



Embedded Linux kernel and driver development

Michael Opdenacker
Thomas Petazzoni
Free Electrons

© Copyright 2004-2009, Free Electrons.

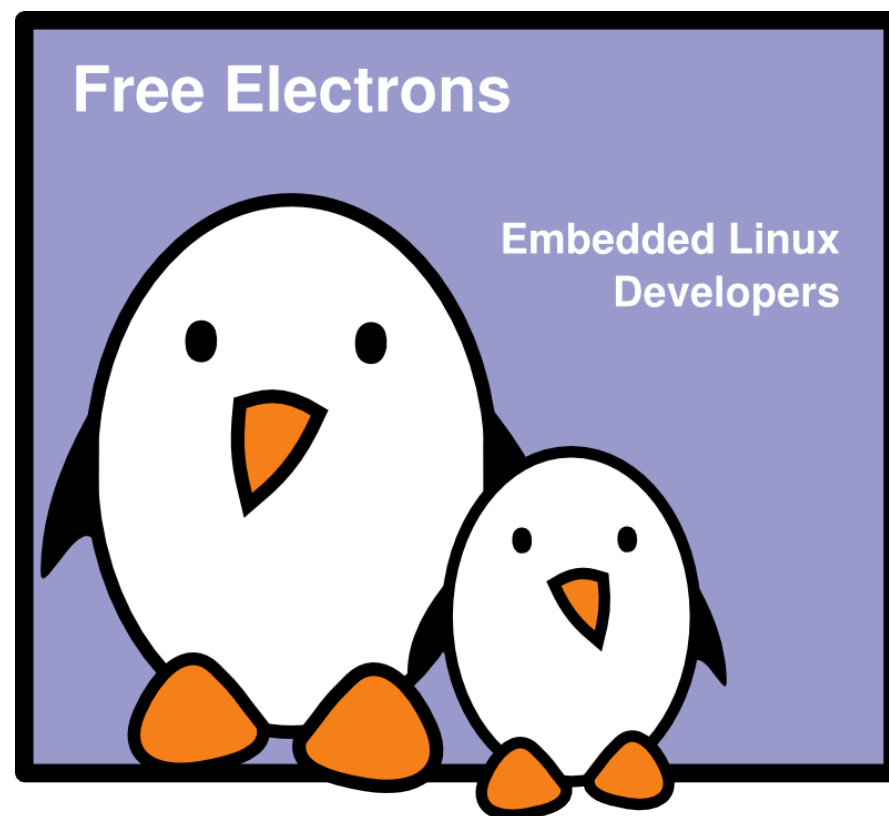
Creative Commons BY-SA 3.0 license

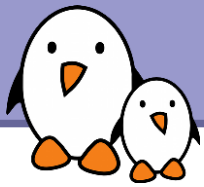
Latest update: Dec 24, 2009,

Document sources, updates and translations:

<http://free-electrons.com/docs/kernel>

Corrections, suggestions, contributions and translations are welcome!

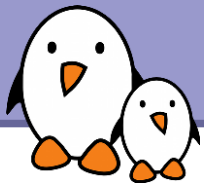




Contents

Driver development

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



Embedded Linux driver development

Linux code and device drivers



Supported kernel version

- ▶ The APIs covered in these training slides should be compliant with **Linux 2.6.31**.
- ▶ We may also mention features in more recent kernels.



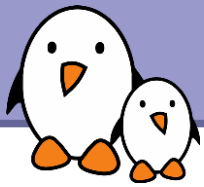
Programming language

- ▶ Implemented in C like all Unix systems.
(C was created to implement the first Unix systems)
- ▶ A little Assembly is used too:
 - ▶ CPU and machine initialization, exceptions
 - ▶ Critical library routines.
- ▶ No C++ used, see <http://www.tux.org/lkml/#s15-3>
- ▶ All the code compiled with gcc, the GNU C Compiler
 - ▶ Many gcc-specific extensions used in the kernel code, any ANSI C compiler will not compile the kernel
 - ▶ A few alternate compilers are supported (Intel and Marvell)
 - ▶ See <http://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc/C-Extensions.html>



No C library

- ▶ The kernel has to be standalone and can't use user-space code.
Userspace is implemented on top of kernel services, not the opposite.
Kernel code has to supply its own library implementations (string utilities, cryptography, uncompression ...)
- ▶ So, you can't use standard C library functions in kernel code. (`printf()`, `memset()`, `malloc()` ...).
You can also use kernel C headers.
- ▶ Fortunately, the kernel provides **similar** C functions for your convenience, like `printk()`, `memset()`, `kmalloc()` ...



Portability

- ▶ The Linux kernel code is designed to be portable
- ▶ All code outside `arch/` should be portable
- ▶ To this aim, the kernel provides macros and functions to abstract the architecture specific details
 - ▶ Endianness
(`cpu_to_be32`, `cpu_to_le32`, `be32_to_cpu`, `le32_to_cpu`)
 - ▶ I/O memory access
 - ▶ Memory barriers to provide ordering guarantees if needed
 - ▶ DMA API to flush and invalidate caches if needed



No floating point computation

- ▶ Never use floating point numbers in kernel code. Your code may be run on a processor without a floating point unit (like on [arm](#)).
- ▶ Don't be confused with floating point related configuration options
 - ▶ They are related to the emulation of floating point operation performed by the user space applications, triggering an exception into the kernel
 - ▶ Using soft-float, i.e. emulation in user-space, is however recommended for performance reasons



No stable Linux internal API (1)

- ▶ The internal kernel API to implement kernel code can undergo changes between two `2.6.x` releases. A stand-alone driver compiled for a given version may no longer compile or work on a more recent one.
See `Documentation/stable_api_nonsense.txt` in kernel sources for reasons why.
- ▶ Of course, the external API must not change (system calls, `/proc`, `/sys`), as it could break existing programs. New features can be added, but kernel developers try to keep backward compatibility with earlier versions, at least for 1 or several years.
- ▶ Whenever a developer changes an internal API, (s)he also has to update all kernel code which uses it. Nothing broken!
- ▶ Works great for code in the mainline kernel tree.
Difficult to keep in line for out of tree or closed-source drivers!
- ▶ Kernel API changes reference: <http://lwn.net/Articles/2.6-kernel-api/>
- ▶ Feature removal schedule: <Documentation/feature-removal-schedule.txt>



No stable Linux internal API (2)

USB example

- ▶ Linux has updated its USB internal API at least 3 times (fixes, security issues, support for high-speed devices) and has now the fastest USB bus speeds (compared to other systems)
- ▶ Windows XP also had to rewrite its USB stack 3 times. But, because of closed-source, binary drivers that can't be updated, they had to keep backward compatibility with all earlier implementation. This is very costly (development, security, stability, performance).

See “Myths, Lies, and Truths about the Linux Kernel”, by Greg K.H., for details about the kernel development process:

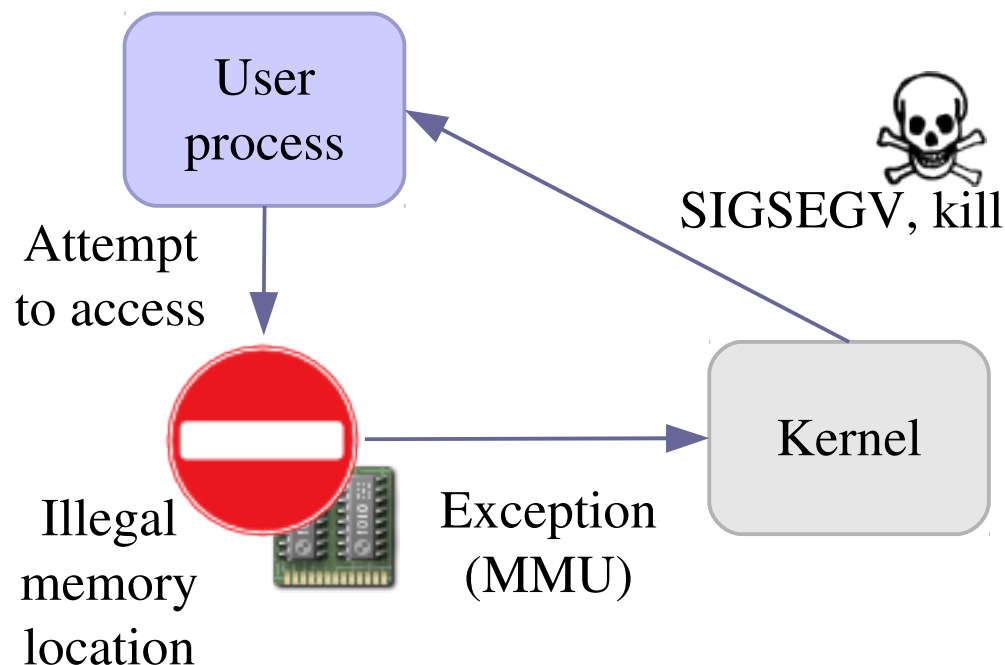
http://kroah.com/log/linux/ols_2006_keynote.html



Kernel memory constraints

Who can look after the kernel?

- ▶ No memory protection
Accessing illegal memory locations result in (often fatal) kernel oopses.
- ▶ Fixed size stack (8 or 4 KB)
Unlike in userspace, no way to make it grow.
- ▶ Kernel memory can't be swapped out (for the same reasons).



Userspace memory management

Used to implement:

- memory protection
- stack growth
- memory swapping to disk
- demand paging



Linux kernel licensing constraints

- ▶ The Linux kernel is licensed under the GNU General Public License version 2
 - ▶ This license gives you the right to use, study, modify and share the software freely
- ▶ However, when the software is redistributed, either modified or unmodified, the GPL requires that you redistribute the software under the same licence, with the source code
 - ▶ If modifications are made to the Linux kernel (for example to adapt it to your hardware), it is a derivative work of the kernel, and therefore must be released under GPLv2
 - ▶ The validity of the GPL on this point has already been verified in courts
- ▶ However, you're only required to do so
 - ▶ At the time the device starts to be distributed
 - ▶ To your customers, not to the entire world



Proprietary code and the kernel

- ▶ It is illegal to distribute a binary kernel that includes statically compiled proprietary drivers
- ▶ The kernel modules are a grey area : are they derived works of the kernel or not ?
 - ▶ The general opinion of the kernel community is that proprietary drivers are bad
http://www.linuxfoundation.org/en/Kernel_Driver_Statement
 - ▶ From a legal point of view, each driver is probably a different case
 - ▶ Is it really useful to keep your drivers secret ?
- ▶ There are some examples of proprietary drivers, like the Nvidia graphics drivers
 - ▶ They use a wrapper between the driver and the kernel
 - ▶ Unclear whether it makes it legal or not



Advantages of GPL drivers

From the driver developer / decision maker point of view

- ▶ You don't have to write your driver from scratch. You can reuse code from similar free software drivers.
- ▶ You get free community contributions, support, code review and testing. Proprietary drivers (even with sources) don't get any.
- ▶ Your drivers can be freely shipped by others (mainly by distributions).
- ▶ Closed source drivers often support a given kernel version. A system with closed source drivers from 2 different sources is unmanageable.
- ▶ Users and the community get a positive image of your company. Makes it easier to hire talented developers.
- ▶ You don't have to supply binary driver releases for each kernel version and patch version (closed source drivers).
- ▶ Drivers have all privileges. You need the sources to make sure that a driver is not a security risk.
- ▶ Your drivers can be statically compiled into the kernel.



Advantages of in-tree kernel drivers

Advantages of having your drivers in the mainline kernel sources

- ▶ Once your sources are accepted in the mainline tree, they are maintained by people making changes.
- ▶ Cost-free maintenance, security fixes and improvements.
- ▶ Easy access to your sources by users.
- ▶ Many more people reviewing your code.



Userspace device drivers (1)

Possible to implement device drivers in user-space!

- ▶ Such drivers just need access to the devices through minimum, generic kernel drivers.
- ▶ Examples
 - ▶ Printer and scanner drivers
(on top of generic parallel port / USB drivers)
 - ▶ X drivers: low level kernel drivers + user space X drivers.
 - ▶ Userspace drivers based on UIO,
framework introduced in 2.6.23



Userspace device drivers (2)

► Advantages

No need for kernel coding skills. Easier to reuse code between devices.

Drivers can be written in any language, even Perl 😊

Drivers can be kept proprietary.

