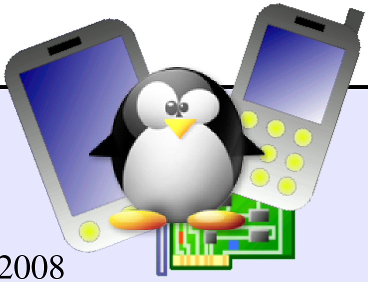
Embedded Linux kernel and driver development

Thomas Petazzoni / Michael Opdenacker

Free Electrons

http://free-electrons.com/

Created with OpenOffice.org 2.x



Rights to copy

creative commons

COMMONS DEED

Attribution – ShareAlike 2.5

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions

Attribution. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

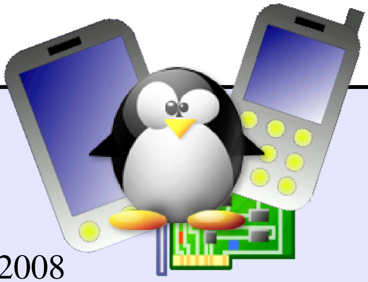
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: http://creativecommons.org/licenses/by-sa/2.5/legalcode

Document sources, updates and translations:
http://free-electrons.com/training/drivers

Corrections, suggestions, contributions and translations are welcome!



Best viewed with...

This document is best viewed with a recent PDF reader or with OpenOffice.org itself!

- Take advantage of internal or external hyperlinks. So, don’t hesitate to click on them! See next page.
- Find pages quickly thanks to automatic search
- Use thumbnails to navigate in the document in a quick way

If you’re reading a paper or HTML copy, you should get your copy in PDF or OpenOffice.org format on http://free-electrons.com/training/drivers!

Hyperlinks in this document

- Links to external sites
  Example: http://kernel.org/

Usable in the PDF and ODP formats

Try them on this page!

- Kernel source files
  Our links let you view them in your browser.
  Example: `kernel/sched.c`

- Kernel source code - Identifiers: functions, macros, type definitions...
  You get access to their definition, implementation and where they are used. This invites you to explore the source by yourself!

click

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

- Table of contents - Directly jump to the corresponding sections.
  Example: `Kernel configuration`

Course prerequisites

Skills to make these lectures and labs profitable

Familiarity with Unix concepts and its command line interface

- Essential to manipulate sources and files
- Essential to understand and debug the system that you build
- You should read http://free-electrons.com/training/intro_unix_linux
  This Unix command line interface training also explains Unix concepts not repeated in this document.

Experience with C programming

- On-line C courses can be found on
  http://dmoz.org/Computers/Programming/Languages/C/Tutorials/

Contents (1)

Kernel overview

- Linux features
- Kernel code
- Kernel subsystems
- Linux versioning scheme and development process
- Legal issues
- Kernel user interface

Contents (2)

Compiling and booting

- Linux kernel sources
- Kernel source managers
- Kernel configuration
- Compiling the kernel
- Overall system startup
- Bootloaders
- Linux device files
- Cross-compiling the kernel

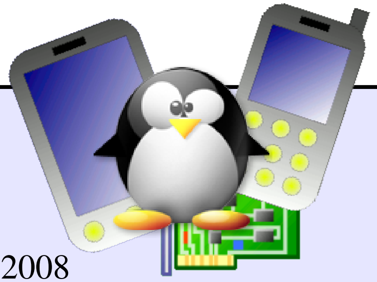
Basic driver development

- Loadable kernel modules
- Module parameters
- Adding sources to the tree

Contents (3)

Driver development

- Memory management
- I/O memory and ports
- Character drivers
- Debugging
- Processes and scheduling
- Sleeping, Interrupt management
- Handling concurrency
- mmap, DMA



Contents (4)

Driver development

- New device model, sysfs
- udev and hotplug

Advice and resources

- Getting help and contributions
- Bug report and patch submission
- References
- Last advice

Contents (5)

Annexes

- Quiz answers
- Kernel sources
- Slab caches and memory pools
- U-boot details
- Grub details
- Init runlevels

Kernel overview

Linux features

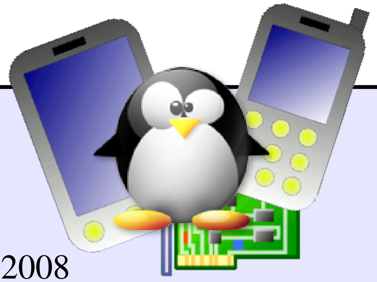
Studied kernel version: 2.6

Linux 2.6

- Linux 2.6.0 was released in December 2003.
- Lots of features and new drivers
  have been added at a quick pace since then.
- It is getting more and more difficult to get support or drivers for recent hardware in 2.4. No community support at all!
- These training slides are compliant with **Linux 2.6.22**.
  It's best to start to learn about the most recent features and updates!

Linux kernel key features

- Portability and hardware support
  Runs on most architectures.
- Scalability
  Can run on super computers as well as on tiny devices
  (4 MB of RAM is enough).
- Compliance to standards and interoperability.
- Exhaustive networking support.
- Security
  It can't hide its flaws. Its code is reviewed by many experts.
- Stability and reliability.
- Modularity
  Can include only what a system needs even at run time.
- Easy to program
  You can learn from existing code.
  Many useful resources on the net.



Supported hardware architectures

- See the `arch/` directory in the kernel sources
- Minimum: 32 bit processors, with or without MMU
- 32 bit architectures (`arch/` subdirectories)
  `alpha`, `arm`, `avr32`, `cris`, `frv`, `h8300`, `i386`, `m32r`, `m68k`, `m68knommu`, `mips`, `parisc`, `powerpc`, `ppc`, `s390`, `sh`, `sparc`, `um`, `v850`, `xtensa`
- 64 bit architectures:
  `ia64`, `mips`, `powerpc`, `sh64`, `sparc64`, `x86_64`
- See `arch/<arch>/Kconfig`, `arch/<arch>/README`, or `Documentation/<arch>/` for details

Kernel overview

Kernel code

Implemented in C

- 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,
  and critical library routines.

See http://www.tux.org/lkml/#s15-3
for reasons for not using C++
(main reason: the kernel requires efficient code).

Programming languages in the kernel sources

Linux 2.6.22 report by SLOCCount (http://dwheeler.com/sloccount/)

```
Totals grouped by language (dominant language first):

ansic:       5215716 (95.78%)
asm:          216350 (3.97%)
perl:           4215 (0.08%)
yacc:           2637 (0.05%)
sh:             2233 (0.04%)
cpp:            2129 (0.04%)
lex:            1510 (0.03%)
python:          331 (0.01%)
lisp:            218 (0.00%)
awk:              96 (0.00%)
```

Compiled with GNU C

- Need GNU C extensions to compile the kernel.
  So, you cannot use any ANSI C compiler!
  You can also use the Intel and Marvell compilers (only on their respective platforms) which identify themselves as a GNU compiler.
- Some GNU C extensions used in the kernel:
  - Inline C functions
  - Inline assembly
  - Structure member initialization in any order (also in ANSI C99)
  - Branch annotation (see next page)
- Requires at least gcc 3.2.
  See `Documentation/Changes` in kernel sources.

Help gcc to optimize your code!

- Use the `likely` and `unlikely` statements (`include/linux/compiler.h`)
- Example:

```
if (unlikely(err)) {
     ...
}
```

- The GNU C compiler will make your code faster for the most likely case.

Used in many places in kernel code!

Don't forget to use these statements!

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()` ...

Managing endianism

Linux supports both little and big endian architectures

- Each architecture defines __BIG_ENDIAN or __LITTLE_ENDIAN in <asm/byteorder.h>
  Can be configured in some platforms supporting both.

- To make your code portable, the kernel offers conversion macros (that do nothing when no conversion is needed). Most useful ones:

```
u32 cpu_to_be32(u32);  // CPU byte order to big endian
u32 cpu_to_le32(u32);  // CPU byte order to little endian
u32 be32_to_cpu(u32);  // Big endian to CPU byte order
u32 le32_to_cpu(u32);  // Little endian to CPU byte order
```

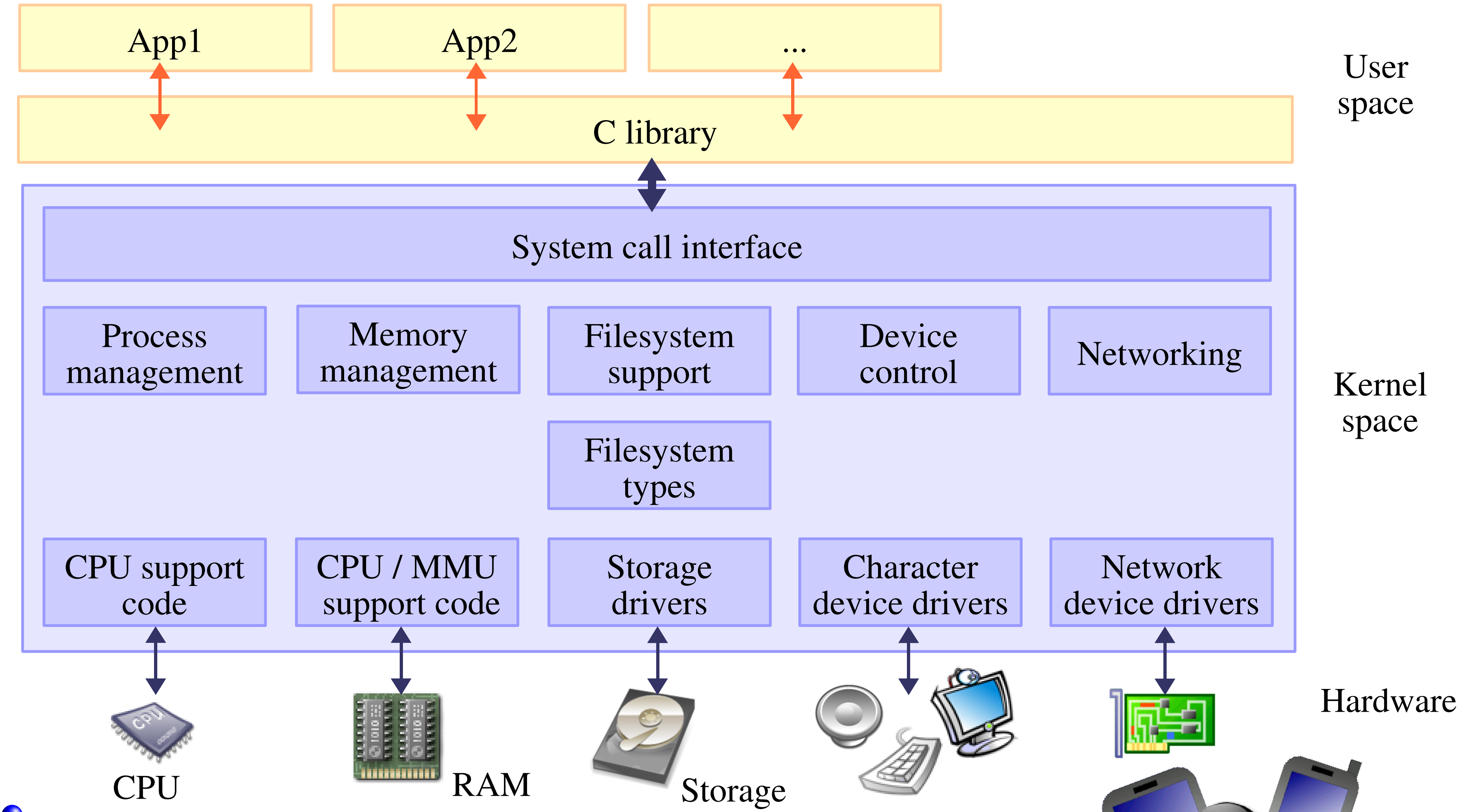
Kernel coding guidelines

- Never use floating point numbers in kernel code. Your code may be run on a processor without a floating point unit (like on `arm`). Floating point can be emulated by the kernel, but this is very slow.
- Define all symbols as static, except exported ones (to avoid namespace pollution)
- See `Documentation/CodingStyle` for more guidelines

Kernel overview

Kernel subsystems

Kernel architecture



App1

App2

...

