drivers/spi/` contains many adapter drivers, for various platforms: Atmel, OMAP, Xilinx, Samsung, etc.
    - Most of them are platform_drivers or of_platform_drivers, one pci_driver, one amba_driver, one partport_driver
  - `drivers/spi/spidev.c` provides an infrastructure to access SPI bus from userspace
  - SPI device drivers are present all over the kernel tree

SPI components



SPI AT91 SoC code

```
static struct resource spi0_resources[] = {
        [0] = {
                .start  = AT91SAM9260_BASE_SPI0,
                .end    = AT91SAM9260_BASE_SPI0 + SZ_16K - 1,
                .flags  = IORESOURCE_MEM,
        },
        [1] = {
                .start  = AT91SAM9260_ID_SPI0,
                .end    = AT91SAM9260_ID_SPI0,
                .flags  = IORESOURCE_IRQ,
        },
};

static struct platform_device at91sam9260_spi0_device = {
        .name           = "atmel_spi",
        .id             = 0,
        .dev            = {
                                .dma_mask               = &spi_dmamask,
                                .coherent_dma_mask      = DMA_BIT_MASK(32),
        },
        .resource       = spi0_resources,
        .num_resources  = ARRAY_SIZE(spi0_resources),
};
```

arch/arm/mach-at91/at91sam9260_devices.c

SPI AT91 SoC code (2)

Registration of SPI devices with `spi_register_board_info()`, registration of SPI adapter with `platform_device_register()`

```
void __init at91_add_device_spi(struct spi_board_info *devices,
                                int nr_devices)
{
    […]

    spi_register_board_info(devices, nr_devices);

    /* Configure SPI bus(es) */
    if (enable_spi0) {
        at91_set_A_periph(AT91_PIN_PA0, 0);     /* SPI0_MISO */
        at91_set_A_periph(AT91_PIN_PA1, 0);     /* SPI0_MOSI */
        at91_set_A_periph(AT91_PIN_PA2, 0);     /* SPI1_SPCK */

        at91_clock_associate("spi0_clk", &at91sam9260_spi0_device.dev,
                             "spi_clk");
        platform_device_register(&at91sam9260_spi0_device);
    }

    […]
}
```

arch/arm/mach-at91/at91sam9260_devices.c

AT91RM9200DK board code for SPI

One `spi_board_info` structure for each SPI device connected to the system.