Driver code can be killed and debugged. Cannot crash the kernel.

Can be swapped out (kernel code cannot be).

Can use floating-point computation.

Less in-kernel complexity.

► Drawbacks

Less straightforward to handle interrupts.

Increased latency vs. kernel code.

See UIO-Howto in the kernel documentation for detailed informations and the «Using UIO on an Embedded platform» talk at ELC 2008, http://www.celinux.org/elc08_presentations/uio080417celfelc08.pdf.



Embedded Linux driver development

Linux sources



Linux sources structure (1)

<code>arch/<arch></code>	Architecture specific code
<code>arch/<arch>/<mach></code>	Machine / board specific code
<code>block/</code>	Block layer core
<code>COPYING</code>	Linux copying conditions (GNU GPL)
<code>CREDITS</code>	Linux main contributors
<code>crypto/</code>	Cryptographic libraries
<code>Documentation/</code>	Kernel documentation. Don't miss it!
<code>drivers/</code>	All device drivers except sound ones (usb, pci...)
<code>fs/</code>	Filesystems (<code>fs/ext3/</code> , etc.)
<code>include/</code>	Kernel headers
<code>include/asm-<arch></code>	Architecture and machine dependent headers
<code>include/linux</code>	Linux kernel core headers
<code>init/</code>	Linux initialization (including <code>main.c</code>)



Linux sources structure (2)

<code>ipc/</code>	Code used for process communication
<code>Kbuild</code>	Part of the kernel build system
<code>kernel/</code>	Linux kernel core (very small!)
<code>lib/</code>	Misc library routines (zlib , crc32...)
<code>MAINTAINERS</code>	Maintainers of each kernel part. Very useful!
<code>Makefile</code>	Top Linux makefile (sets arch and version)
<code>mm/</code>	Memory management code (small too!)
<code>net/</code>	Network support code (not drivers)
<code>README</code>	Overview and building instructions
<code>REPORTING-BUGS</code>	Bug report instructions
<code>samples/</code>	Sample code (markers, kprobes, kobjects)
<code>scripts/</code>	Scripts for internal or external use
<code>security/</code>	Security model implementations (SELinux...)
<code>sound/</code>	Sound support code and drivers
<code>usr/</code>	Code to generate an initramfs cpio archive.



Accessing development sources (1)

Useful if you are involved in kernel development or if you found a bug in the source code.

- ▶ Kernel development sources are now managed with `git`:
<http://kernel.org/pub/software/scm/git/>
- ▶ You can browse Linus' `git` tree (if you just need to check a few files):
<http://www.kernel.org/git/?p=linux/kernel/git/torvalds/linux-2.6.git;a=tree>

You can also directly use `git` on your workstation

- ▶ Debian / Ubuntu: install the `git-core` package
- ▶ If you are behind a proxy, set Unix environment variables defining proxy settings. Example:

```
export http_proxy="proxy.server.com:8080"  
export ftp_proxy="proxy.server.com:8080"
```



Accessing development sources (2)

► Choose a git development tree on <http://git.kernel.org/>

► Get a local copy (“clone”) of this tree.

Example (Linus tree, the one used for [Linux](#) stable releases):

```
git-clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

► Update your copy whenever needed (Linus tree example):

```
cd linux-2.6  
git pull
```

See our training materials on git:

<http://free-electrons.com/docs/git/>



Kernel source management tools



Cscope

<http://cscope.sourceforge.net/>

- ▶ Tool to browse source code
(mainly C, but also C++ or Java)
- ▶ Supports huge projects like the Linux kernel
Takes less than 1 min. to index Linux 2.6.17
sources (fast!)
- ▶ Can be used from editors like `vim` and `emacs`.
- ▶ In Linux kernel sources, run it with:
`cscope -Rk`
(see `man cscope` for details)



Allows searching code for:

- all references to a symbol
- global definitions
- functions called by a function
- functions calling a function
- text string
- regular expression pattern
- a file
- files including a file



Cscope screenshot

```
xterm
C symbol: request_irq

File                Function                Line
0 omap_udc.c         omap_udc_probe           2821 status = request_irq(pdev->resource[1].start, omap_udc_irq,
1 omap_udc.c         omap_udc_probe           2830 status = request_irq(pdev->resource[2].start, omap_udc_pio_irq,
2 omap_udc.c         omap_udc_probe           2838 status = request_irq(pdev->resource[3].start, omap_udc_iso_irq,
3 pxa2xx_udc.c       pxa2xx_udc_probe        2517 retval = request_irq(IRQ_USB, pxa2xx_udc_irq,
4 pxa2xx_udc.c       pxa2xx_udc_probe        2528 retval = request_irq(LUBBOCK_USB_DISC_IRQ,
5 pxa2xx_udc.c       pxa2xx_udc_probe        2539 retval = request_irq(LUBBOCK_USB_IRQ,
6 hc_crisv10.c       etrax_usb_hc_init        4423 if (request_irq(ETRAX_USB_HC_IRQ, etrax_usb_hc_interrupt_top_half,
                                0,
7 hc_crisv10.c       etrax_usb_hc_init        4431 if (request_irq(ETRAX_USB_RX_IRQ, etrax_usb_rx_interrupt, 0,
8 hc_crisv10.c       etrax_usb_hc_init        4439 if (request_irq(ETRAX_USB_TX_IRQ, etrax_usb_tx_interrupt, 0,
9 amifb.c            amifb_init              2431 if (request_irq(IRQ_AMIGA_COPPER, amifb_interrupt, 0,
a arcfb.c            arcfb_probe             564 if (request_irq(par->irq, &arcfb_interrupt, SA_SHIRQ,
b atafb.c            atafb_init              2720 request_irq(IRQ_AUTO_4, falcon_vbl_switcher, IRQ_TYPE_PRIO,
c atyfb_base.c       aty_enable_irq          1562 if (request_irq(par->irq, aty_irq, SA_SHIRQ, "atyfb", par)) {

* 155 more lines - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```



KScope

<http://kscope.sourceforge.net>

- ▶ A graphical front-end to [Cscope](#)
- ▶ Makes it easy to browse and edit the Linux kernel sources
- ▶ Can display a function call tree
- ▶ Nice editing features: symbol completion, spelling checker, automatic indentation...
- ▶ Usage guidelines:
Use the [Kernel](#) setting to ignore standard C includes.
Make sure the project name doesn't contain blank characters!



KScope screenshots (1)

linux-2.6.17 - /home/mike/linux/linux-2.6.17/drivers/char/mem.c - KScope

File Edit View Project Cscope Go Bookmarks Tools Window Settings Help

mem.c uaccess.h uaccess.c

Name

- read_zero
- read_zero
- read_port
- read_oldmem
- read_null
- read_mem
- read_kmem**
- pipe_to_null
- phys_mem_access_prot
- open_port
- null_lseek
- mmap_zero
- mmap_zero
- mmap_mem
- mmap_kmem
- memory_open

Symbols in current file

```
static int mmap_kmem(struct file * file, struct vm_area_struct * vma)
{
    unsigned long pfn;

    /* Turn a kernel-virtual address into a physical page frame */
    pfn = __pa((u64)vma->vm_pgoff << PAGE_SHIFT) >> PAGE_SHIFT;

    /*
     * RED-PEN: on some architectures there is more mapped memory
     * than available in mem_map which pfn_valid checks
     * for. Perhaps should add a new macro here.
     *
     * RED-PEN: vmalloc is not supported right now.
     */
    if (!pfn_valid(pfn))
        return -EIO;

    vma->vm_pgoff = pfn;
    return mmap_mem(file, vma);
}

#ifdef CONFIG_CRASH_DUMP
/*
```

Main window

Project files

- crypto
- cscope.out
- Documentation
- drivers
 - acorn
 - acpi
 - amba
 - atm
 - base
 - block
 - bluetooth
 - cdrom
 - char
 - agp
 - amiserial.c

Set Root... Find File...

Definition of kmalloc

Function	File	Line	Text
kmalloc	/home/mike/linux/linux-2.6.17/mm/slob.c	161	void *kmalloc(size_t size, gfp_t gfp)
kmalloc	/home/mike/linux/linux-2.6.17/include/linux/slab.h	90	static inline void *kmalloc(size_t size, gfp_t flags)
kmalloc	/home/mike/linux/linux-2.6.17/include/asm-s390/debug.h	250	#define kmalloc(x...) (PRINT_INFO(" kmalloc %p\n",b=kmalloc(x)),b)

Query window

Line: 370 Col: 1



KScope screenshots (2)

Called Functions Tree

Function File Line Text

- kmalloc
 - slob_free /home/mike/linux/linux-2.6.17/mm/slob.c 187 slob_free(bb, sizeof(bigblock_t));
 - spin_unlock_irqrestore /home/mike/linux/linux-2.6.17/mm/slob.c 183 spin_unlock_irqrestore(&block_lock, flags);
 - spin_lock_irqsave /home/mike/linux/linux-2.6.17/mm/slob.c 180 spin_lock_irqsave(&block_lock, flags);
 - __get_free_pages /home/mike/linux/linux-2.6.17/mm/slob.c 177 bb->pages = (void *)__get_free_pages(gfp, bb->order);
 - find_order /home/mike/linux/linux-2.6.17/mm/slob.c 176 bb->order = find_order(size);
 - slob_alloc /home/mike/linux/linux-2.6.17/mm/slob.c 172 bb = slob_alloc(sizeof(bigblock_t), gfp, 0);
 - slob_alloc /home/mike/linux/linux-2.6.17/mm/slob.c 168 m = slob_alloc(size + SLOB_UNIT, gfp, 0);
 - __kmalloc /home/mike/linux/linux-2.6.17/include/linux/slab.h 110 return __kmalloc(size, flags);
 - __builtin_return_address /home/mike/linux/linux-2.6.17/mm/slab.c 3300 return __do_kmalloc(size, flags, __builtin_return_address(0));
 - __do_kmalloc /home/mike/linux/linux-2.6.17/mm/slab.c 3300 return __do_kmalloc(size, flags, __builtin_return_address(0));
 - __do_kmalloc /home/mike/linux/linux-2.6.17/mm/slab.c 3298 return __do_kmalloc(size, flags, NULL);
 - kmem_cache_alloc /home/mike/linux/linux-2.6.17/include/linux/slab.h 106 return kmem_cache_alloc((flags & GFP_DMA) ? GFP_DMA : GFP_KERNEL, size);
 - __you_cannot_kmalloc_that_much /home/mike/linux/linux-2.6.17/include/linux/slab.h 103 __you_cannot_kmalloc_that_much();
 - __you_cannot_kmalloc_that_much /home/mike/linux/linux-2.6.17/include/linux/slab.h 102 extern void __you_cannot_kmalloc_that_much(void);
 - __builtin_constant_p /home/mike/linux/linux-2.6.17/include/linux/slab.h 92 if (__builtin_constant_p(size)) {

Right-click a tree item for more options.

Called functions tree



LXR: Linux Cross Reference

<http://sourceforge.net/projects/lxr>


Generic source indexing tool
and code browser

- ▶ Web server based
Very easy and fast to use
- ▶ Identifier or text search available
- ▶ Very easy to find the declaration,
implementation or usages of symbols
- ▶ Supports C and C++
- ▶ Supports huge code projects
such as the Linux kernel
(264 M in version 2.6.26).

- ▶ Takes a little time and patience to setup
(configuration, indexing, server
configuration).
- ▶ Indexing a new version is very fast:
approximately 20 minutes with LXR 0.3.1
(old version, but scales very well).
- ▶ You don't need to set up LXR by yourself.
Use our <http://lxr.free-electrons.com>
server! Other servers available on the
Internet:
<http://free-electrons.com/community/kernel/lxr/>
- ▶ This makes LXR the simplest solution
to browse standard kernel sources.