C library

User space

System call interface

Process management

Memory management

Filesystem support

Device control

Networking

Filesystem types

Kernel space

CPU support code

CPU / MMU support code

Storage drivers

Character device drivers

Network device drivers

CPU

RAM

Storage

Hardware

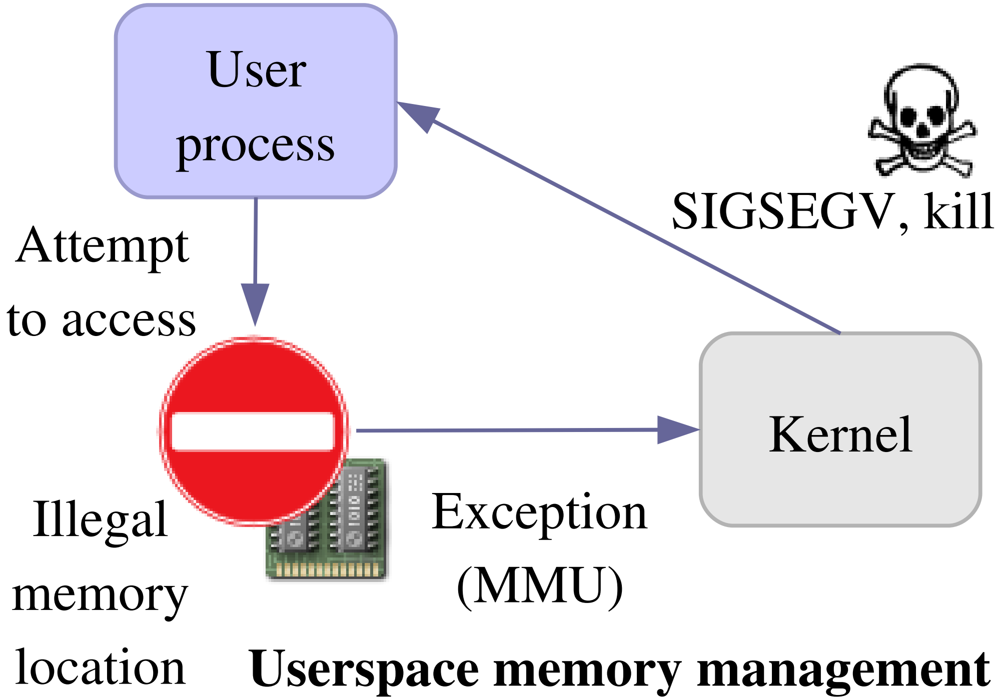
Kernel memory constraints

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).



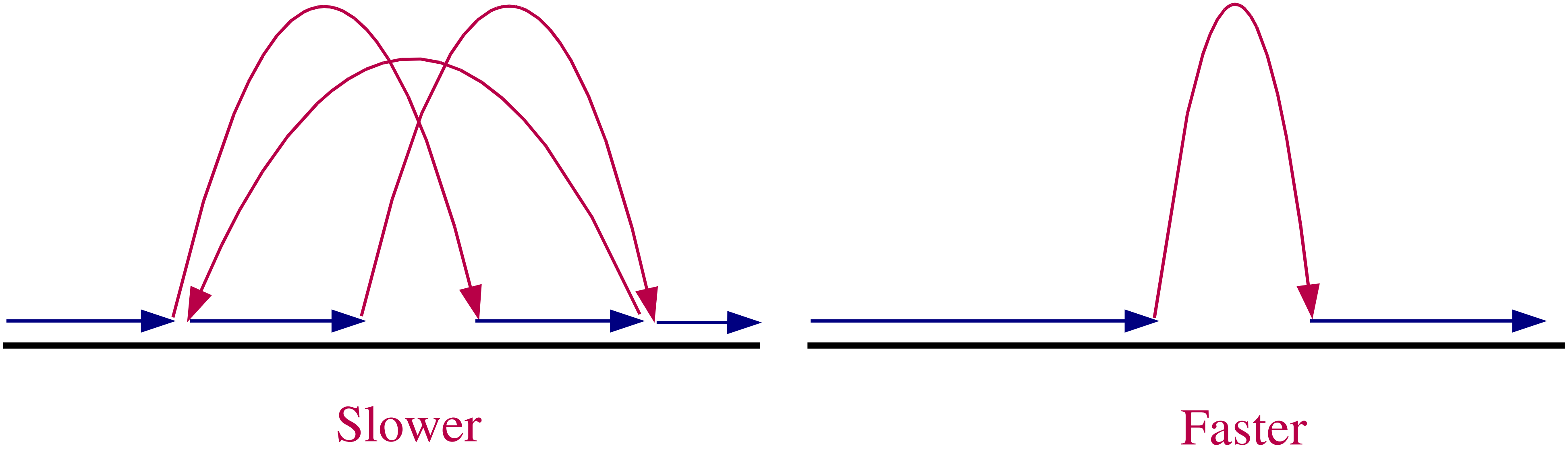
Userspace memory management

Used to implement:

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

I/O schedulers

- Mission of I/O schedulers: re-order reads and writes to disk to minimize disk head moves (time consuming!)



- Not needed in embedded systems with no hard disks (data access time independent of location on flash storage) Build your kernel with no-op I/O scheduler then!

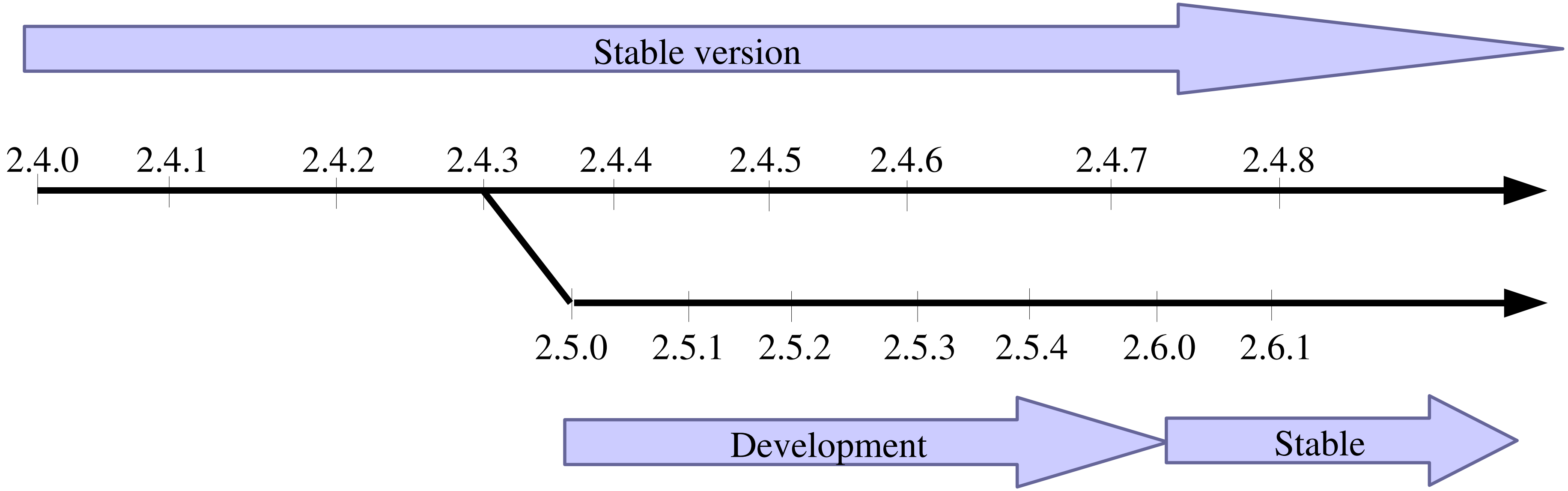
Kernel overview

Linux versioning scheme and development process

Until 2.6 (1)

- One stable major branch every 2 or 3 years
  - Identified by an even middle number
  - Examples: `1.0, 2.0, 2.2, 2.4`
- One development branch to integrate new functionalities and major changes
  - Identified by an odd middle number
  - Examples: `2.1, 2.3, 2.5`
  - After some time, a development version becomes the new base version for the stable branch
- Minor releases once in while: `2.2.23, 2.5.12`, etc.

Until 2.6 (2)



Note: in reality, many more minor versions exist inside the stable and development branches

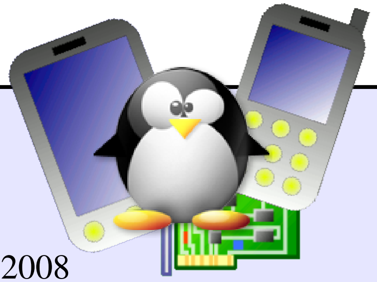
Changes since Linux 2.6 (1)

- Since `2.6.0`, kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make major changes in existing subsystems.
- Opening a new Linux `2.7` (or `2.9`) development branch will be required only when Linux `2.6` is no longer able to accommodate key features without undergoing traumatic changes.
- Thanks to this, more features are released to users at a faster pace.

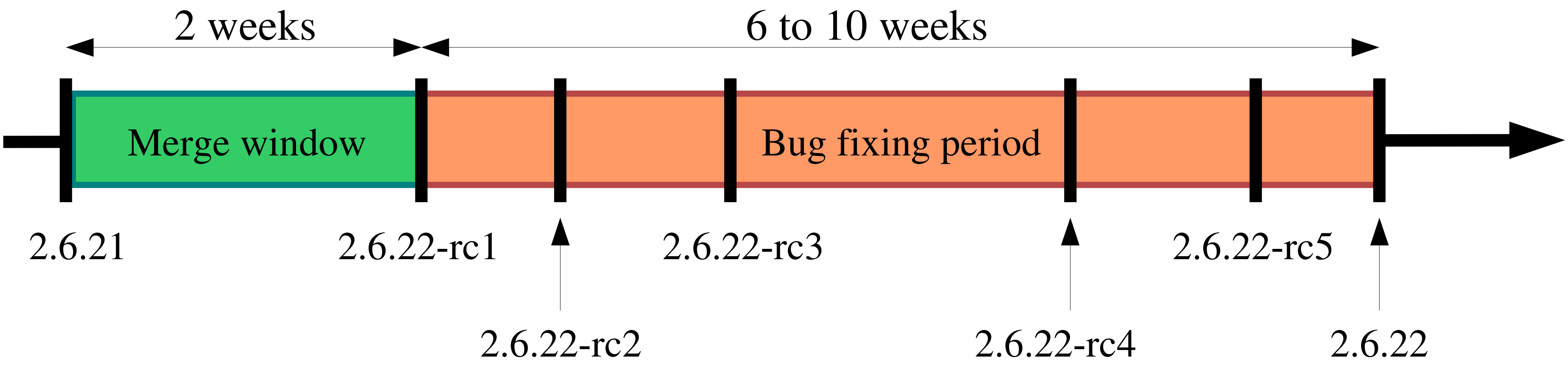
Changes since Linux 2.6 (2)

Since 2.6.14, the kernel developers agreed on the following development model

- After the release of a `2.6.x` version, a two-weeks merge window opens, during which major additions are merged
- The merge window is closed by the release of test version `2.6.(x+1)-rc1`
- The bug fixing period opens, for six to ten weeks
- At regular intervals during the bug fixing period, `2.6.(x+1)-rcY` test versions are released
- When considered sufficiently stable, kernel `2.6.(x+1)` is released, and the process starts again

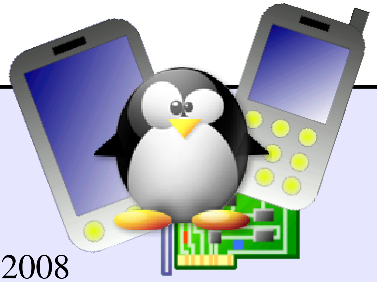


Merge and bug fixing windows



No stable Linux internal API (1)

- 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.
- The internal kernel API can now 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.
- 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!



No stable Linux internal API (2)

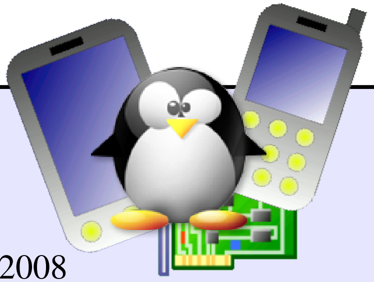
USB example

- 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

More stability for the 2.6 kernel tree

- Issue: security fixes only released for last (or last two) stable kernel versions (like 2.6.16 and 2.6.17), and of course by distributions for the exact version that you're using.
- Some people need to have a recent kernel, but with long term support for security updates.
- That's why Adrian Bunk proposed to maintain a 2.6.16 stable tree, for as long as needed (years!).



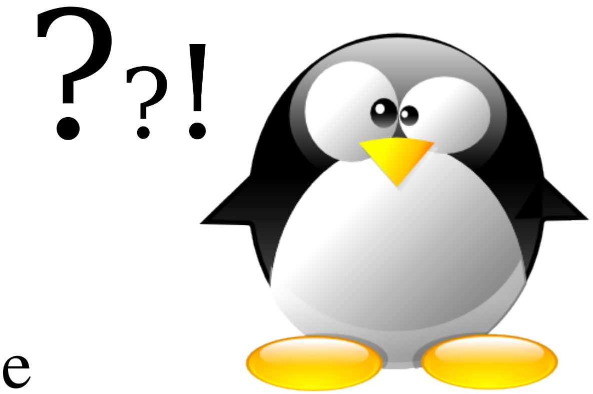
What's new in each Linux release? (1)

```
commit 3c92c2ba33cd7d666c5f83cc32aa590e794e91b0
Author: Andi Kleen <ak@suse.de>
Date:   Tue Oct 11 01:28:33 2005 +0200

    [PATCH] i386: Don't discard upper 32bits of HWCR on K8

    Need to use long long, not long when RMWing a MSR. I think
    it's harmless right now, but still should be better fixed
    if AMD adds any bits in the upper 32bit of HWCR.

    Bug was introduced with the TLB flush filter fix for i386

    Signed-off-by: Andi Kleen <ak@suse.de>
    Signed-off-by: Linus Torvalds <torvalds@osdl.org>
...
```

- The official list of changes for each Linux release is just a huge list of individual patches!
- Very difficult to find out the key changes and to get the global picture out of individual changes.