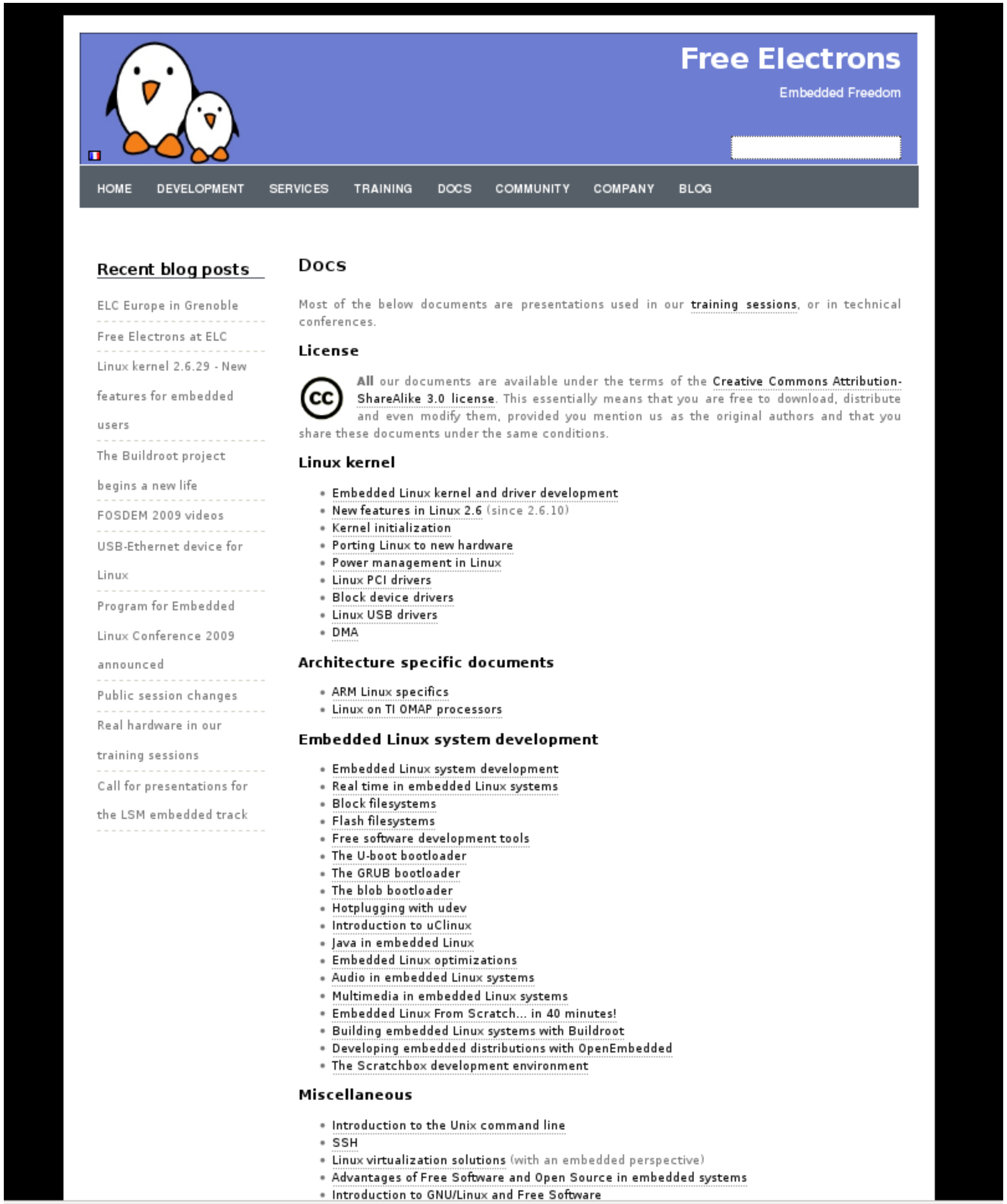
```
static struct spi_board_info dk_spi_devices[] = {
        {       /* DataFlash chip */
                .modalias       = "mtd_dataflash",
                .chip_select    = 0,
                .max_speed_hz   = 15 * 1000 * 1000,
        },
        {       /* UR6HCPS2-SP40 PS2-to-SPI adapter */
                .modalias       = "ur6hcps2",
                .chip_select    = 1,
                .max_speed_hz   = 250 *  1000,
        },
        […]
};

static void __init dk_board_init(void)
{
    […]
    at91_add_device_spi(dk_spi_devices, ARRAY_SIZE(dk_spi_devices));
    […]
}
```

arch/arm/mach-at91/board-dk.c

References

- Kernel documentation
  `Documentation/driver-model/`
  `Documentation/filesystems/sysfs.txt`
- Linux 2.6 Device Model
  http://www.bravegnu.org/device-model/device-model.html
- Linux Device Drivers, chapter 14 «The Linux Device Model»
  http://lwn.net/images/pdf/LDD3/ch14.pdf
- The kernel source code
  Full of examples of other drivers!

Related documents



All our technical presentations on http://free-electrons.com/docs

- Linux kernel
- Device drivers
- Architecture specifics
- Embedded Linux system development

How to help

You can help us to improve and maintain this document...

- By sending corrections, suggestions, contributions and translations
- By asking your organization to order development, consulting and training services performed by the authors of these documents (see http://free-electrons.com/).
- By sharing this document with your friends, colleagues and with the local Free Software community.
- By adding links on your website to our on-line materials, to increase their visibility in search engine results.

Free Electrons

Our services

Linux kernel

Linux device drivers
Board support code
Mainstreaming kernel code
Kernel debugging

Embedded Linux Training

All materials released with a free license!

Unix and GNU/Linux basics
Linux kernel and drivers development
Real-time Linux, uClinux
Development and profiling tools
Lightweight tools for embedded systems
Root filesystem creation
Audio and multimedia
System optimization

Custom Development

System integration
Embedded Linux demos and prototypes
System optimization
Application and interface development

Consulting and technical support

Help in decision making
System architecture
System design and performance review
Development tool and application support
Investigating issues and fixing tool bugs