LXR screenshot



Linux Cross Reference

Free Electrons
Embedded Freedom

• [Source Navigation](#) • [Diff Markup](#) • [Identifier Search](#) • [Freetext Search](#) •

Version: 2.6.24 2.6.25 2.6.26 2.6.27 2.6.28 2.6.29
Architecture: x86 m68k m68knommu mips powerpc sh blackfin

Linux/kernel/user.c

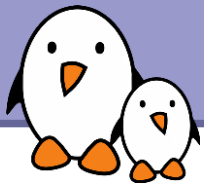
```
1 /*
2  * The "user cache".
3  *
4  * (C) Copyright 1991-2000 Linus Torvalds
5  *
6  * We have a per-user structure to keep track of how many
7  * processes, files etc the user has claimed, in order to be
8  * able to have per-user limits for system resources.
9  */
10
11 #include <linux/init.h>
12 #include <linux/sched.h>
13 #include <linux/slab.h>
14 #include <linux/bitops.h>
15 #include <linux/key.h>
16 #include <linux/interrupt.h>
17 #include <linux/module.h>
18 #include <linux/user_namespace.h>
19 #include "cred-internals.h"
20
21 struct user_namespace init_user_ns = {
22     .kref = {
23         .refcount      = ATOMIC_INIT(1),
24     },
25     .creator = &root_user,
26 };
27 EXPORT_SYMBOL_GPL(init_user_ns);
28
```



Practical lab – Kernel sources

- ▶ Get familiar with the sources
- ▶ Use a kernel source indexing tool





Embedded Linux driver development

Driver development
Loadable kernel modules



Loadable kernel modules

- ▶ Modules: add a given functionality to the kernel (drivers, filesystem support, and many others).
- ▶ Can be loaded and unloaded at any time, only when their functionality is need. Once loaded, have full access to the whole kernel address space. No particular protection.
- ▶ Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...



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>



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 out-of-tree module

- ▶ The below Makefile should be reusable for any out-of-tree Linux 2.6 module
 - ▶ In-tree modules are covered later
- ▶ 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)

```
# Makefile for the hello module
```

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

```
KDIR := /lib/modules/$(shell uname -r)/build
```

```
PWD := $(shell pwd)
```

```
default:
```

```
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

[Tab]!
(no spaces)

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

Example available on <http://free-electrons.com/doc/c/Makefile>



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 »



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.



Module dependencies

- ▶ Definition: `module2` depends on `module1` if `module2` uses a symbol exported by `module1`.
- ▶ Module dependencies are stored in `/lib/modules/<version>/modules.dep`
- ▶ They are automatically computed during kernel building from module exported symbols.
- ▶ Example: `usb_storage` depends on `usbcore`, because it uses some of the functions exported by `usbcore`.
- ▶ You can also update the `modules.dep` file by yourself, by running (as `root`):
`depmod -a [<version>]`



hello module with parameters

```
/* 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



Passing module parameters

- ▶ Through `insmod`:

```
sudo insmod ./hello_param.ko howmany=2 whom=universe
```

- ▶ Through `modprobe`:

Set parameters in `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:

```
options hello_param howmany=2 whom=universe
```

- ▶ Through the kernel command line,
when the module is built statically into the kernel:

```
hello_param.howmany=2 hello_param.whom=universe
```

driver name ↑
driver parameter name ↑
driver parameter value ↗



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);
```



Declaring a module parameter array

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

module_param_array(
    name,      /* name of an already defined array */
    type,      /* same as in module_param */
    num,       /* number of elements in the array, or NULL (no check?) */
    perm       /* same as in module_param */
);
```

Example

```
static int base[MAX_DEVICES] = { 0x820, 0x840 };
module_param_array(base, int, NULL, 0);
```



Driver development

Adding sources to the kernel tree



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`
- ▶ 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
```

- ▶ Run `make xconfig` and see your new options!
- ▶ Run `make` and your new files are compiled!
- ▶ See [Documentation/kbuild/](#) for details



How to create Linux patches

- ▶ Download the **latest** kernel sources
- ▶ Make a copy of these sources:
`rsync -a linux-2.6.9-rc2/ linux-2.6.9-rc2-patch/`
- ▶ Apply your changes to the copied sources, and test them.
- ▶ Run `make distclean` to keep only source files.
- ▶ Create a patch file:
`diff -Nur linux-2.6.9-rc2/ \`
`linux-2.6.9-rc2-patch/ > patchfile`
 - ▶ Always compare the whole source structures
(suitable for `patch -p1`)
 - ▶ Patch file name: should recall what the patch is about.
- ▶ If you need to manage a lot of patches, use `git` or `quilt` instead



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



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





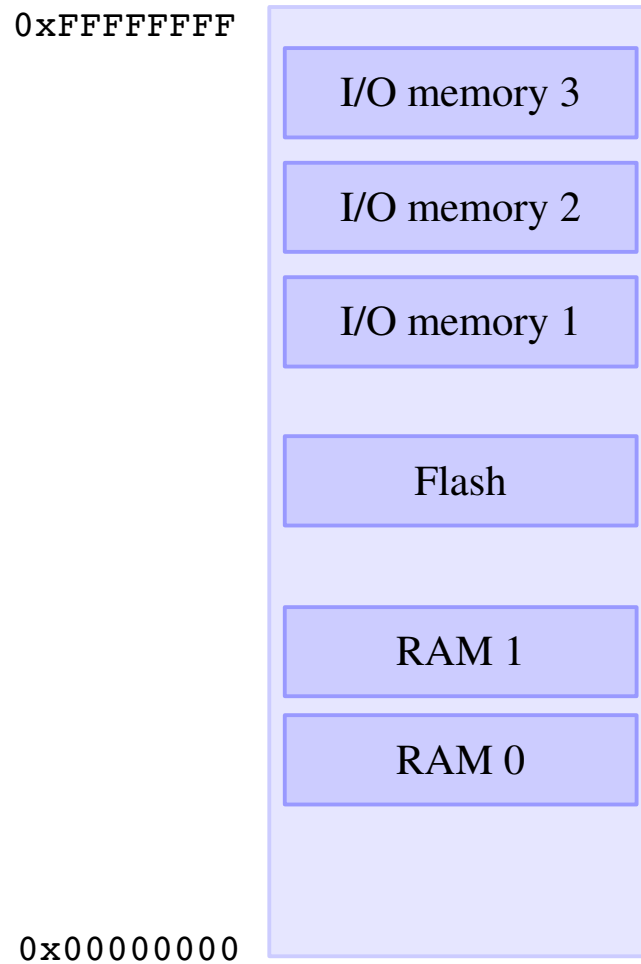
Embedded Linux driver development

Driver development
Memory management

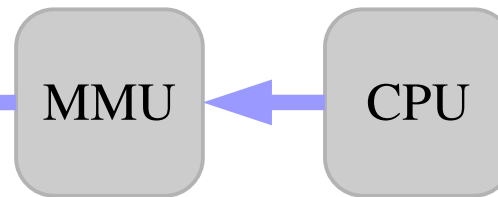


Physical and virtual memory

Physical address space

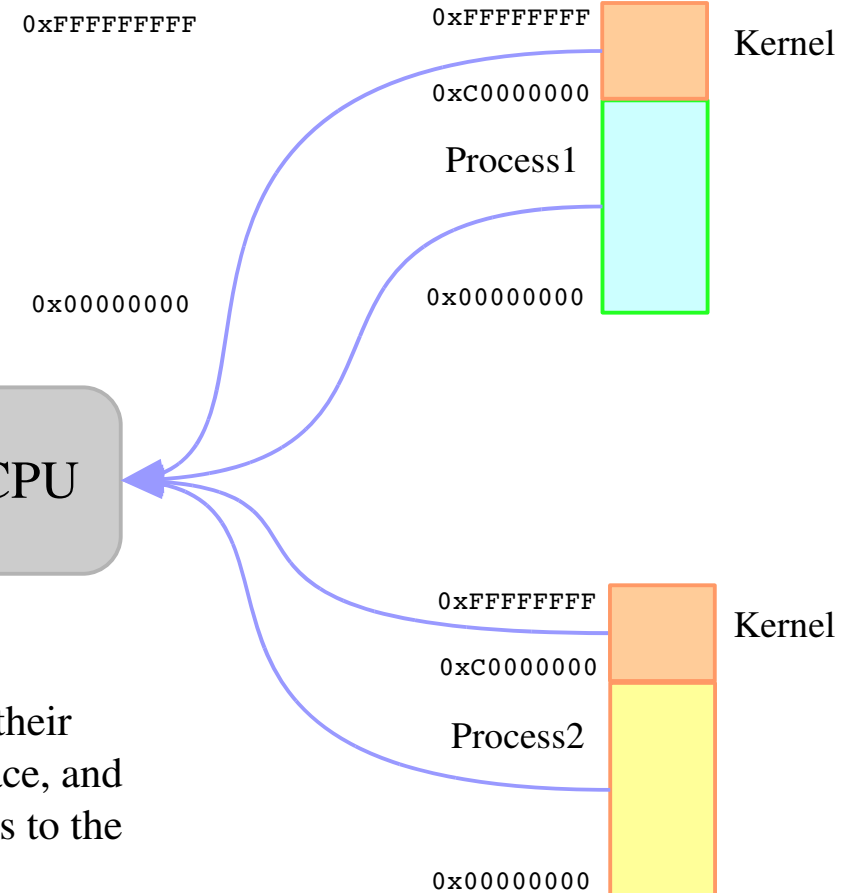


Memory
Management
Unit



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

Virtual address spaces





kmalloc and kfree

- ▶ Basic allocators, kernel equivalents of `glibc`'s `malloc` and `free`.
- ▶ `#include <linux/slab.h>`
- ▶ `static inline void *kmalloc(size_t size, int flags);`
 `size`: number of bytes to allocate
 `flags`: priority (explained in a few pages)
- ▶ `void kfree (const void *objp);`
- ▶ Example: (`drivers/infiniband/core/cache.c`)

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



kmalloc features

- ▶ Quick (unless it's blocked waiting for memory to be freed).
- ▶ Doesn't initialize the allocated area.
- ▶ The allocated area is contiguous in physical RAM.
- ▶ Allocates by 2^n sizes, and uses a few management bytes.
So, don't ask for 1024 when you need 1000! You'd get 2048!
- ▶ Caution: drivers shouldn't try to `kmalloc` more than 128 KB (upper limit in some architectures).
- ▶ Minimum memory consumption:
32 or 64 bytes (page size dependent).





Main kmalloc flags (1)

Defined in `include/linux/gfp.h` (GFP: `__get_free_pages`)

- ▶ `GFP_KERNEL`

Standard kernel memory allocation. May block. Fine for most needs.

- ▶ `GFP_ATOMIC`

RAM allocated from code which is not allowed to block (interrupt handlers) or which doesn't want to block (critical sections). Never blocks.

- ▶ `GFP_USER`

Allocates memory for user processes. May block. Lowest priority.



Main kmalloc flags (2)

Extra flags (can be added with |)

▶ `__GFP_DMA` or `GFP_DMA`

Allocate in DMA zone

▶ `__GFP_ZERO`

Returns a zeroed page.

▶ `__GFP_NOFAIL`

Must not fail. Never gives up. Caution: use only when mandatory!

▶ `__GFP_NORETRY`

If allocation fails, doesn't try to get free pages.

▶ Example:

`GFP_KERNEL | __GFP_DMA`

▶ Note: almost only

`__GFP_DMA` or `GFP_DMA` used in device drivers.



Related allocation functions

Again, names similar to those of C library functions

- ▶ `static inline void *kzalloc(
 size_t size, gfp_t flags);`

Zeroes the allocated buffer.

- ▶ `static inline 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 * __must_check krealloc(
 const void *, size_t, gfp_t);`

Changes the size of the given buffer.



Available allocators

Memory is allocated using *slabs* (groups of one or more continuous pages from which objects are allocated). Several compatible slab allocators are available:

- ▶ **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.

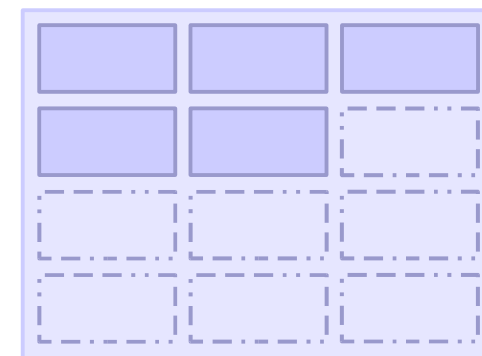
⊖ Choose SLAB allocator (NEW)

- | | |
|---|------|
| <input checked="" type="radio"/> SLAB | SLAB |
| <input type="radio"/> SLUB (Unqueued Allocator) (NEW) | SLUB |
| <input type="radio"/> SLOB (Simple Allocator) | SLOB |



Slab caches and memory pools

- ▶ *Slab caches*: make it possible to allocate multiple objects of the same size, without wasting RAM.
- ▶ So far, mainly used in core subsystems, but not much in device drivers (except USB and SCSI drivers)
- ▶ Memory pools: pools of preallocated objects, to increase the chances of allocations to succeed. Often used with file caches.





Allocating by pages

More appropriate when you need big slices of RAM:

▶ 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).

▶ `unsigned long get_zeroed_page(int flags);`
Returns a pointer to a free page and fills it up with zeros

▶ `unsigned long __get_free_page(int flags);`
Same, but doesn't initialize the contents

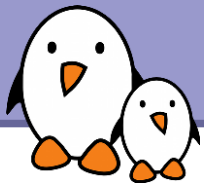
▶ `unsigned long __get_free_pages(int flags,
 unsigned int order);`

Returns a pointer on an area of several contiguous pages in physical RAM.

`order: $\log_2(\text{<number_of_pages>})$`

If variable, can be computed from the size with the `get_order` function.

Maximum: 8192 KB (`MAX_ORDER=11` in `include/linux/mmzone.h`),
except in a few architectures when overwritten with `CONFIG_FORCE_MAX_ZONEORDER`.



Freeing pages

- ▶ `void free_page(unsigned long addr);`
- ▶ `void free_pages(unsigned long addr,
 unsigned int order);`

Need to use the same `order` as in allocation.



vmalloc

`vmalloc` can be used to obtain contiguous memory zones in **virtual** address space (even if pages may not be contiguous in physical memory).

- ▶ `void *vmalloc(unsigned long size);`
- ▶ `void vfree(void *addr);`



Memory utilities

▶ `void * memset(void * s, int c, size_t count);`

Fills a region of memory with the given value.

▶ `void * memcpy(void * dest,
 const void *src,
 size_t count);`

Copies one area of memory to another.

Use `memmove` with overlapping areas.

▶ Lots of functions equivalent to standard C library ones defined in `include/linux/string.h` and in `include/linux/kernel.h` (`sprintf`, etc.)



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 production!



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



Requesting I/O ports

`/proc/ioprots` 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
...
```