What's new in each Linux release? (2)

- Fortunately, a summary of key changes with enough details is available on http://wiki.kernelnewbies.org/LinuxChanges

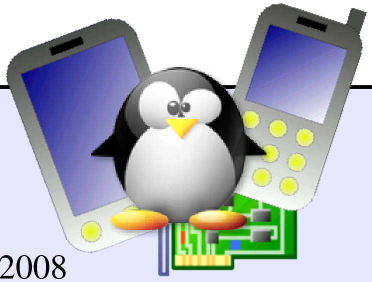


- For each new kernel release, you can also get the changes in the kernel internal API: http://lwn.net/Articles/2.6-kernel-api/

- What's next? Documentation/feature-removal-schedule.txt lists the features, subsystems and APIs that are planned for removal (announced 1 year in advance).

Kernel overview

Legal issues



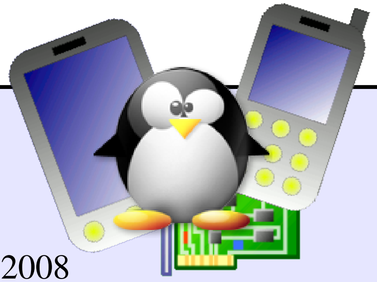
Linux license

- The whole Linux sources are Free Software released under the GNU General Public License version 2 (GPL v2).
- See our http://free-electrons.com/articles/freesw/ training for details about Free Software and its licenses.

Linux kernel licensing constraints

Constraints at release time (no constraint before!)

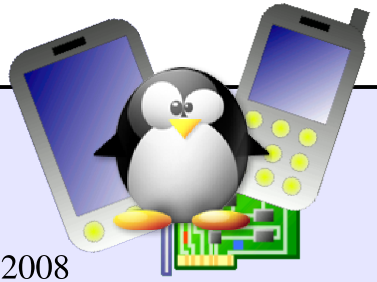
- For any device embedding Linux and Free Software, you have to release sources to the end user. You have no obligation to release them to anybody else!
- According to the GPL, only Linux drivers with a GPL compatible license are allowed.
- Proprietary drivers are less and less tolerated. Lawyers say that they are illegal.
- Proprietary drivers must not be statically compiled into the kernel.
- You are not allowed to reuse code from other kernel drivers (GPL) in a proprietary driver.



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.



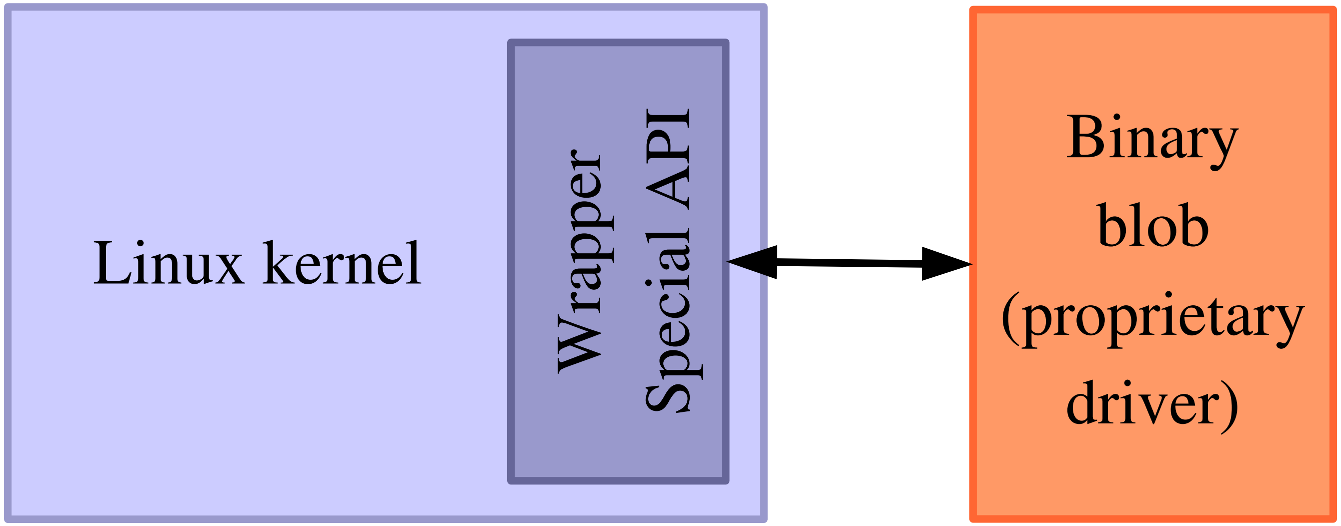
Advantages of in-tree kernel drivers

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.

Legal proprietary Linux drivers (1)

Working around the GPL by creating a GPL wrapper:



The proprietary blob is **not broken** when you recompile or update the kernel and/or driver. Hence, the proprietary driver may not be considered as a derivative work. However, the kernel is monolithic and the blob still belongs to a single executable. This is still controversial!

Legal proprietary Linux drivers (2)

2 example cases

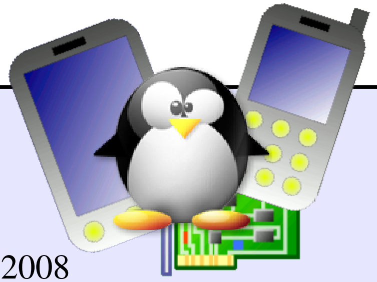
- Nvidia graphic card drivers
- Supporting wireless network cards using Windows drivers.

  The NdisWrapper project (http://ndiswrapper.sourceforge.net/) implements the Windows kernel API and NDIS (Network Driver Interface Spec.) API within the Linux kernel.

  Useful for using cards for which no specifications are released.

Drawbacks

- Still some maintenance issues.
  Example: Nvidia proprietary driver incompatible with X.org 7.1.
- Performance issues.
  Wrapper overhead and optimizations not available.
- Security issues. The drivers are executed with full kernel privileges.
- ... and all other issues with proprietary drivers. Users lose most benefits of Free Software.

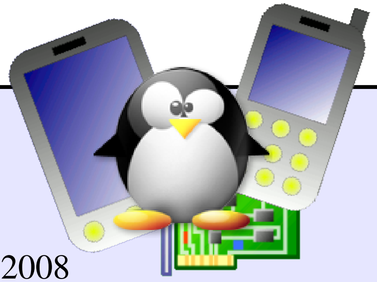


Software patent issues in the kernel

Linux Kernel driver issues because of patented algorithms

Check for software patent warnings when you configure your kernel!

- Patent warnings issued in the documentation of drivers, shown in the kernel configuration interface.
- Flash Translation Layer
  `drivers/mtd/ftl.c`
  In the USA, this driver can only be used on PCMCIA hardware (MSystems patent).
- Nand Flash Translation Layer
  In the USA, can only be used on DiskOnChip hardware.
- Networking compression
  `drivers/net/bsd_comp.c`
  Can't send a CCP reset-request as a result of an error detected after decompression (Motorola patent).
- Other drivers not accepted in Linux releases or algorithms not implemented because of such patents!
  Otherwise, more examples would be available in the source code.



Kernel overview

Kernel user interface

Mounting virtual filesystems

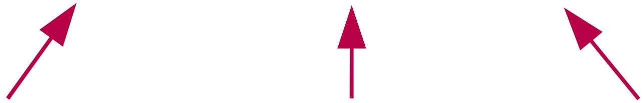
- Linux makes system and kernel information available in user-space through virtual filesystems (virtual files not existing on any real storage). No need to know kernel programming to access this!

- Mounting `/proc`:

```
sudo mount -t proc none /proc
```

- Mounting `/sys`:

```
sudo mount -t sysfs none /sys
```



Filesystem type

Raw device
or filesystem image
In the case of virtual
filesystems, any string is fine

Mount point

Kernel userspace interface

A few examples:

- `/proc/cpuinfo`: processor information
- `/proc/meminfo`: memory status
- `/proc/version`: kernel version and build information
- `/proc/cmdline`: kernel command line
- `/proc/<pid>/environ`: calling environment
- `/proc/<pid>/cmdline`: process command line

... and many more! See by yourself!

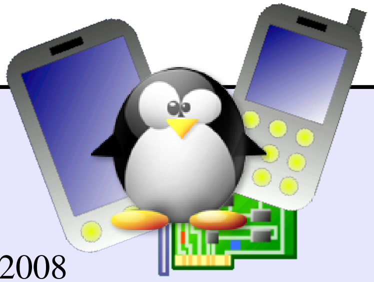
Userspace interface documentation

- Lots of details about the `/proc` interface are available in `Documentation/filesystems/proc.txt` (almost 2000 lines) in the kernel sources.
- You can also find other details in the `proc` manual page: `man proc`
- See the New Device Model section for details about `/sys`

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 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).

  Less in-kernel complexity.

- Drawbacks

  Less straightforward to handle interrupts.

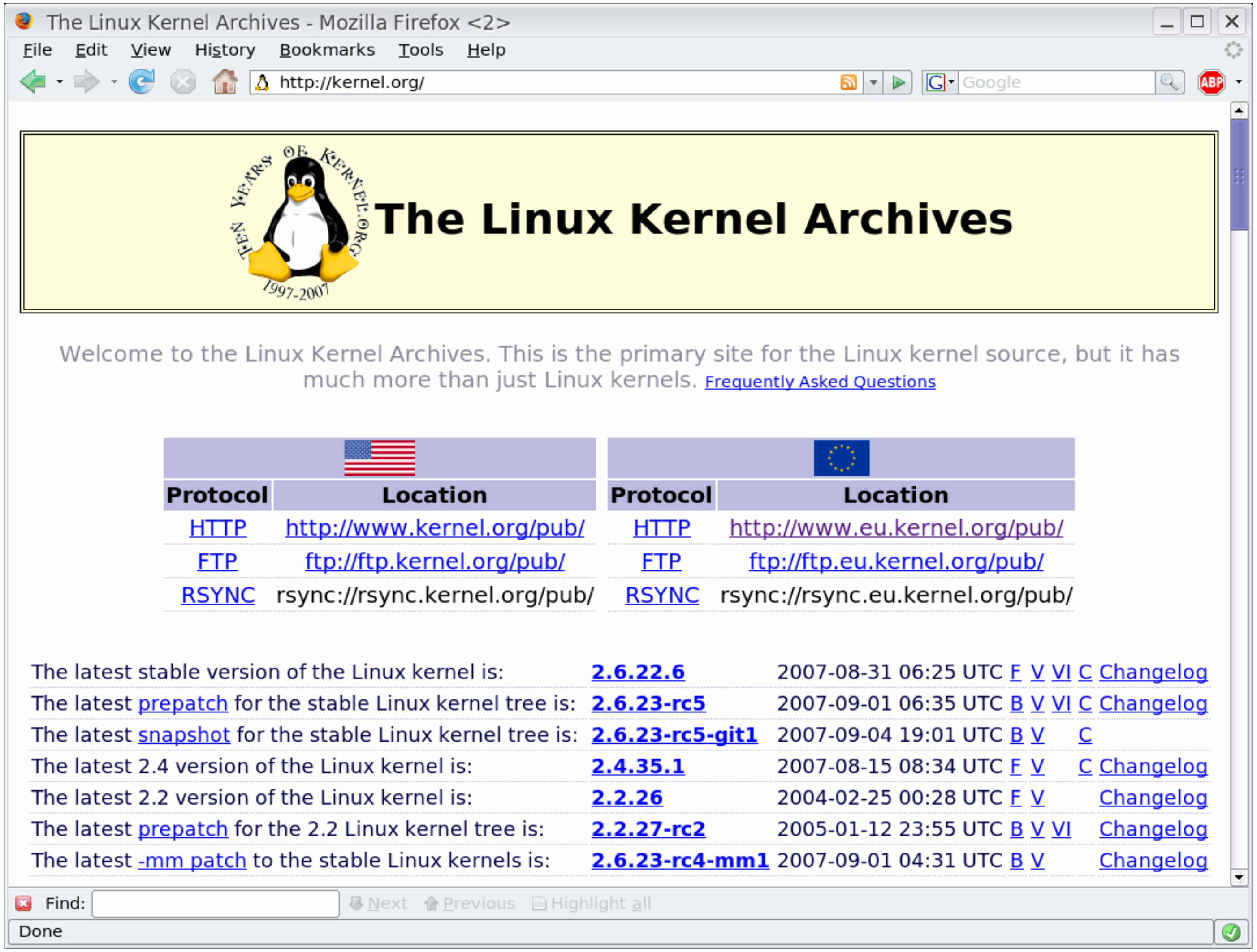
  Increased latency vs. kernel code.

See http://free-electrons.com/redirect/elc2006-uld.html
for practical details and techniques for overcoming the drawbacks.