► `struct resource *request_region(
 unsigned long start,
 unsigned long len,
 char *name);`

Tries to reserve the given region and returns `NULL` if unsuccessful. Example:

► `request_region(0x0170, 8, "ide1");`

► `void release_region(
 unsigned long start,
 unsigned long len);`

► See `include/linux/ioport.h` and `kernel/resource.c`



Mapping I/O ports in virtual memory

- ▶ Since Linux 2.6.9, it is possible to get a virtual address corresponding to an I/O ports range.
- ▶ Allows for transparent use of PIO and MMIO at the same time, as this memory can be accessed in the same way as MMIO memory.
- ▶ The mapping is done using the `ioport_map` function:

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

```
void *ioport_map(unsigned long port,  
                unsigned long nr);
```

```
void ioport_unmap(void *address);
```



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
...
```

- ▶ Equivalent functions with the same interface
- ▶ `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

- ▶ To access I/O memory, drivers need to have a virtual address that the processor can handle.
- ▶ 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!
- ▶ Note that an `ioremap_nocache` function exists.
This disables the CPU cache at the given address range.



Differences with standard memory

- ▶ Reads and writes on memory can be cached
- ▶ The compiler may choose to write the value in a cpu register, and may never write it in main memory.
- ▶ The compiler may decide to optimize or reorder read and write instructions.



Avoiding I/O access issues

- ▶ Caching on I/O ports or memory already disabled, either by the hardware or by Linux init code.
- ▶ Use the `volatile` statement in your C code to prevent the compiler from using registers instead of writing to memory.
- ▶ Memory barriers are supplied to avoid reordering

Hardware independent

```
#include <asm/kernel.h>
void barrier(void);
```

Only impacts the behavior of the compiler. Doesn't prevent reordering in the processor: may not work on SMP or out-of-order architectures!

Hardware dependent

```
#include <asm/system.h>
void rmb(void);
void wmb(void);
void mb(void);
```

Safe on all architectures!



Accessing I/O ports – The “old” style

Functions to read/write bytes, word and longs 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);
```

Not usable with `ioport_map()`:

those functions need an I/O port number, not a pointer to memory.

Perfectly fine if doing only PIO



Accessing MMIO devices – The “old” style

- ▶ 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:

```
unsigned read[bwl](void *addr);  
void write[bwl](unsigned a, void *addr);
```
- ▶ Perfectly fine if doing only MMIO. When doing mixed PIO and MMIO, the `ioread/iowrite` family of functions can be used everywhere instead of `read/write`



Accessing I/O memory – The “new” style

- ▶ Thanks to `ioremap()`, it is possible to mix PIO and MMIO in a transparent way. The following functions can be used to access memory areas returned by `ioremap()` or `ioremap_nocache()`:

```
unsigned int ioread8(void *addr); (same for 16 and 32)
```

```
void iowrite8(u8 value, void *addr); (same for 16 and 32)
```

- ▶ To read or write a series of values:

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
```

```
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
```

- ▶ Other useful functions:

```
void memset_io(void *addr, u8 value, unsigned int count);
```

```
void memcpy_fromio(void *dest, void *source, unsigned int count);
```

```
void memcpy_toio(void *dest, void *source, unsigned int count);
```

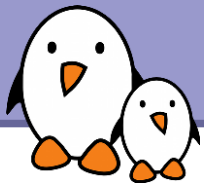
- ▶ Note: many drivers still use old functions instead:

```
readb, readl, readw, writeb, writel, writew, outb, inb, ...
```



/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.
- ▶ Since 2.6.26 (`x86` only, 2.6.32 status): only non-RAM can be accessed for security reasons, unless explicitly configured otherwise (`CONFIG_STRICT_DEVMEM`).



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!





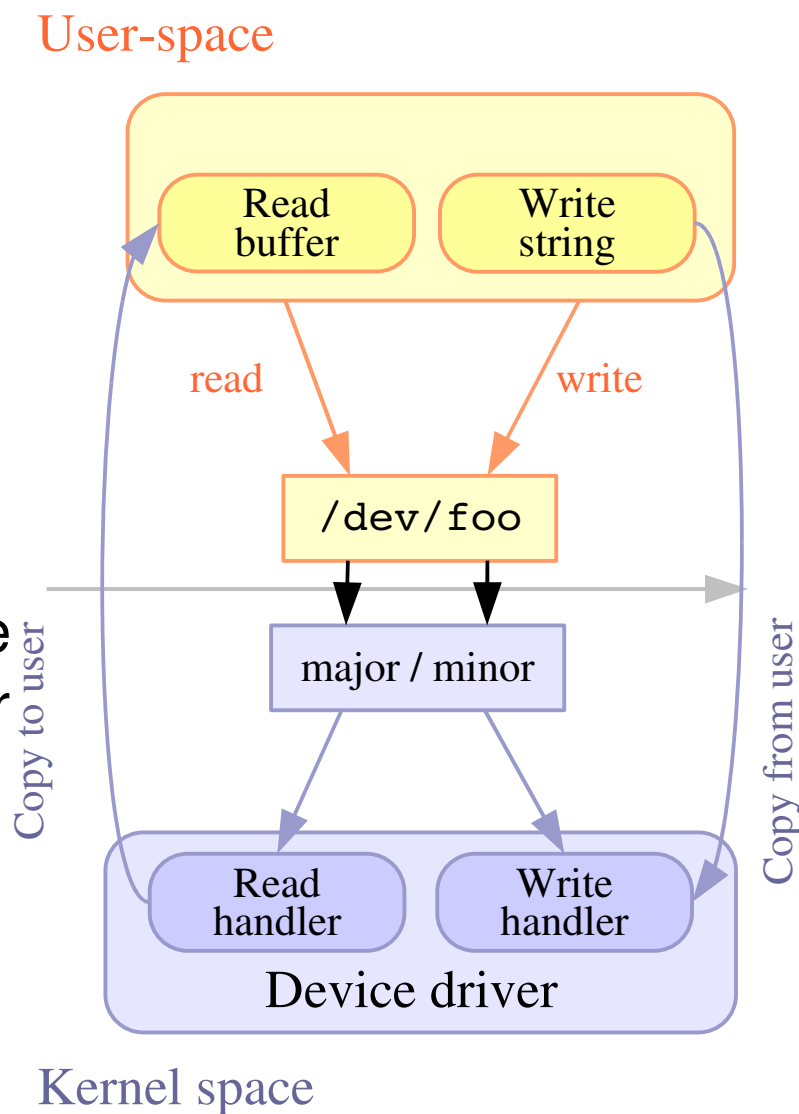
Creating a character driver

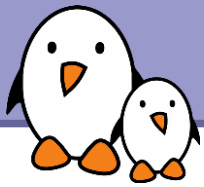
User-space needs

- ▶ 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.





Declaring a character driver

Device number registration

- ▶ Need to register one or more device numbers (major / minor pairs), depending on the number of devices managed by the driver.
- ▶ Need to find free ones!

File operations registration

- ▶ Need to register handler functions called when user space programs access the device files: `open`, `read`, `write`, `ioctl`, `close`...



Information on registered devices

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

Character devices:

```
1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
6 lp
10 misc
13 input
14 sound
...
```

Block devices:

```
1 ramdisk
3 ide0
8 sd
9 md
22 ide1
65 sd
66 sd
67 sd
68 sd
```

Major
number

Registered
name

Can be used to
find free major
numbers



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>

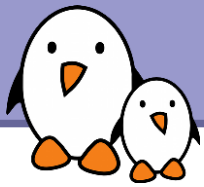
int register_chrdev_region(
    dev_t from,                /* Starting device number */
    unsigned count,           /* Number of device numbers */
    const char *name);        /* Registered name */
```

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.txt`.



File operations (1)

Before registering character devices, you have to define `file_operations` (called *fops*) for the device files.

Here are the main ones:

```
▶ int (*open) (  
    struct inode *, /* Corresponds to the device file */  
    struct file *); /* Corresponds to the open file descriptor */
```

Called when user-space opens the device file.

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

Called when user-space closes the file.



The file structure

Is created by the kernel during the `open` call.
Represents open files.

▶ `mode_t f_mode;`

The file opening mode (`FMODE_READ` and/or `FMODE_WRITE`)

▶ `loff_t f_pos;`

Current offset in the file.

▶ `struct file_operations *f_op;`

Allows to change file operations for different open files!

▶ `struct dentry *f_dentry`

Useful to get access to the inode: `f_dentry->d_inode`.



File operations (2)

```
▶ ssize_t (*read) (  
    struct file *,           /* Open file descriptor */  
    __user char *,           /* User-space buffer to fill up */  
    size_t,                  /* Size of the user-space buffer */  
    loff_t *);               /* Offset in the open file */
```

Called when user-space reads from the device file.

```
▶ ssize_t (*write) (  
    struct file *,           /* Open file descriptor */  
    __user const char *,     /* User-space buffer to write  
                               to the device */  
    size_t,                  /* Size of the user-space buffer */  
    loff_t *);               /* Offset in the open file */
```

Called when user-space writes to the device file.



Exchanging data with user-space (1)

In driver code, you can't just `memcpy` between an address supplied by user-space and the address of a buffer in kernel-space!

- ▶ Correspond to completely different address spaces (thanks to virtual memory)
- ▶ The user-space address may be swapped out to disk
- ▶ The user-space address may be invalid (user space process trying to access unauthorized data)





Exchanging data with user-space (2)

You must use dedicated functions such as the following ones in your `read` and `write` file operations code:

```
include <asm/uaccess.h>
```

```
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);
```

Make sure that these functions return `0`!

Another return value would mean that they failed.



File operations (3)

```
▶ int (*ioctl) (struct inode *, struct file *,  
               unsigned int, unsigned long);
```

Can be used to send specific commands to the device, which are neither reading nor writing (e.g. changing the speed of a serial port, setting video output format, querying a device serial number...).



File operations (4)

```
▶ int (*mmap) (struct file *,  
               struct vm_area_struct *);
```

Asking for device memory to be mapped
into the address space of a user process.

More in our [mmap section](#).

▶ These were just the main ones:
about 25 file operations can be set, corresponding to all
the system calls that can be performed on open files.



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>



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.



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);
```



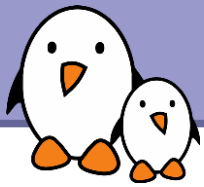
Character device registration (2)

- ▶ 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 */
```

- ▶ 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

Try to report errors with error numbers as accurate as possible! Fortunately, macro names are explicit and you can remember them quickly.

▶ Generic error codes:

```
include/asm-generic/errno-base.h
```

▶ Platform specific error codes:

```
include/asm/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.

Kernel

- Executes the corresponding file operations

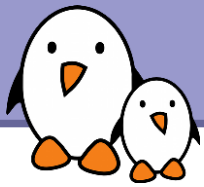
Kernel



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.



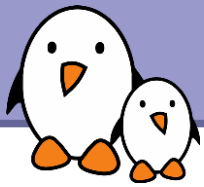
Driver development Processes and scheduling



Processes

A process is an instance of a running program

- ▶ Multiple instances of the same program can be running. Program code (“text section”) memory is shared.
- ▶ Each process has its own data section, address space, processor state, open files and pending signals.
- ▶ The kernel has a separate data structure for each process.



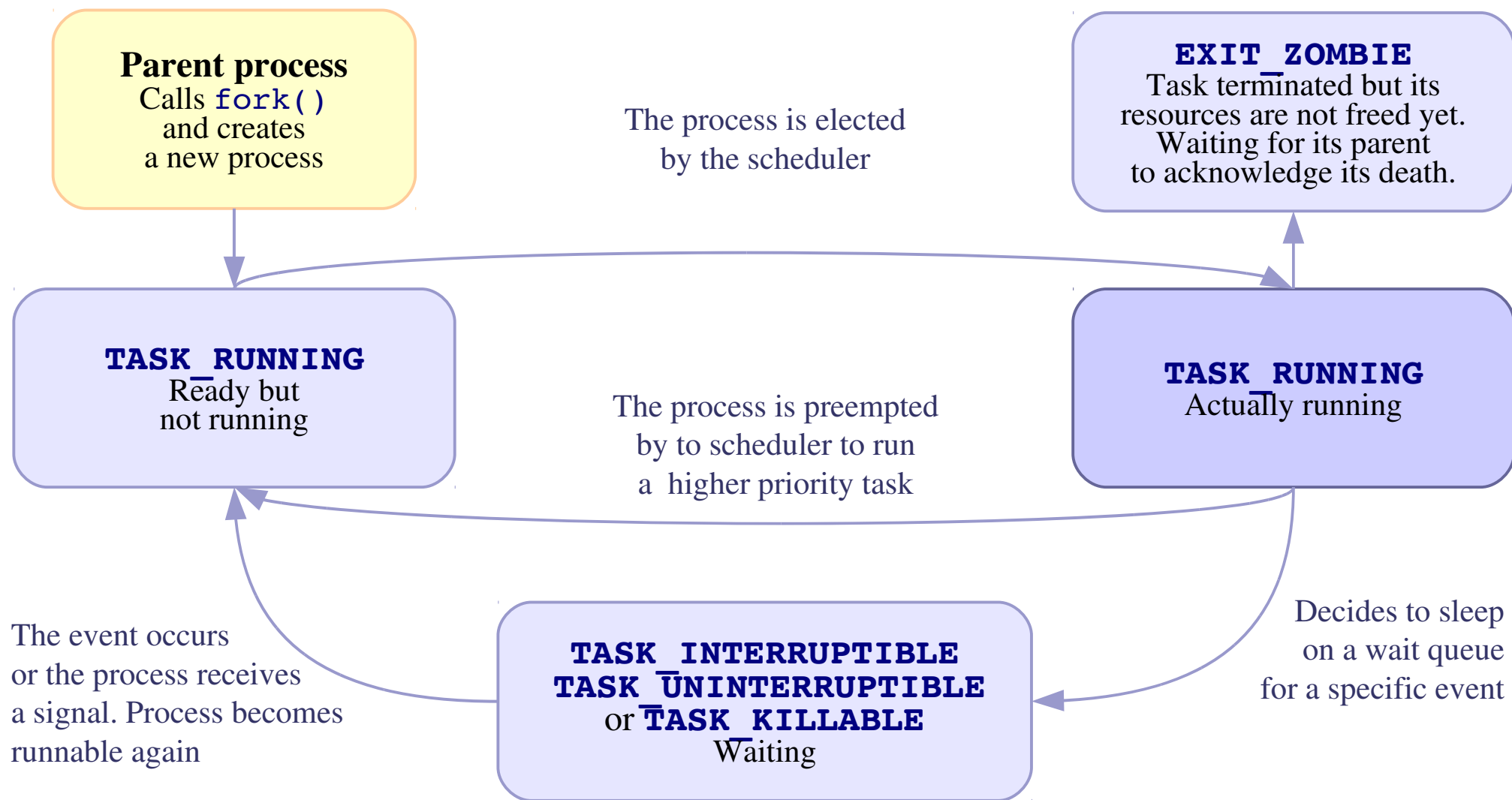
Threads

In Linux, threads are just implemented as processes!

- ▶ New threads are implemented as regular processes, with the particularity that they are created with the same address space, filesystem resources, file descriptors and signal handlers as their parent process.



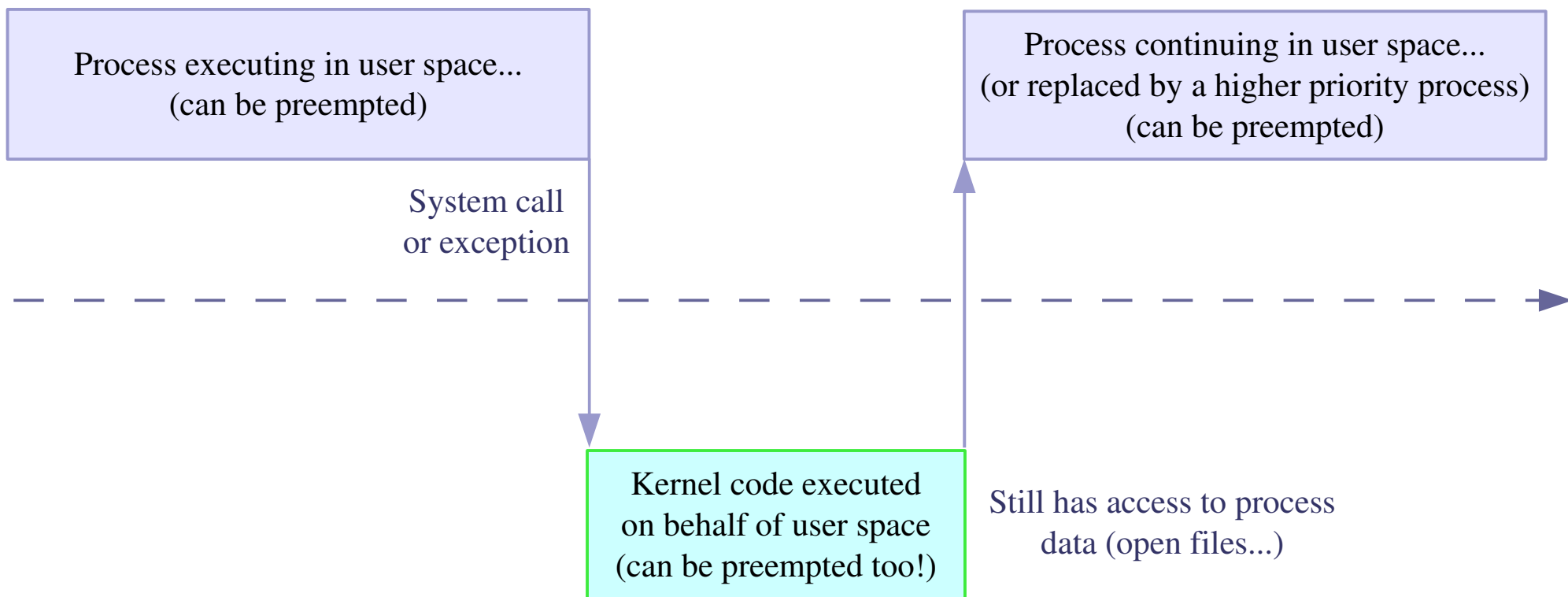
A process life

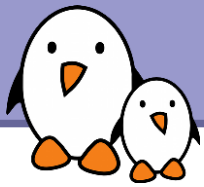




Process context

User space programs and system calls are scheduled together





Kernel threads

- ▶ The kernel does not only react from user-space (system calls, exceptions) or hardware events (interrupts). It also runs its own processes.
- ▶ Kernel threads are standard processes scheduled and preempted in the same way (you can view them with `top` or `ps`!) They just have no special address space and usually run forever.
- ▶ Kernel thread examples:
 - ▶ `pdflush`: regularly flushes “dirty” memory pages to disk (file changes not committed to disk yet).
 - ▶ `migration/<n>`: Per CPU threads to migrate processes between processors, to balance CPU load between processors.



Process priorities

Regular processes

- ▶ Priorities from **-20** (maximum) to **19** (minimum)
- ▶ Only **root** can set negative priorities
(**root** can give a negative priority to a regular user process)
- ▶ Use the **nice** command to run a job with a given priority:
nice -n <priority> <command>
- ▶ Use the **renice** command to change a process priority:
renice <priority> -p <pid>



Timeslices

The scheduler prioritizes high priority processes by giving them a bigger timeslice.

- ▶ Initial process timeslice: parent's timeslice split in 2 (otherwise process would cheat by forking).
- ▶ Minimum priority: 5 ms or 1 jiffy (whichever is larger)
- ▶ Default priority in jiffies: 100 ms
- ▶ Maximum priority: 800 ms

Note: actually depends on [HZ](#).

See [kernel/sched.c](#) for details.



Real-time priorities

Processes with real-time priority can be started by `root` using the POSIX API

- ▶ Available through `<sched.h>` (see `man sched.h` for details)
- ▶ 100 real-time priorities available
- ▶ `SCHED_FIFO` scheduling class:
The process runs until completion unless it is blocked by an I/O, voluntarily relinquishes the CPU, or is preempted by a higher priority process.
- ▶ `SCHED_RR` scheduling class:
Difference: the processes are scheduled in a Round Robin way. Each process is run until it exhausts a max time quantum. Then other processes with the same priority are run, and so and so...



When is scheduling run?

Each process has a `need_resched` flag which is set:

- ▶ After a process exhausted its timeslice.
- ▶ After a process with a higher priority is awakened.

This flag is checked (possibly causing the execution of the scheduler)

- ▶ When returning to user-space from a system call
- ▶ When returning from interrupts (including the cpu timer), when kernel preemption is enabled.

Scheduling also happens when kernel code explicitly runs `schedule()` or executes an action that sleeps.

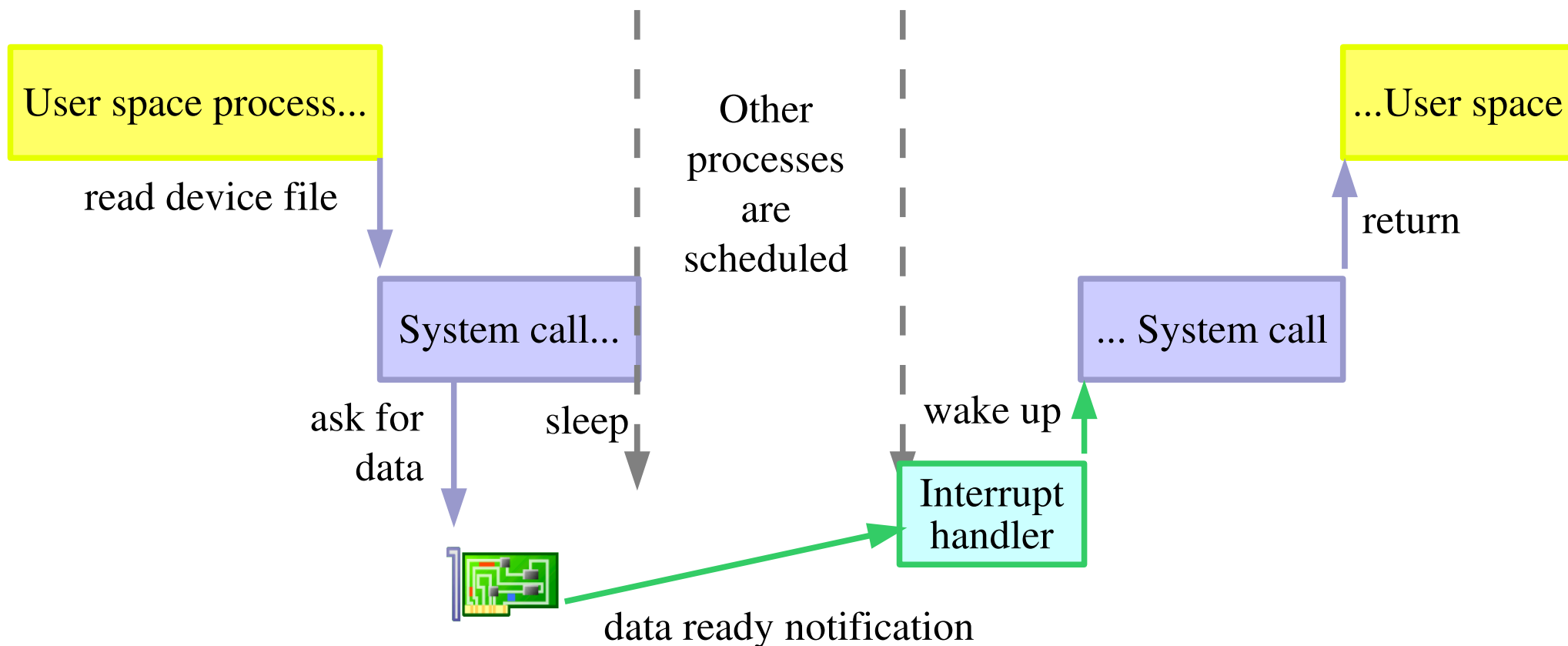


Driver development Sleeping



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

► Static queue declaration

```
DECLARE_WAIT_QUEUE_HEAD (module_queue);
```

► Or dynamic queue declaration

```
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!)