Compiling and booting Linux

Linux kernel sources

kernel.org


The Linux Kernel Archives - Mozilla Firefox <2>
File Edit View History Bookmarks Tools Help
http://kernel.org/
Google
TEN YEARS OF KERNEL.ORG
1997-2007
Find:
Next
Previous
Highlight all
Done

The Linux Kernel Archives

Welcome to the Linux Kernel Archives. This is the primary site for the Linux kernel source, but it has much more than just Linux kernels. Frequently Asked Questions

Protocol	Location
HTTP	http://www.kernel.org/pub/
FTP	ftp://ftp.kernel.org/pub/
RSYNC	rsync://rsync.kernel.org/pub/

Protocol	Location
HTTP	http://www.eu.kernel.org/pub/
FTP	ftp://ftp.eu.kernel.org/pub/
RSYNC	rsync://rsync.eu.kernel.org/pub/

The latest stable version of the Linux kernel is:	2.6.22.6	2007-08-31 06:25 UTC	F	V	VI	C	Changelog
The latest prepatch for the stable Linux kernel tree is:	2.6.23-rc5	2007-09-01 06:35 UTC	B	V	VI	C	Changelog
The latest snapshot for the stable Linux kernel tree is:	2.6.23-rc5-git1	2007-09-04 19:01 UTC	B	V		C	
The latest 2.4 version of the Linux kernel is:	2.4.35.1	2007-08-15 08:34 UTC	F	V		C	Changelog
The latest 2.2 version of the Linux kernel is:	2.2.26	2004-02-25 00:28 UTC	F	V			Changelog
The latest prepatch for the 2.2 Linux kernel tree is:	2.2.27-rc2	2005-01-12 23:55 UTC	B	V	VI		Changelog
The latest -mm patch to the stable Linux kernels is:	2.6.23-rc4-mm1	2007-09-01 04:31 UTC	B	V			Changelog

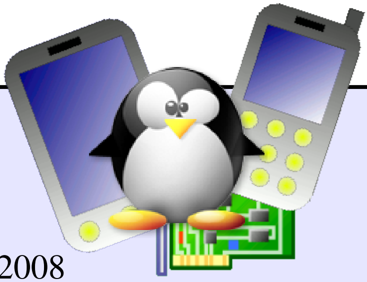
Download kernel sources from http://kernel.org

Linux sources structure (1)

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

Linux sources structure (2)

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



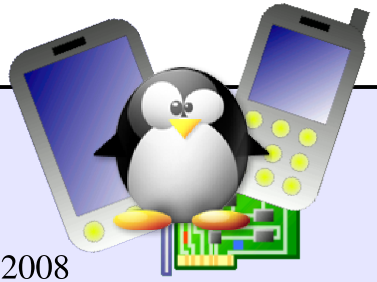
Linux kernel size (1)

- Linux 2.6.17 sources:
  Raw size: 224 MB (20400 files, approx 7 million lines of code)
  `gzip` compressed tar archive: 50 MB
  `bzip2` compressed tar archive: 40 MB (better)
  `7zip` compressed tar archive: 33 MB (best)

- Minimum compiled Linux kernel size (with Linux-Tiny patches)
  approx 300 KB (compressed), 800 KB (raw) (Linux 2.6.14)

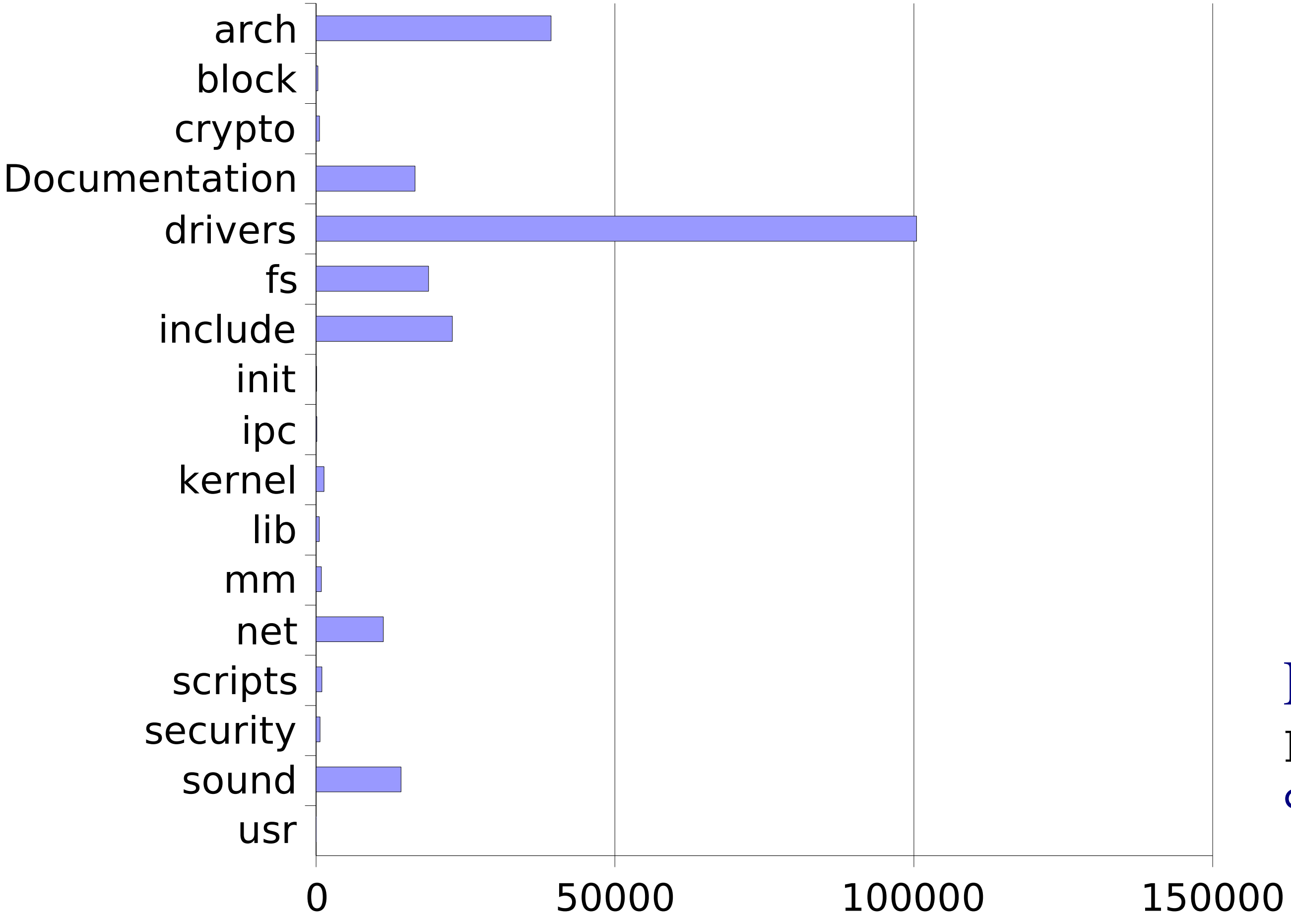
- Why are these sources so big?
  Because they include thousands of device drivers, many network protocols, support many architectures and filesystems...

- The Linux core (scheduler, memory management...) is pretty small!



Linux kernel size (2)

Size of Linux source directories (KB)



Linux 2.6.17

Measured with:

`du -s --apparent-size`

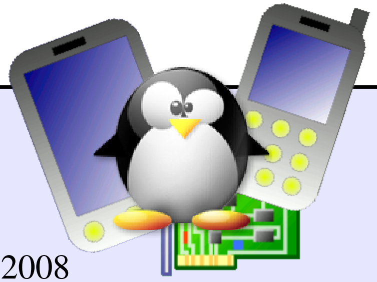
Getting Linux sources: 2 possibilities

Full sources

- The easiest way, but longer to download.
- Example:
  http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.14.7.tar.bz2

Or patch against the previous version

- Assuming you already have the full sources of the previous version
- Example:
  http://kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.bz2 (2.6.13 to 2.6.14)
  http://kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.7.bz2 (2.6.14 to 2.6.14.7)



Downloading full kernel sources

Downloading from the command line

- With a web browser, identify the version you need on http://kernel.org
- In the right directory, download the source archive and its signature (copying the download address from the browser):

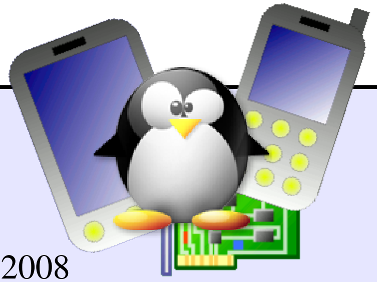
```
wget http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.11.12.tar.bz2
```

- Extract the contents of the source archive:

```
tar jxf linux-2.6.11.12.tar.bz2
```

`~/.wgetrc` config file for proxies:

```
http_proxy = <proxy>:<port>
ftp_proxy = <proxy>:<port>
proxy_user = <user> (if any)
proxy_password = <passwd> (if any)
```



Downloading kernel source patches (1)

Assuming you already have the `linux-x.y.<n-1>` version

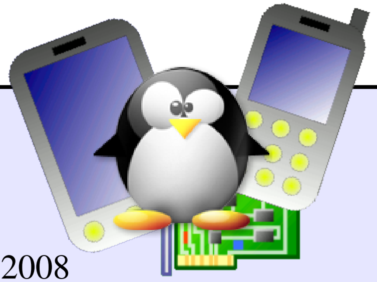
- Identify the patches you need on http://kernel.org with a web browser
- Download the patch files and their signature:

Patch from `2.6.10` to `2.6.11`

```
wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/patch-2.6.11.bz2
```

Patch from `2.6.11` to `2.6.11.12` (latest stable fixes)

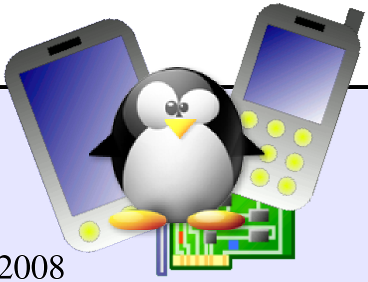
```
wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.11.12.bz2
```



Downloading kernel source patches (2)

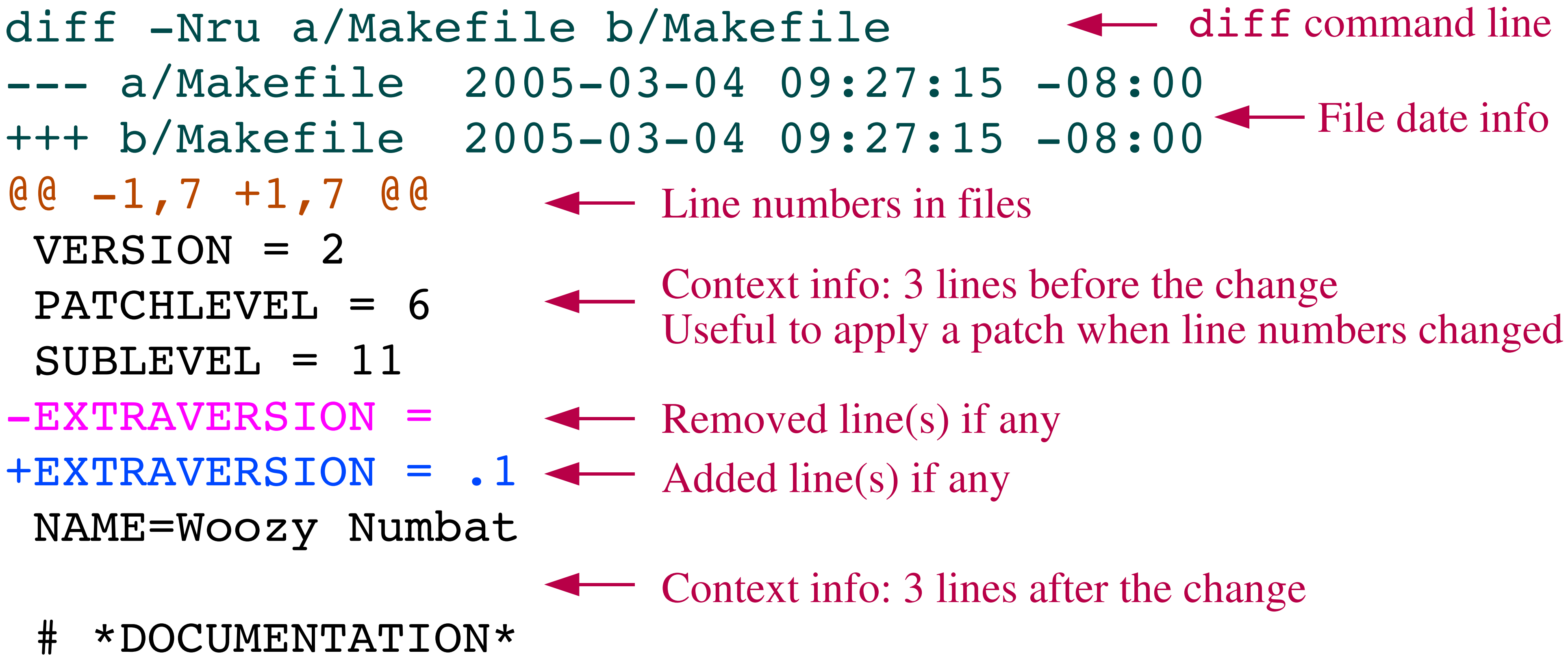
Apply the patches in the right order:

```
cd linux-2.6.10/
bzcat ../patch-2.6.11.bz2 | patch -p1
bzcat ../patch-2.6.11.12.bz2 | patch -p1
cd ..
mv linux-2.6.10 linux-2.6.11.12
```



Anatomy of a patch file

A patch file is the output of the `diff` command



```
diff -Nru a/Makefile b/Makefile
--- a/Makefile  2005-03-04 09:27:15 -08:00
+++ b/Makefile  2005-03-04 09:27:15 -08:00
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
 SUBLEVEL = 11
-EXTRAVERSION =
+EXTRAVERSION = .1
 NAME=Woozy Numbat

 # *DOCUMENTATION*
```

`diff` command line

File date info

Line numbers in files

Context info: 3 lines before the change
Useful to apply a patch when line numbers changed

Removed line(s) if any

Added line(s) if any

Context info: 3 lines after the change

Using the patch command

The `patch` command applies changes to files in the current directory:

- Making changes to existing files
- Creating or deleting files and directories

`patch` usage examples:

```
patch -p<n> < diff_file
cat diff_file | patch -p<n>
bzcat diff_file.bz2 | patch -p<n>
zcat diff_file.gz | patch -p<n>
```

`n`: number of directory levels to skip in the file paths

You can reverse a patch with the `-R` option

You can test a patch with the `--dry-run` option

Applying a Linux patch

Linux patches...

- Always to apply to the `x.y.<z-1>` version
  Downloadable in `gzip`
  and `bzip2` (much smaller) compressed files.
- Always produced for `n=1`
  (that's what everybody does... do it too!)
- Linux patch command line example:

```
cd linux-2.6.10
bzcat ../patch-2.6.11.bz2 | patch -p1
cd ..; mv linux-2.6.10 linux-2.6.11
```

- Keep patch files compressed: useful to check their signature later.
  You can still view (or even edit) the uncompressed data with `vi`:
  `vi patch-2.6.11.bz2` (on the fly (un)compression)

You can make `patch` 30% faster by using `-sp1` instead of `-p1` (**s**ilent)

Tested on `patch-2.6.23.bz2`

Accessing development sources (1)

- 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
- 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)

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

More details available

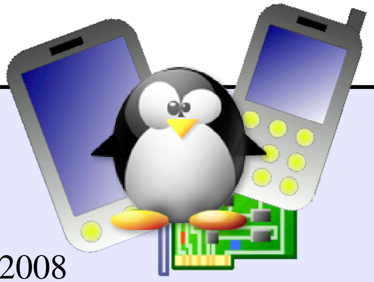
on http://git.or.cz/ or http://linux.yyz.us/git-howto.html

On-line kernel documentation

http://free-electrons.com/kerneldoc/

- Provided for all recent kernel releases
- Easier than downloading kernel sources to access documentation
- Indexed by Internet search engines
  Makes kernel pieces of documentation easier to find!
- Unlike most other sites offering this service too, also includes an HTML translation of kernel documents in the DocBook format.

Never forget documentation in the kernel sources! It's a very valuable way of getting information about the kernel.



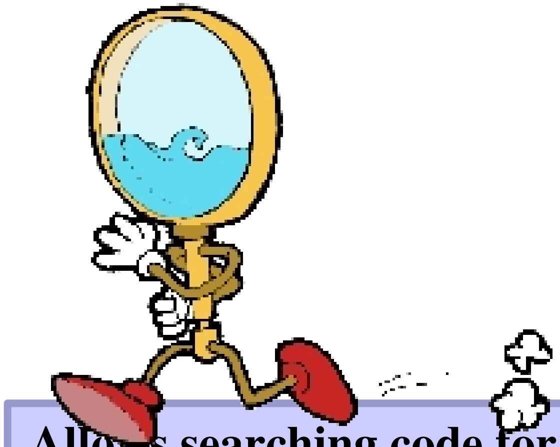
Compiling and booting Linux

Kernel source management tools

Cscope

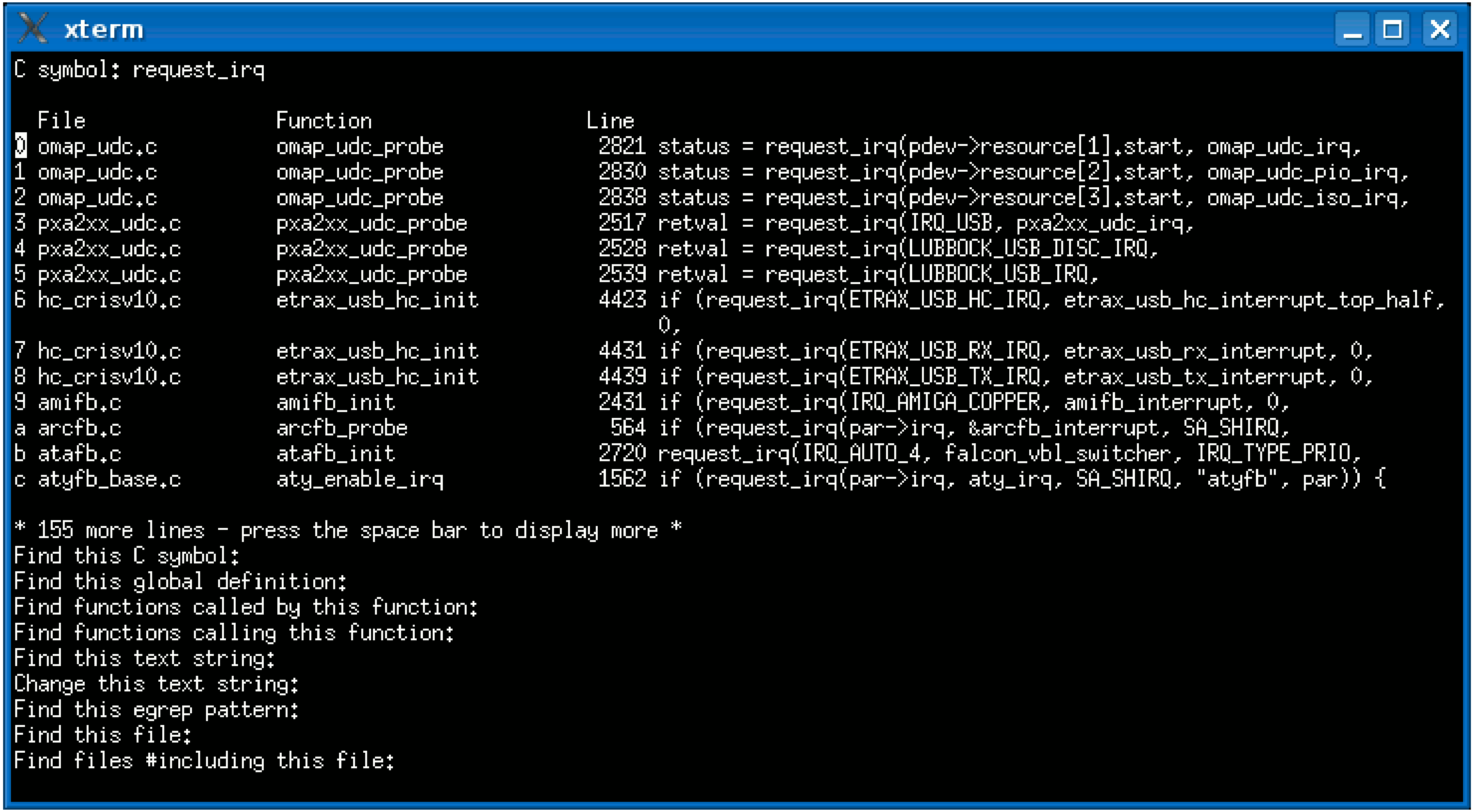
http://cscope.sourceforge.net/

- 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



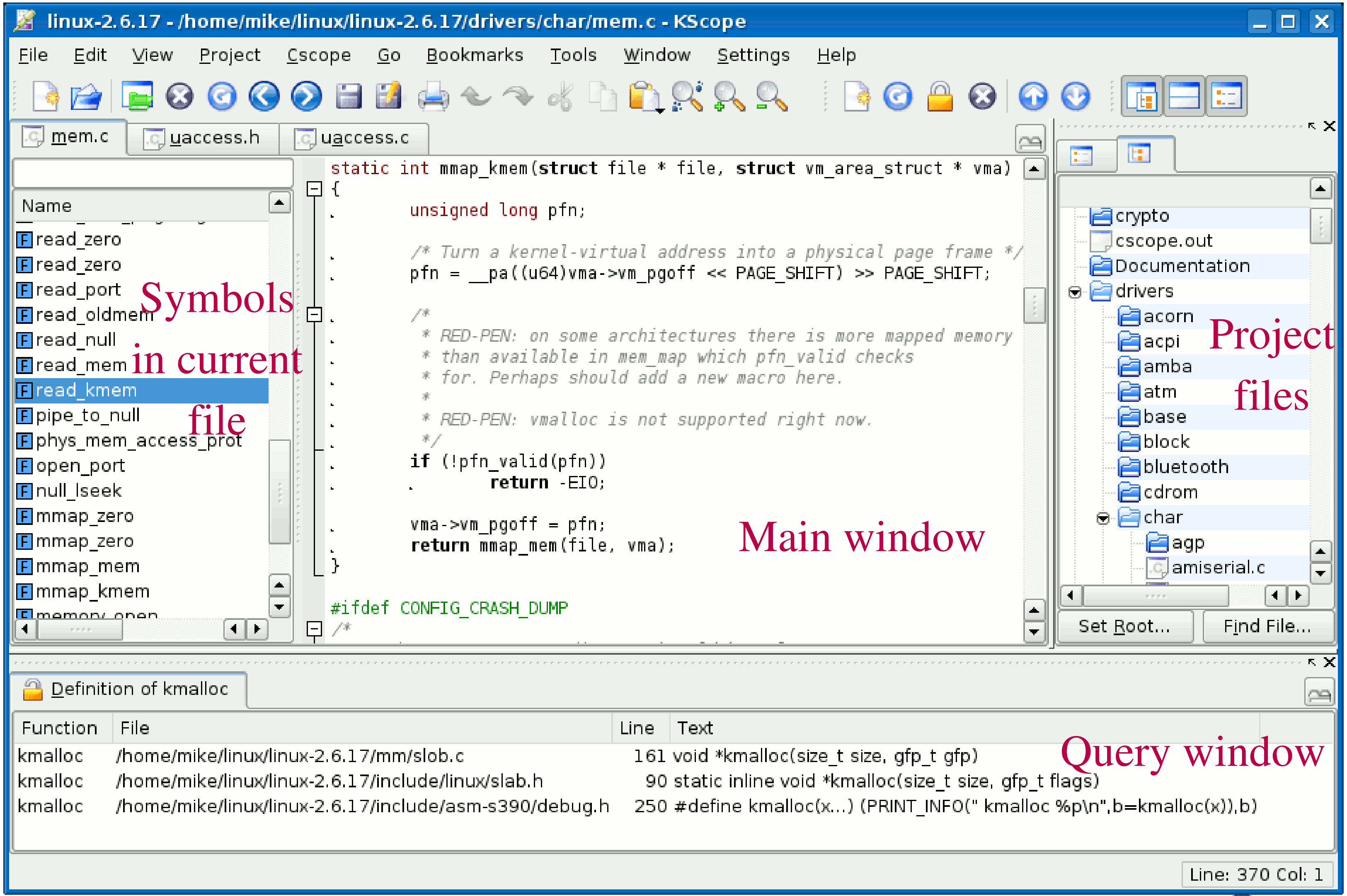
```
C symbol: request_irq

  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)



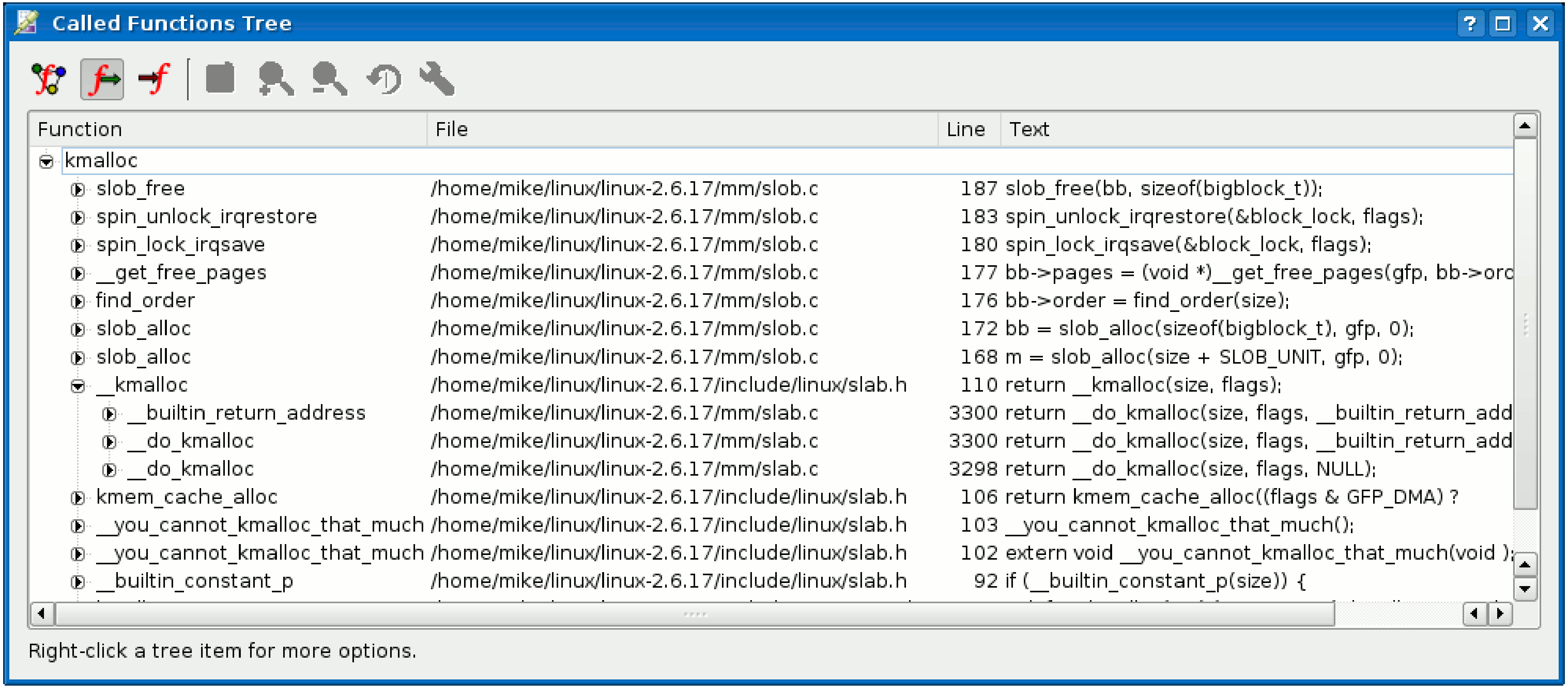
- F read_zero
- F read_zero
- F read_port
- F read_oldmem
- F read_null
- F read_mem
- F read_kmem
- F pipe_to_null
- F phys_mem_access_prot
- F open_port
- F null_lseek
- F mmap_zero
- F mmap_zero
- F mmap_mem
- F mmap_kmem
- F memory_open