- ▶ `wait_event_killable(queue, condition);` (Since Linux 2.6.25)

Can be interrupted, but only by a “fatal” signal (`SIGKILL`)

- ▶ `wait_event_interruptible(queue, condition);`

Can be interrupted by any signal

- ▶ `wait_event_timeout(queue, condition, timeout);`

Also stops sleeping when the task is woken up and the timeout expired.

- ▶ `wait_event_interruptible_timeout(queue, condition, timeout);`

Same as above, interruptible.



How to sleep - Example

From `drivers/ieee1394/video1394.c`

```
wait_event_interruptible(  
    d->waitq,  
    (d->buffer_status[v.buffer]  
     == VIDEO1394_BUFFER_READY)  
);
```

```
if (signal_pending(current))  
    return -EINTR;
```



Currently running process



Waking up!

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

- ▶ `wake_up(&queue);`

Wakes up all the waiting processes on the given queue

- ▶ `wake_up_interruptible(&queue);`

Wakes up only the processes waiting in an interruptible sleep on the given queue



Sleeping and waking up - implementation

The scheduler doesn't keep evaluating the sleeping condition!

▶ `wait_event_interruptible(&queue, condition);`

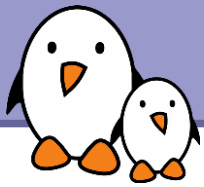
The process is put in the `TASK_INTERRUPTIBLE` state.

▶ `wake_up_interruptible(&queue);`

For all processes waiting in `queue`, `condition` is evaluated.

When it evaluates to true, the process is put back to the `TASK_RUNNING` state, and the `need_resched` flag for the current process is set.

This way, several processes can be woken up at the same time.



Driver development Interrupt management



Interrupt handler constraints

- ▶ Not run from a user context:
Can't transfer data to and from user space
(need to be done by system call handlers)
- ▶ 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.



Registering an interrupt handler (1)

Defined in `include/linux/interrupt.h`

- ▶ `int request_irq(`
 `unsigned int irq,`
 `irq_handler_t handler,`
 `unsigned long irq_flags,`
 `const char * devname,`
 `void *dev_id);`
 - Returns 0 if successful
 - Requested irq channel
 - Interrupt handler
 - Option mask (see next page)
 - Registered name
 - 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.



When to register the handler

- ▶ Either at driver initialization time:
consumes lots of IRQ channels!
- ▶ Or at device open time (first call to the `open` file operation):
better for saving free IRQ channels.
Need to count the number of times the device is opened, to
be able to free the IRQ channel when the device is no
longer in use.



Information on installed handlers

`/proc/interrupts`

```

          CPU0
0:         5616905      XT-PIC  timer # Registered name
1:           9828      XT-PIC  i8042
2:              0      XT-PIC  cascade
3:        1014243      XT-PIC  orinoco_cs
7:           184       XT-PIC  Intel 82801DB-ICH4
8:              1      XT-PIC  rtc
9:              2      XT-PIC  acpi
11:        566583      XT-PIC  ehci_hcd, uhci_hcd,
uhci_hcd, uhci_hcd, yenta, yenta, radeon@PCI:1:0:0
12:           5466      XT-PIC  i8042
14:        121043      XT-PIC  ide0
15:        200888      XT-PIC  ide1
NMI:              0      Non Maskable Interrupts
ERR:              0      Spurious interrupt count
```

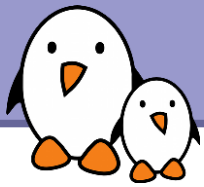


Total number of interrupts

```
cat /proc/stat | grep intr
```

```
intr 8190767 6092967 10377 0 1102775 5 2 0 196 ...
```

Total number of interrupts	IRQ1 total	IRQ2 total	IRQ3 ...
-------------------------------	---------------	---------------	-------------



The interrupt handler's job

- ▶ Acknowledge the interrupt to the device
(otherwise no more interrupts will be generated)
- ▶ 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);
```



Interrupt handler prototype

```
irqreturn_t (*handler) (  
    int,                // irq number of the current interrupt  
    void *dev_id        // Pointer used to keep track  
                        // of the corresponding device.  
                        // Useful when several devices  
                        // are managed by the same module  
);
```

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.



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.

Best implemented by *tasklets* (also called *soft irqs*).



Top half and bottom half processing (2)

- ▶ 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.

By default, tasklets are executed right after all top halves (hard irqs)



Interrupt management fun

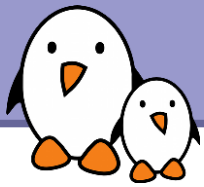
- ▶ In a training lab, somebody forgot to unregister a handler on a shared interrupt line in the module exit function.



Why did his kernel oops with a segmentation fault at module unload?

Answer...

- ▶ In a training lab, somebody freed the timer interrupt handler by mistake (using the wrong irq number). The system froze. Remember the kernel is not protected against itself!



Interrupt management summary

Device driver

- ▶ When the device file is first open, 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.



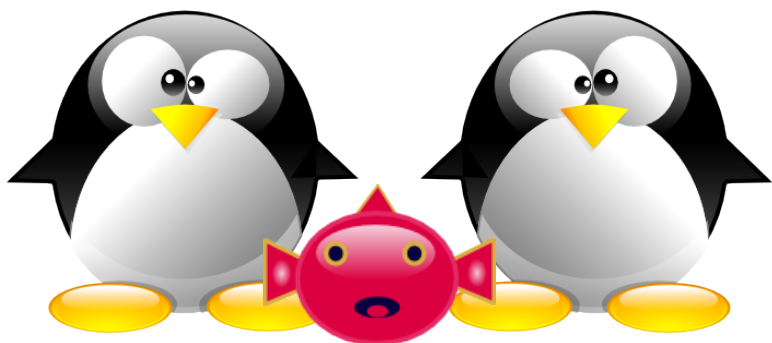
Practical lab – Interrupts

- ▶ 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.



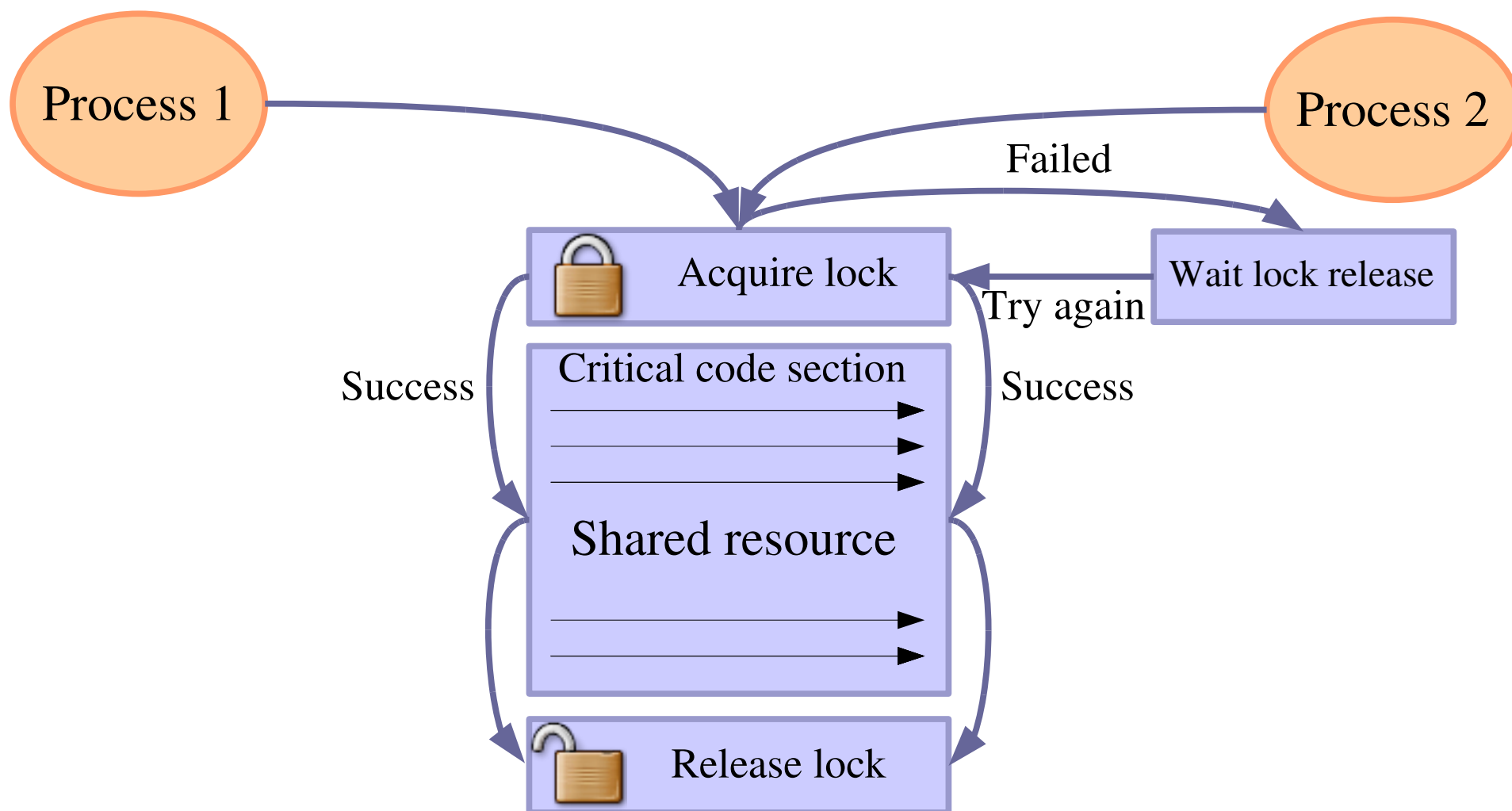
Avoiding concurrency issues

- ▶ 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.
- ▶ 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.



Reader / writer semaphores

Allow shared access by unlimited readers, or by only 1 writer. Writers get priority.

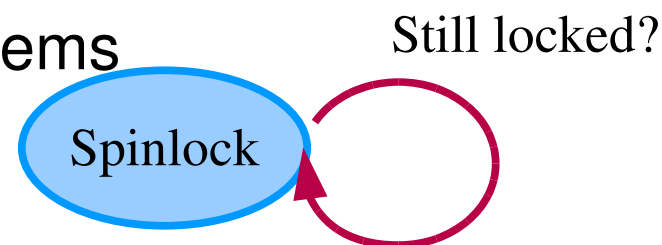
```
void init_rwsem (struct rw_semaphore *sem);  
  