```
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
/*
```

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

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)

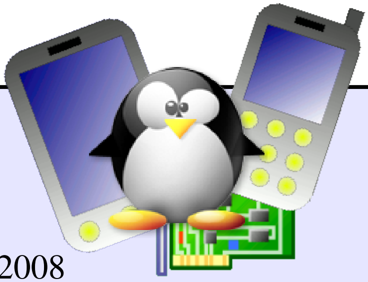
KScope screenshots (2)



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->orc
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_add
__do_kmalloc	/home/mike/linux/linux-2.6.17/mm/slab.c	3300	return __do_kmalloc(size, flags, __builtin_return_add
__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) ?
__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



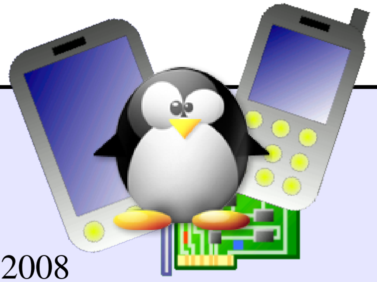
LXR: Linux Cross Reference

http://sourceforge.net/projects/lxr

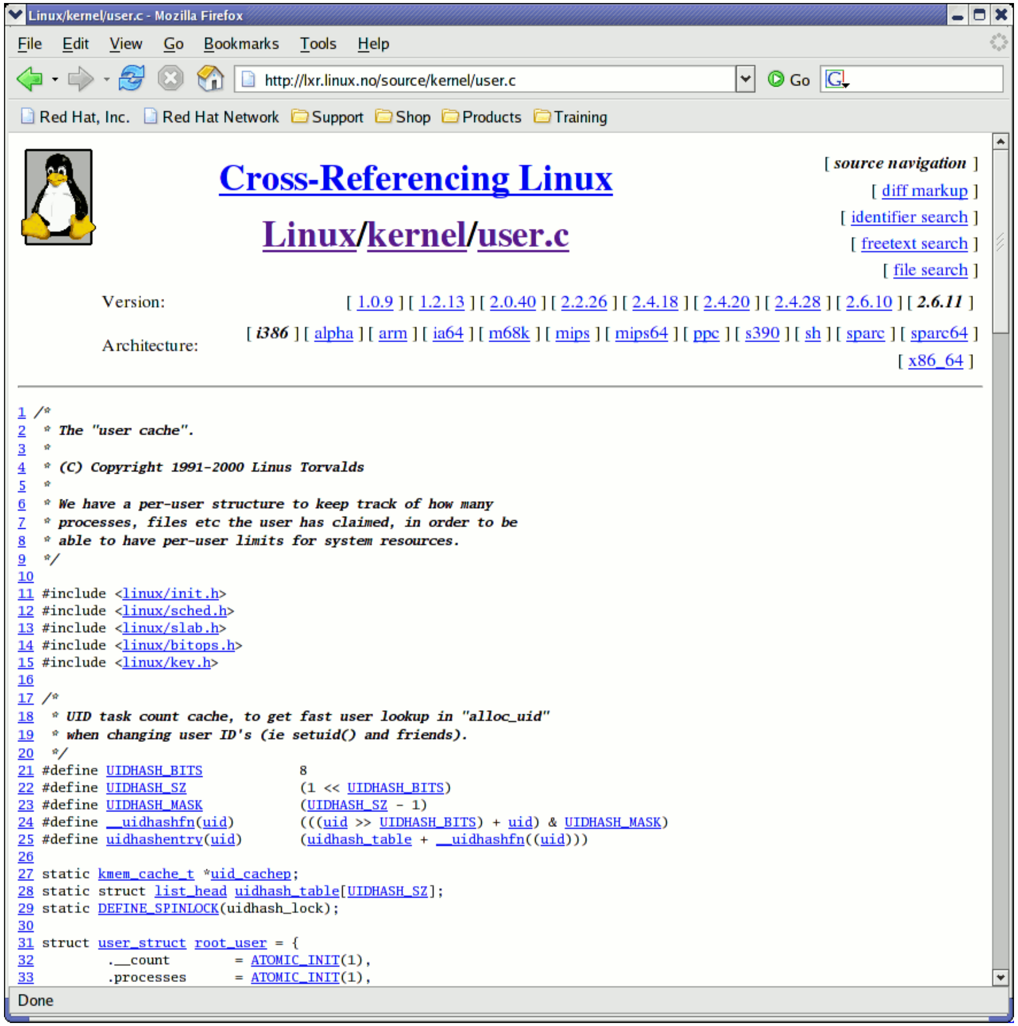
Generic source indexing tool and code browser

- 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 (274 M in version 2.6.17).

- Takes some time and patience to setup (configuration, indexing, server configuration).
- Initial indexing very slow:
  Linux 2.6.17: several hours on a server with an AMD Sempron 2200+ CPU.
  Using Kscope is the easiest and fastest solution for modified kernel sources.
- 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



Cross-Referencing Linux

Linux/kernel/user.c

[*source navigation*]
[diff markup]
[identifier search]
[freetext search]
[file search]

Version: [1.0.9] [1.2.13] [2.0.40] [2.2.26] [2.4.18] [2.4.20] [2.4.28] [2.6.10] [***2.6.11***]

Architecture: [*i386*] [alpha] [arm] [ia64] [m68k] [mips] [mips64] [ppc] [s390] [sh] [sparc] [sparc64] [x86_64]

```
/*
 * The "user cache".
 *
 * (C) Copyright 1991-2000 Linus Torvalds
 *
 * We have a per-user structure to keep track of how many
 * processes, files etc the user has claimed, in order to be
 * able to have per-user limits for system resources.
 */

#include <linux/init.h>
#include <linux/sched.h>
#include <linux/slab.h>
#include <linux/bitops.h>
#include <linux/key.h>

/*
 * UID task count cache, to get fast user lookup in "alloc_uid"
 * when changing user ID's (ie setuid() and friends).
 */
#define UIDHASH_BITS            8
#define UIDHASH_SZ              (1 << UIDHASH_BITS)
#define UIDHASH_MASK            (UIDHASH_SZ - 1)
#define __uidhashfn(uid)        (((uid >> UIDHASH_BITS) + uid) & UIDHASH_MASK)
#define uidhashentry(uid)       (uidhash_table + __uidhashfn((uid)))

static kmem_cache_t *uid_cachep;
static struct list_head uidhash_table[UIDHASH_SZ];
static DEFINE_SPINLOCK(uidhash_lock);

struct user_struct root_user = {
        .__count        = ATOMIC_INIT(1),
        .processes      = ATOMIC_INIT(1),
```

Ketchup - Easy access to kernel source trees

http://www.selenic.com/ketchup/wiki/

- Makes it easy to get the latest version of a given kernel source tree (`2.4`, `2.6`, `2.6-rc`, `2.6-git`, `2.6-mm`, `2.6-rt`...)
- Only downloads the needed patches.
  Reverts patches when needed to apply a more recent patch.
- Also checks the signature of sources and patches.

Ketchup examples

- Get the version in the current directory:

```
> ketchup -m
2.6.10
```

- Upgrade to the latest stable version:

```
> ketchup -G 2.6-tip
2.6.10 -> 2.6.12.5
Applying patch-2.6.11.bz2
Applying patch-2.6.12.bz2
Applying patch-2.6.12.5.bz2
```

- You can get back to `2.6.8`:

```
> ketchup -G 2.6.8
```

The `-G` option of ketchup disables source signature checking.

See our Kernel sources annex for details about enabling kernel source integrity checking.

More on http://selenic.com/ketchup/wiki/index.cgi/ExampleUsage

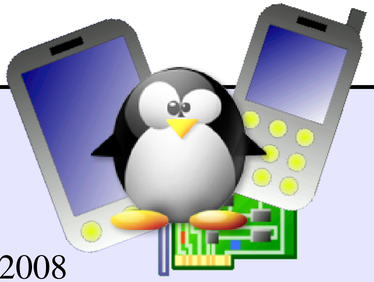
Practical lab – Kernel sources

Time to start `Lab 1`!

- Get the sources
- Check the authenticity of sources
- Apply patches
- Get familiar with the sources
- Use a kernel source indexing tool

Compiling and booting Linux

Kernel configuration



Kernel configuration

Defines what features to include in the kernel:

- Stored in the `.config` file at the root of kernel sources.
- Most useful commands to create this config file:
  ```
  make [xconfig|gconfig|menuconfig|oldconfig]
  ```
- To modify a kernel in a GNU/Linux distribution:

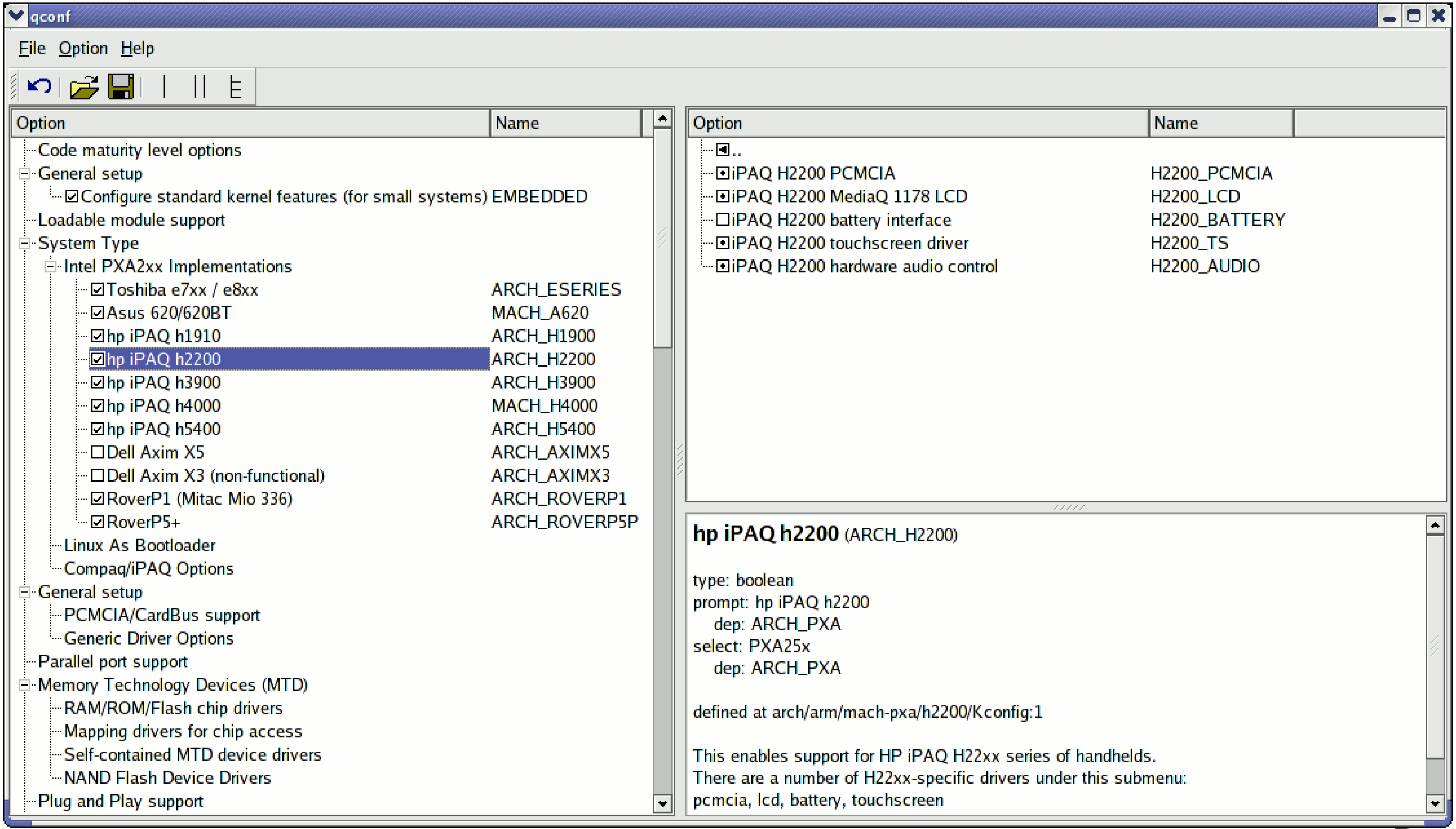
  config files usually released in `/boot/`, together with kernel images:
  ```
  /boot/config-2.6.17-11-generic
  ```
- The configuration file can also be found in the kernel itself:
  ```
  > zcat /proc/config.gz
  ```
  (if enabled in `General Setup -> Kernel .config support`)

make xconfig