void down_read (struct rw_semaphore *sem);  
int down_read_trylock (struct rw_semaphore *sem);  
int up_read (struct rw_semaphore *sem);  
  
void down_write (struct rw_semaphore *sem);  
int down_write_trylock (struct rw_semaphore *sem);  
int up_write (struct rw_semaphore *sem);
```

Well suited for rare writes, holding the semaphore briefly. Otherwise, readers get *starved*, waiting too long for the semaphore to be released.



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.





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.

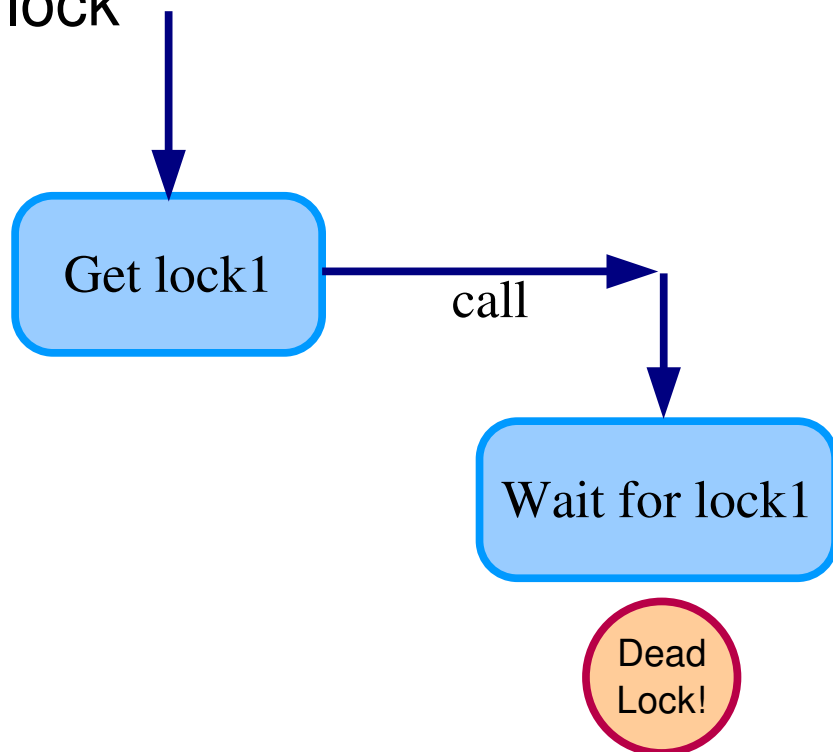
Note that reader / writer spinlocks also exist.



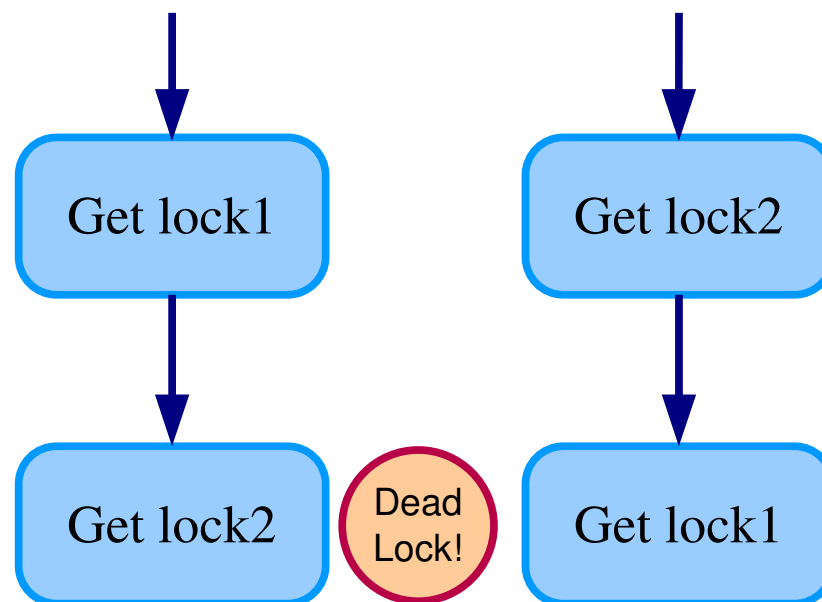
Deadlock situations

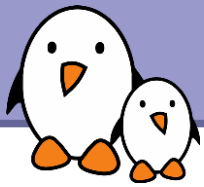
They can lock up your system. Make sure they never happen!

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);
```

- ▶ Similar 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 resources





Embedded Linux driver development



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 or `/var/log/messages` 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 `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

Linux also has 3-finger-keys to save your work ;-)

- ▶ Allows to run multiple debug / rescue commands even when the kernel seems to be in deep trouble. Example commands:
 - ▶ `[ALT][SysRq][d]`: kills all processes, except init.
 - ▶ `[ALT][SysRq][n]`: makes RT processes nice-able.
 - ▶ `[ALT][SysRq][s]`: attempts to sync all mounted filesystems.
 - ▶ `[ALT][SysRq][b]`: immediately reboot without syncing and unmounting.
- ▶ Typical combination: `[ALT][SysRq][s]`
and then `[ALT][SysRq][b]`
- ▶ Detailed in `Documentation/sysrq.txt`



Debugging with gdb

- ▶ If you execute the kernel from a debugger on the same machine, this will interfere with the kernel behavior.
- ▶ However, you can access the current kernel state with `gdb`:
`gdb /usr/src/linux/vmlinux /proc/kcore`
 uncompressed kernel kernel address space
- ▶ You can access kernel structures, follow pointers...
(read only!)
- ▶ Requires the kernel to be compiled with
`CONFIG_DEBUG_INFO` (Kernel hacking section)



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][SysRq][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/>

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 investigate 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])/1000)
        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

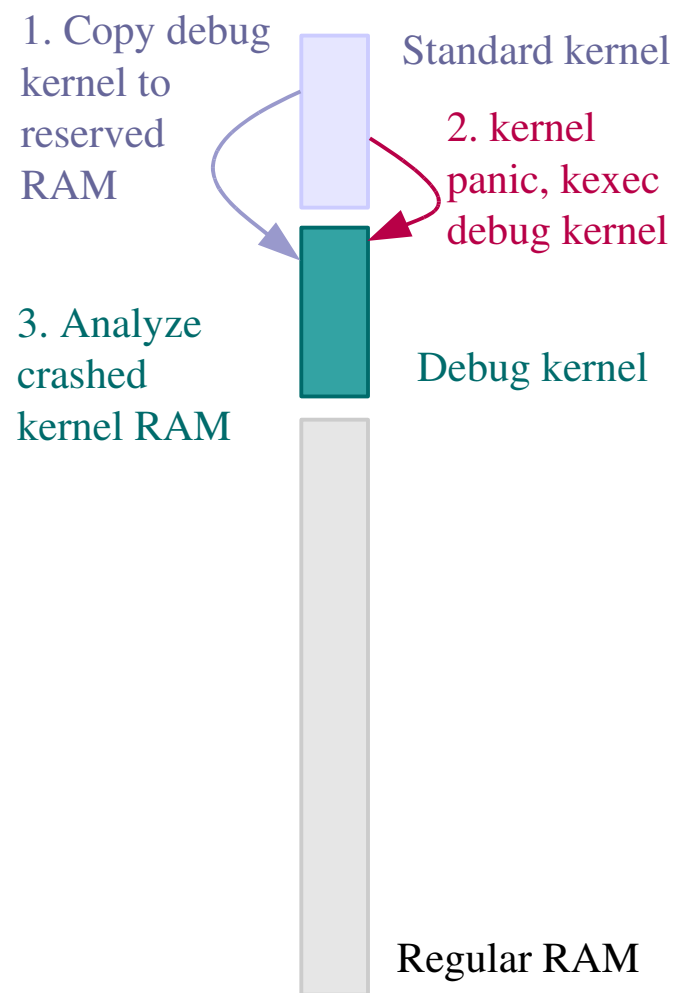
probe kernel.function ("vfs_read")
{
    dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
    inode_nr = $file->f_dentry->d_inode->i_ino
    printf ("%s(%d) %s 0x%x/%d\n",
            execname(), pid(), probefunc(), dev_nr, inode_nr)
}
```

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.

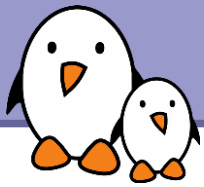




Kernel markers

- ▶ 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.

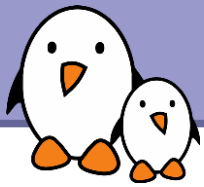
See http://en.wikipedia.org/wiki/Kernel_marker



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>



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	ffffff00	---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!

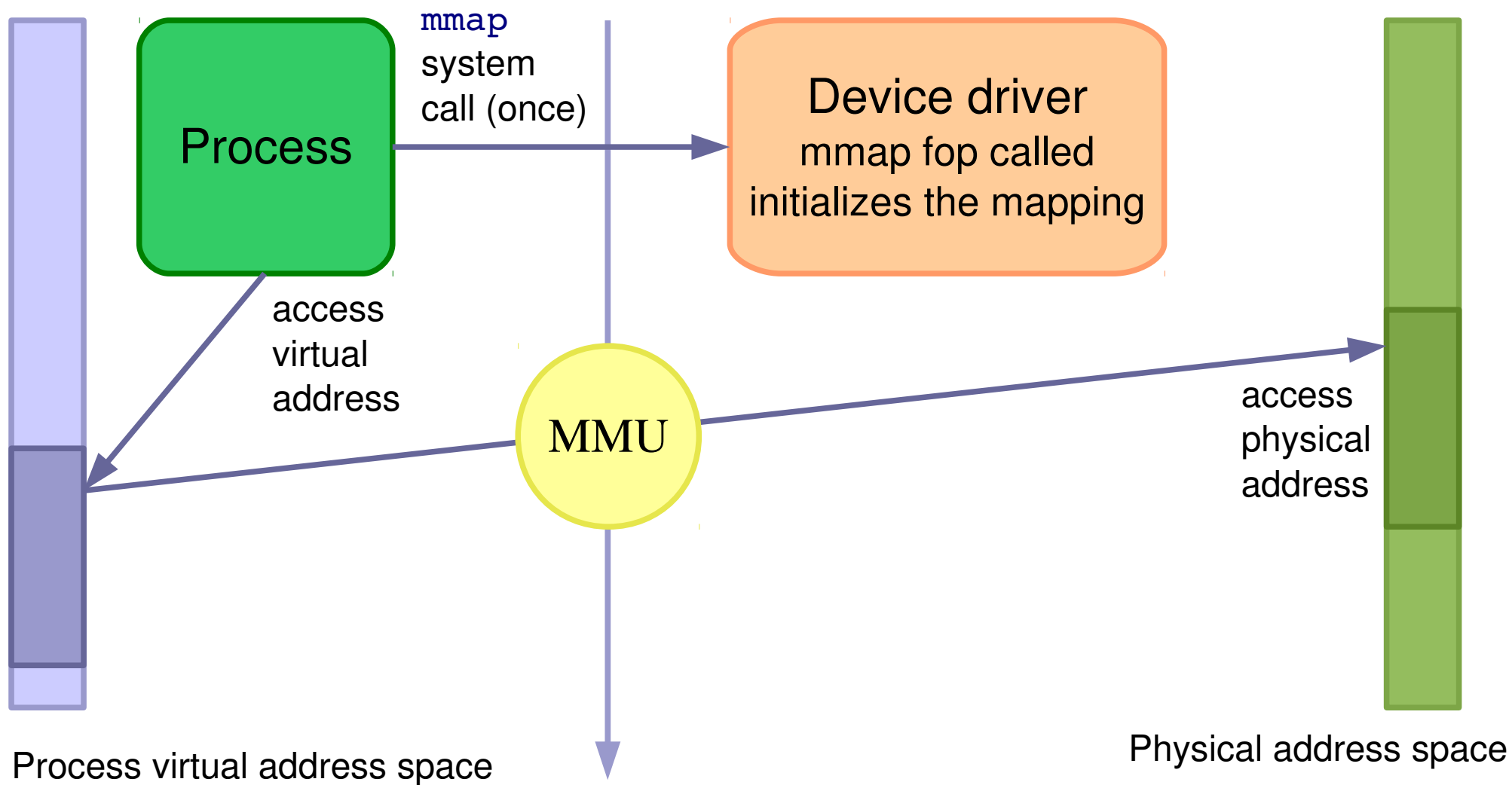
`X server` example (maps excerpt)

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;  
}
```

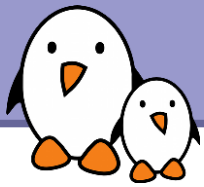



devmem2

<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.