```
make xconfig
```

- New Qt configuration interface for Linux 2.6: qconf. Much easier to use than in Linux 2.4!
- Make sure you read `help -> introduction: useful options!`
- File browser: easier to load configuration files
- New search interface to look for parameters

make xconfig screenshot



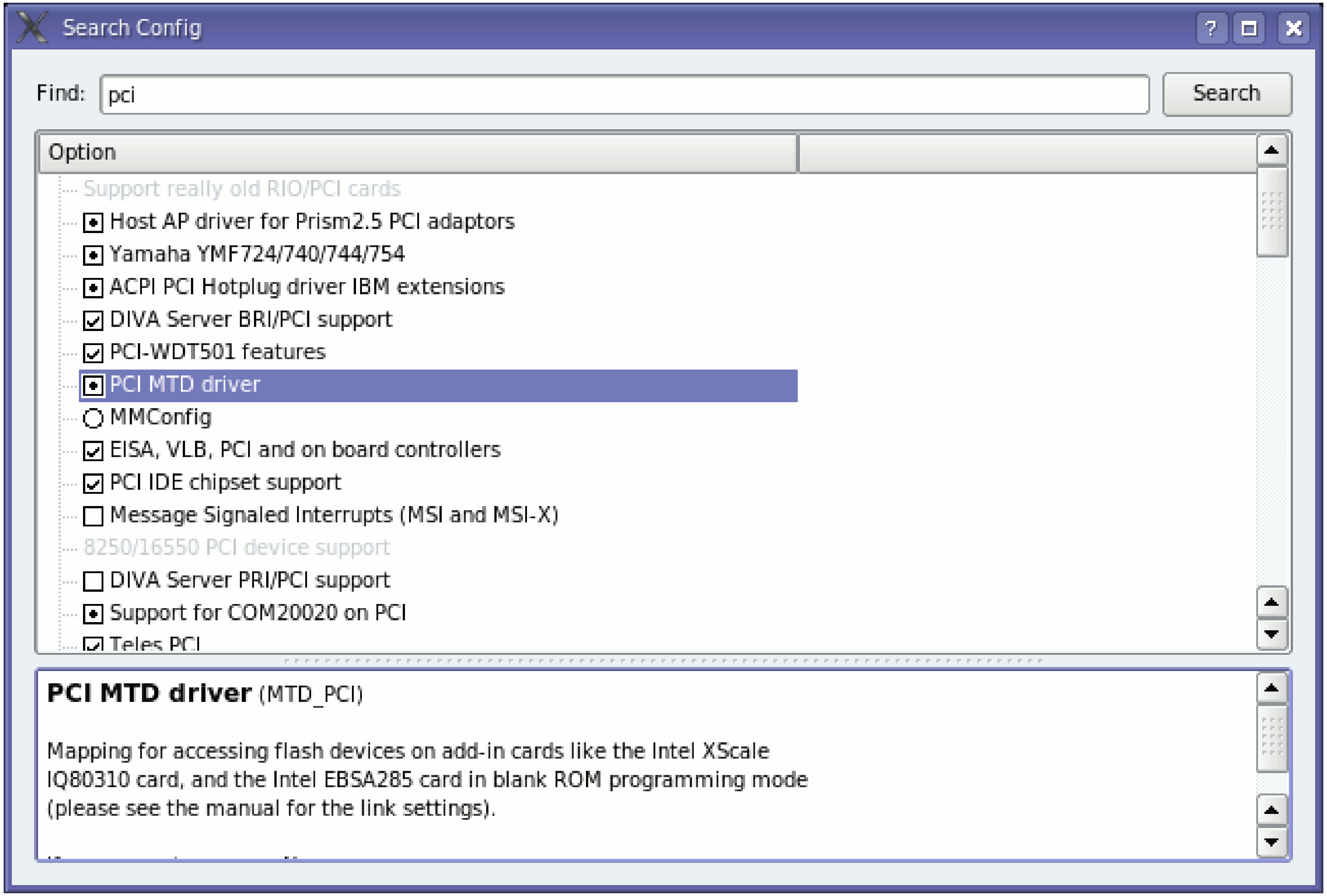
```
Code maturity level options
General setup
    ☑Configure standard kernel features (for small systems) EMBEDDED
Loadable module support
System Type
    Intel PXA2xx Implementations
        ☑Toshiba e7xx / e8xx                      ARCH_ESERIES
        ☑Asus 620/620BT                           MACH_A620
        ☑hp iPAQ h1910                            ARCH_H1900
        ☑hp iPAQ h2200                            ARCH_H2200
        ☑hp iPAQ h3900                            ARCH_H3900
        ☑hp iPAQ h4000                            MACH_H4000
        ☑hp iPAQ h5400                            ARCH_H5400
        ☐Dell Axim X5                             ARCH_AXIMX5
        ☐Dell Axim X3 (non-functional)            ARCH_AXIMX3
        ☑RoverP1 (Mitac Mio 336)                  ARCH_ROVERP1
        ☑RoverP5+                                 ARCH_ROVERP5P
    Linux As Bootloader
    Compaq/iPAQ Options
General setup
    PCMCIA/CardBus support
    Generic Driver Options
Parallel port support
Memory Technology Devices (MTD)
    RAM/ROM/Flash chip drivers
    Mapping drivers for chip access
    Self-contained MTD device drivers
    NAND Flash Device Drivers
Plug and Play support
```

```
..
☑iPAQ H2200 PCMCIA                    H2200_PCMCIA
☑iPAQ H2200 MediaQ 1178 LCD           H2200_LCD
☐iPAQ H2200 battery interface         H2200_BATTERY
☑iPAQ H2200 touchscreen driver        H2200_TS
☑iPAQ H2200 hardware audio control    H2200_AUDIO
```

hp iPAQ h2200 (ARCH_H2200)

```
type: boolean
prompt: hp iPAQ h2200
    dep: ARCH_PXA
select: PXA25x
    dep: ARCH_PXA

defined at arch/arm/mach-pxa/h2200/Kconfig:1

This enables support for HP iPAQ H22xx series of handhelds.
There are a number of H22xx-specific drivers under this submenu:
pcmcia, lcd, battery, touchscreen
```

make xconfig search interface



Find: pci

- ···· Support really old RIO/PCI cards
- ⊡ Host AP driver for Prism2.5 PCI adaptors
- ⊡ Yamaha YMF724/740/744/754
- ⊡ ACPI PCI Hotplug driver IBM extensions
- ☑ DIVA Server BRI/PCI support
- ☑ PCI-WDT501 features
- ⊡ PCI MTD driver
- ○ MMConfig
- ☑ EISA, VLB, PCI and on board controllers
- ☑ PCI IDE chipset support
- ☐ Message Signaled Interrupts (MSI and MSI-X)
- ···· 8250/16550 PCI device support
- ☐ DIVA Server PRI/PCI support
- ⊡ Support for COM20020 on PCI
- ☑ Teles PCI

PCI MTD driver (MTD_PCI)

Mapping for accessing flash devices on add-in cards like the Intel XScale IQ80310 card, and the Intel EBSA285 card in blank ROM programming mode (please see the manual for the link settings).

Looks for a keyword in the description string

Allows to select or unselect found parameters.

Kernel configuration options

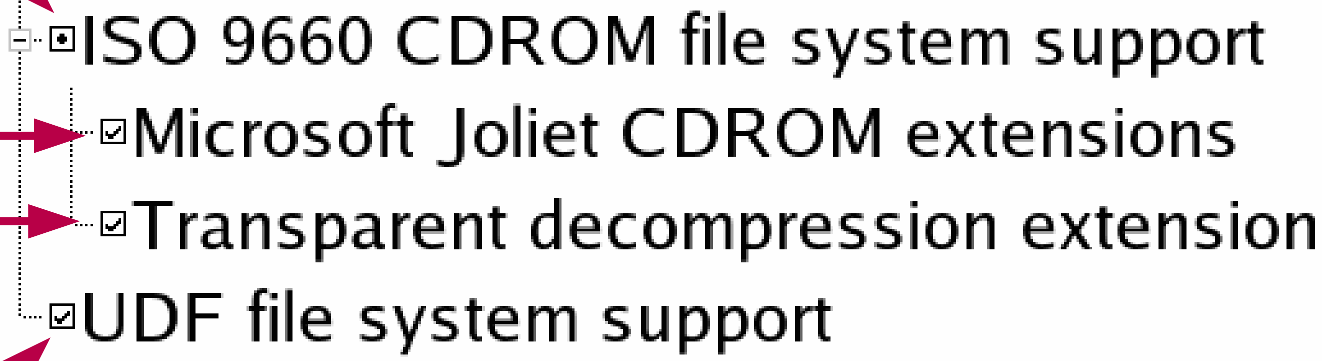
Compiled as a module (separate file)

CONFIG_ISO9660_FS=m

Driver options

CONFIG_JOLIET=y

CONFIG_ZISOFS=y



⊡ ISO 9660 CDROM file system support

- ☑ Microsoft Joliet CDROM extensions
- ☑ Transparent decompression extension

☑ UDF file system support

Compiled statically into the kernel

CONFIG_UDF_FS=y

Corresponding .config file excerpt