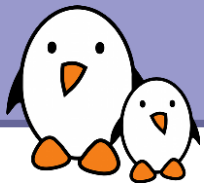


mmap summary

- ▶ 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 Device Model



Unified device model

- ▶ The 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, categorize them by classes



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, ISA, etc.
- ▶ It is responsible for
 - ▶ Registering the bus type
 - ▶ Allowing registration of adapter drivers (USB controllers, I2C adapters, etc.), able of detecting the connected devices
 - ▶ Allowing 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



List of device identifiers

- ▶ Depending on the bus type, the method for binding a device to a driver is different. For many buses, it is based on unique identifiers
- ▶ The device driver defines a table with the list of device identifiers it is able to manage :

```
static const struct pci_device_id rhine_pci_tbl[] = {  
    { 0x1106, 0x3043, PCI_ANY_ID, PCI_ANY_ID, },    /* VT86C100A */  
    { 0x1106, 0x3053, PCI_ANY_ID, PCI_ANY_ID, },    /* VT6105M */  
    { }        /* terminate list */  
};  
MODULE_DEVICE_TABLE(pci, rhine_pci_tbl);
```

Code on this slide and the next slides are taken
from the [via-rhine](#) driver in [drivers/net/via-rhine.c](#)



Defining the driver

- ▶ The device driver defines a driver structure, usually specialized by the bus infrastructure (`pci_driver`, `usb_driver`, etc.)
- ▶ The structure points to : the device table, a probe function, called when a device is detected and various other callbacks

```
static struct pci_driver rhine_driver = {
    .name          = DRV_NAME,
    .id_table       = rhine_pci_tbl,
    .probe          = rhine_init_one,
    .remove         = __devexit_p(rhine_remove_one),
#ifdef CONFIG_PM
    .suspend        = rhine_suspend,
    .resume         = rhine_resume,
#endif /* CONFIG_PM */
    .shutdown       = rhine_shutdown,
};
```



Registering the driver

- ▶ In the module initialization function, the driver is registered to the bus infrastructure, in order to let the bus know that the driver is available to handle devices.

```
static int __init rhine_init(void)
{
    [...]
    return pci_register_driver(&rhine_driver);
}
static void __exit rhine_cleanup(void)
{
    pci_unregister_driver(&rhine_driver);
}
```

- ▶ If a new PCI device matches one of the identifiers of the table, the `probe()` method of the PCI driver will get called.



Probe method

- ▶ The `probe()` method receives as argument a structure describing the device, usually specialized by the bus infrastructure (`pci_dev`, `usb_device`, etc.)
- ▶ This function is responsible for
 - ▶ Initializing the device, mapping the 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.

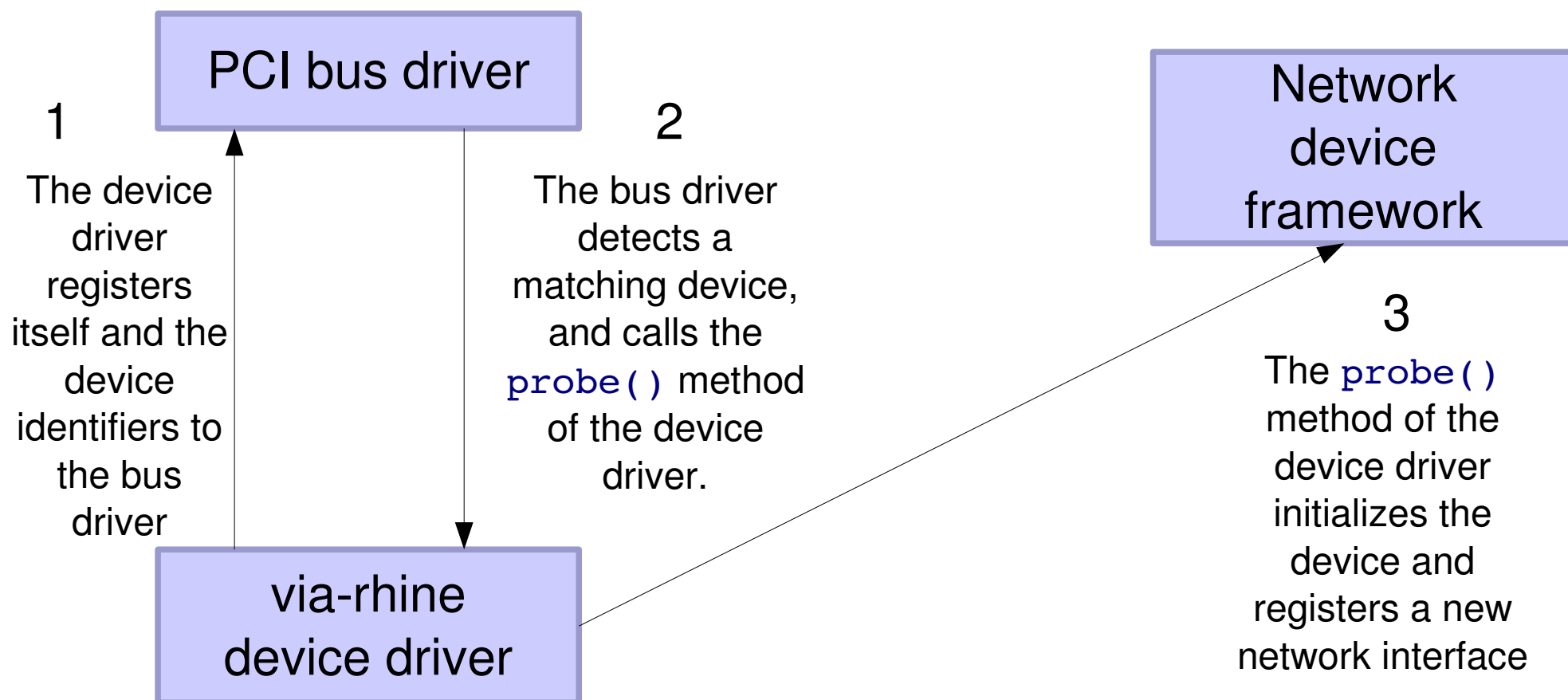


Device driver (5)

```
static int __devinit rhine_init_one(struct pci_dev *pdev,
                                   const struct pci_device_id *ent)
{
    struct net_device *dev;
    [...]
    rc = pci_enable_device(pdev);
    [...]
    pioaddr = pci_resource_start(pdev, 0);
    memaddr = pci_resource_start(pdev, 1);
    [...]
    dev = alloc_etherdev(sizeof(struct rhine_private));
    [...]
    SET_NETDEV_DEV(dev, &pdev->dev);
    [...]
    rc = pci_request_regions(pdev, DRV_NAME);
    [...]
    ioaddr = pci_iomap(pdev, bar, io_size);
    [...]
    rc = register_netdev(dev);
    [...]
}
```



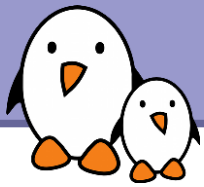
Global architecture





sysfs

- ▶ The bus, device, drivers, etc. structures are internal to the kernel
- ▶ The `sysfs` virtual filesystem provides 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);
    [...]
}
```



Instantiation of the platform device

- In the board-specific code, the platform devices are instantiated (`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,
    }
};
```

- The match between the device and the driver is made using the name. It must be unique amongst drivers !



Instantiation of the platform device (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 mxlads_init(void)  
{  
    [...]  
    platform_add_devices(devices, ARRAY_SIZE(devices));  
}
```

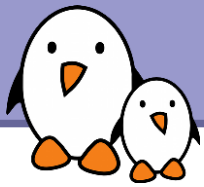



I/O resources

- ▶ Each platform device is associated with a set of I/O resources, referenced in the `platform_device` structure

```
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,
    },
};
```

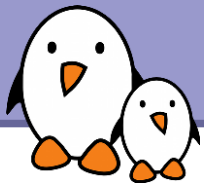
- ▶ It allows the driver to be independent of the I/O addresses, IRQ numbers ! See `imx_uart2_device` for another device using the same platform driver.



Inside the platform driver

- ▶ 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);
```



sysfs tools

<http://linux-diag.sourceforge.net/Sysfsutils.html>

- ▶ `libsysfs` - The library's purpose is to provide a consistent and stable interface for querying system device information exposed through `sysfs`.
- ▶ `systool` - A utility built upon `libsysfs` that lists devices by bus, class, and topology.



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!



Annexes

Quiz answers



Quiz answers


► Interrupt handling

Q: Why did the kernel segfault at module unload (forgetting to unregister a handler in a shared interrupt line)?

A: Kernel memory is allocated at module load time, to host module code. This memory is freed at module unload time. If you forget to unregister a handler and an interrupt comes, the cpu will try to jump to the address of the handler, which is in a freed memory area.
Crash!



Related documents



Free Electrons

Embedded Freedom

HOME DEVELOPMENT SERVICES TRAINING DOCS COMMUNITY COMPANY BLOG

Recent blog posts

ELC Europe in Grenoble

Free Electrons at ELC

Linux kernel 2.6.29 - New features for embedded users

The Buildroot project begins a new life

FOSDEM 2009 videos

USB-Ethernet device for Linux

Program for Embedded Linux Conference 2009 announced

Public session changes


Real hardware in our training sessions

Call for presentations for the LSM embedded track

Docs

Most of the below documents are presentations used in our [training sessions](#), or in technical conferences.

License

 All our documents are available under the terms of the [Creative Commons Attribution-ShareAlike 3.0 license](#). This essentially means that you are free to download, distribute and even modify them, provided you mention us as the original authors and that you share these documents under the same conditions.

Linux kernel

- [Embedded Linux kernel and driver development](#)
- [New features in Linux 2.6](#) (since 2.6.10)
- [Kernel initialization](#)
- [Porting Linux to new hardware](#)
- [Power management in Linux](#)
- [Linux PCI drivers](#)
- [Block device drivers](#)
- [Linux USB drivers](#)
- [DMA](#)

Architecture specific documents

- [ARM Linux specifics](#)
- [Linux on TI OMAP processors](#)

Embedded Linux system development

- [Embedded Linux system development](#)
- [Real time in embedded Linux systems](#)
- [Block filesystems](#)
- [Flash filesystems](#)
- [Free software development tools](#)
- [The U-boot bootloader](#)
- [The GRUB bootloader](#)
- [The blob bootloader](#)
- [Hotplugging with udev](#)
- [Introduction to uClinux](#)
- [Java in embedded Linux](#)
- [Embedded Linux optimizations](#)
- [Audio in embedded Linux systems](#)
- [Multimedia in embedded Linux systems](#)
- [Embedded Linux From Scratch... in 40 minutes!](#)
- [Building embedded Linux systems with Buildroot](#)
- [Developing embedded distributions with OpenEmbedded](#)
- [The Scratchbox development environment](#)

Miscellaneous

- [Introduction to the Unix command line](#)
- [SSH](#)
- [Linux virtualization solutions](#) (with an embedded perspective)
- [Advantages of Free Software and Open Source in embedded systems](#)
- [Introduction to GNU/Linux and Free Software](#)

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.

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

Free Electrons

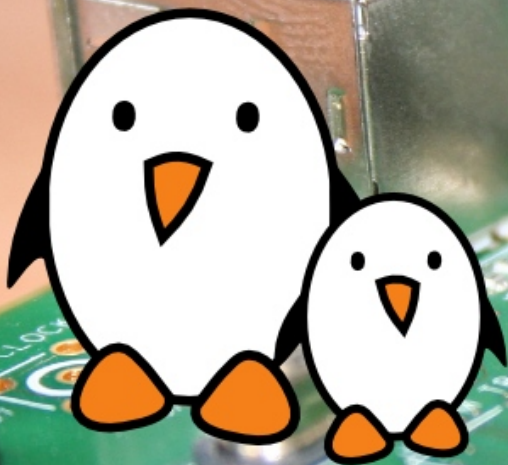
Our services

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



Free Electrons
Embedded Linux Experts

<http://free-electrons.com>