```
#
# CD-ROM/DVD Filesystems
#
CONFIG_ISO9660_FS=m
CONFIG_JOLIET=y
CONFIG_ZISOFS=y
CONFIG_UDF_FS=y
CONFIG_UDF_NLS=y

#
# DOS/FAT/NT Filesystems
#
# CONFIG_MSDOS_FS is not set
# CONFIG_VFAT_FS is not set
CONFIG_NTFS_FS=m
# CONFIG_NTFS_DEBUG is not set
CONFIG_NTFS_RW=y
```

Section name

(helps to locate settings in the interface)

All parameters are prefixed

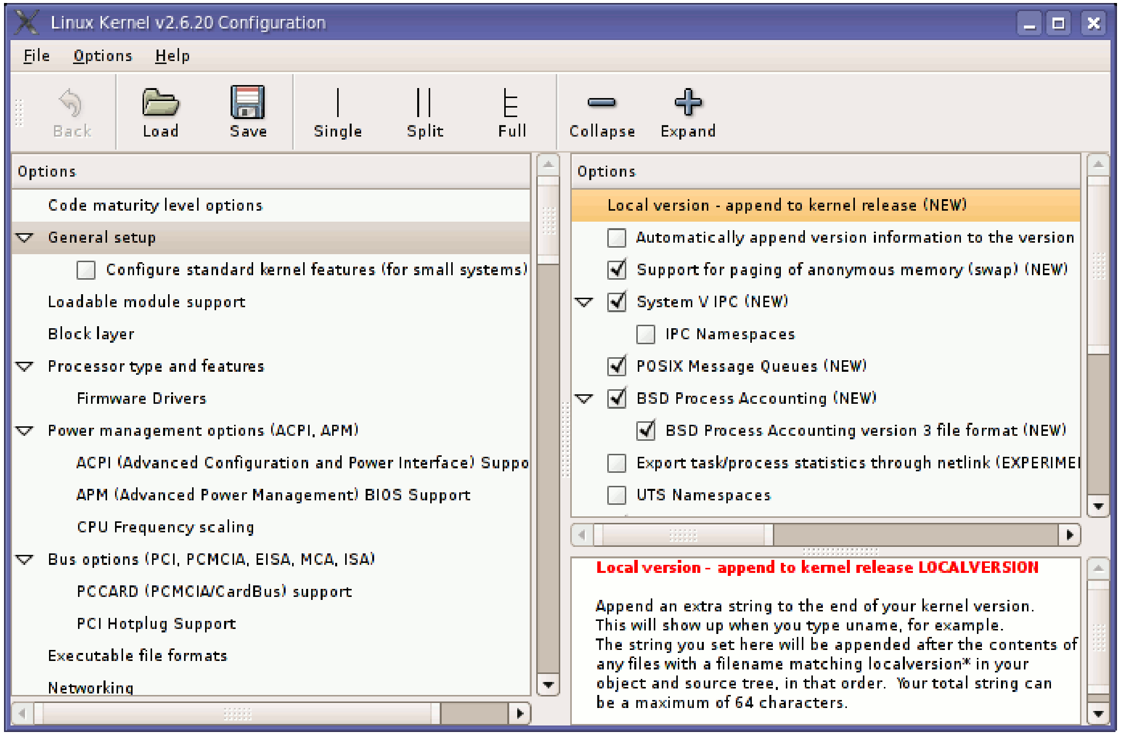
with `CONFIG_`

make gconfig

`make gconfig`

New GTK based graphical configuration interface. Functionality similar to that of `make xconfig`.

Just lacking a search functionality.



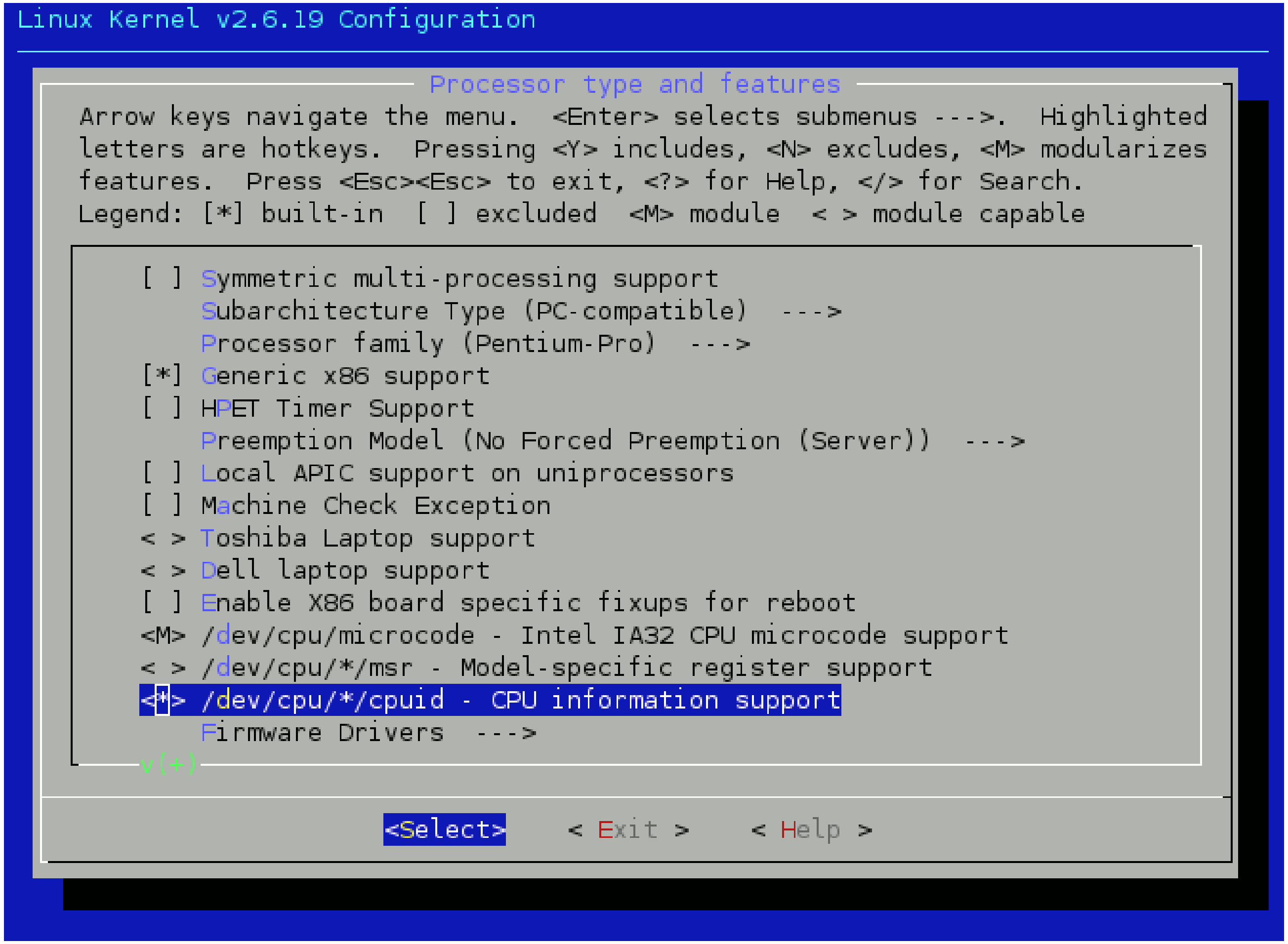
```
Code maturity level options
▽ General setup
    □ Configure standard kernel features (for small systems)
Loadable module support
Block layer
▽ Processor type and features
    Firmware Drivers
▽ Power management options (ACPI, APM)
    ACPI (Advanced Configuration and Power Interface) Suppo
    APM (Advanced Power Management) BIOS Support
    CPU Frequency scaling
▽ Bus options (PCI, PCMCIA, EISA, MCA, ISA)
    PCCARD (PCMCIA/CardBus) support
    PCI Hotplug Support
Executable file formats
Networking
```

```
   Local version - append to kernel release (NEW)
   □ Automatically append version information to the version
   ☑ Support for paging of anonymous memory (swap) (NEW)
▽  ☑ System V IPC (NEW)
       □ IPC Namespaces
   ☑ POSIX Message Queues (NEW)
▽  ☑ BSD Process Accounting (NEW)
       ☑ BSD Process Accounting version 3 file format (NEW)
   □ Export task/process statistics through netlink (EXPERIMEI
   □ UTS Namespaces
```

Local version - append to kernel release LOCALVERSION

Append an extra string to the end of your kernel version.
This will show up when you type uname, for example.
The string you set here will be appended after the contents of
any files with a filename matching localversion* in your
object and source tree, in that order. Your total string can
be a maximum of 64 characters.

make menuconfig



```
Linux Kernel v2.6.19 Configuration

 ┌──────────────────── Processor type and features ─────────────────────┐
 │  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted
 │  letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
 │  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
 │  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable
 │ ┌───────────────────────────────────────────────────────────────────┐
 │ │    [ ] Symmetric multi-processing support
 │ │        Subarchitecture Type (PC-compatible)  --->
 │ │        Processor family (Pentium-Pro)  --->
 │ │    [*] Generic x86 support
 │ │    [ ] HPET Timer Support
 │ │        Preemption Model (No Forced Preemption (Server))  --->
 │ │    [ ] Local APIC support on uniprocessors
 │ │    [ ] Machine Check Exception
 │ │    < > Toshiba Laptop support
 │ │    < > Dell laptop support
 │ │    [ ] Enable X86 board specific fixups for reboot
 │ │    <M> /dev/cpu/microcode - Intel IA32 CPU microcode support
 │ │    < > /dev/cpu/*/msr - Model-specific register support
 │ │    <*> /dev/cpu/*/cpuid - CPU information support
 │ │        Firmware Drivers  --->
 │ └────v(+)───────────────────────────────────────────────────────────┘
 ├──────────────────────────────────────────────────────────────────────┤
 │                   <Select>    < Exit >    < Help >
 └──────────────────────────────────────────────────────────────────────┘
```

`make menuconfig`

Same old text interface as in Linux 2.4.

Useful when no graphics are available. Pretty convenient too!

Same interface found in other tools: BusyBox, buildroot...

make oldconfig

`make oldconfig`

- Needed very often!
- Useful to upgrade a `.config` file from an earlier kernel release
- Issues warnings for obsolete symbols
- Asks for values for new symbols

If you edit a `.config` file by hand, it's strongly recommended to run `make oldconfig` afterwards!

make allnoconfig

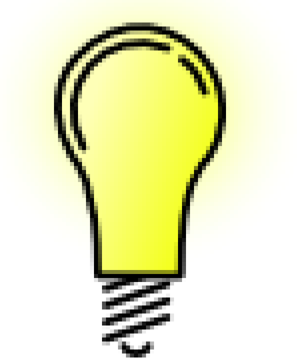
`make allnoconfig`

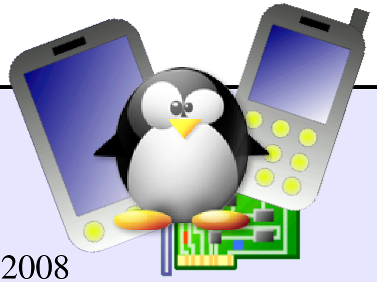
- Only sets strongly recommended settings to `y`.
- Sets all other settings to `n`.
- Very useful in embedded systems to select only the minimum required set of features and drivers.
- Much more convenient than unselecting hundreds of features one by one!

Undoing configuration changes

A frequent problem:

- After changing several kernel configuration settings, your kernel no longer works.
- If you don't remember all the changes you made, you can get back to your previous configuration:
  ```
  > cp .config.old .config
  ```
- All the configuration interfaces of the kernel (`xconfig`, `menuconfig`, `allnoconfig`...) keep this `.config.old` backup copy.





make help

`make help`

- Lists all available `make` targets
- Useful to get a reminder, or to look for new or advanced options!

Customizing the version string

- To identify your kernel image with others built from the same sources (but a different configuration), use the `LOCALVERSION` setting (in `General Setup`)
- Example:

```
#
# General setup
#
CONFIG_LOCALVERSION="-acme1"
```

- The `uname -r` command (in the running system) will return: `2.6.20-acme1`

Embedded Linux driver development

Compiling and booting Linux

Compiling the kernel

Compiling and installing the kernel

Compiling step

```
make
```

Install steps

```
sudo make install
```

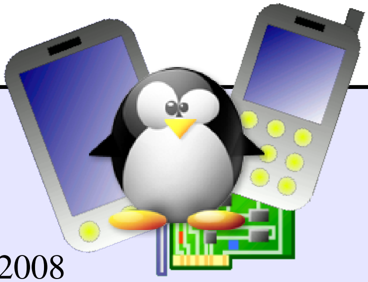
```
sudo make modules_install
```

Dependency management

- When you modify a regular kernel source file, `make` only rebuilds what needs recompiling. That's what it is used for.
- However, the `Makefile` is quite pessimistic about dependencies. When you make significant changes to the `.config` file, don't be surprised if `make` often recompiles most files, even when it doesn't seem necessary.

Compiling faster on multiprocessor hosts

- If you are using a workstation with `n` processors, you may roughly divide your compile time by `n` by compiling several files in parallel
- `make -j <n>`
  Runs several targets in parallel, whenever possible
- Using `make -j 2` or `make -j 3` on single processor workstations. This doesn't help much. In theory, several parallel compile jobs keep the processor busy while other processes are waiting for files to be read of written. In practice, you don't get any significant speedup (not more than 10%), unless your I/Os are very slow.



Compiling faster with ccache

http://ccache.samba.org/

Compiler cache for C and C++, already shipped by some distributions

Much faster when compiling the same file a second time!

- Very useful when `.config` file change are frequent.
- Use it by adding a `ccache` prefix

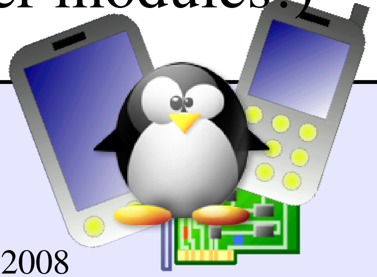
  to the `CC` and `HOSTCC` definitions in `Makefile`:

```
CC              = ccache $(CROSS_COMPILE)gcc
HOSTCC          = ccache gcc
```

- Performance benchmarks:

  -63%: with a Fedora Core 3 config file (many modules!)

  -82%: with an embedded Linux config file (much fewer modules!)



Kernel compiling tips

- View the full (gcc, ld...) command line:
  `make V=1`

- Clean-up generated files
  (to force re-compiling drivers):
  `make clean`

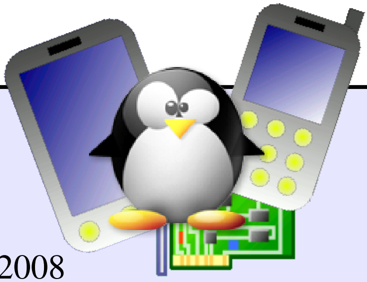
- Remove all generated files
  Caution: also removes your .config file!
  `make mrproper`

- Also remove editor backup and patch reject files:
  (mainly to generate patches):
  `make distclean`

Generated files

Created when you run the `make` command

- `vmlinux`
  Raw Linux kernel image, non compressed.
- `arch/<arch>/boot/zImage` (default image on `arm`)
  `zlib` compressed kernel image
- `arch/<arch>/boot/bzImage` (default image on `i386`)
  Also a `zlib` compressed kernel image.
  Caution: `bz` means "big zipped" but not "`bzip2` compressed"!
  (`bzip2` compression support only available on `i386` as a tactical patch. Not very attractive for small embedded systems though: consumes 1 MB of RAM for decompression).



Files created by make install

- `/boot/vmlinuz-<version>`
  Compressed kernel image. Same as the one in `arch/<arch>/boot`
- `/boot/System.map-<version>`
  Stores kernel symbol addresses
- `/boot/initrd-<version>.img` (when used by your distribution)
  Initial RAM disk, storing the modules you need to mount your root filesystem. `make install` runs `mkinitrd` for you!
- `/boot/grub/menu.lst` or `/etc/lilo.conf`
  `make install` updates your bootloader configuration files to support your new kernel! It reruns `/sbin/lilo` if LILO is your bootloader.

Files created by make modules_install (1)

`/lib/modules/<version>/`: Kernel modules + extras

- `build/`

  Everything needed to build more modules for this kernel: `Makefile`, `.config` file, module symbol information (`module.symVers`), kernel headers (`include/` and `include/asm/`)

- `kernel/`

  Module `.ko` (Kernel Object) files, in the same directory structure as in the sources.

Files created by make modules_install (2)

`/lib/modules/<version>/` (continued)

- `modules.alias`

  Module aliases for module loading utilities. Example line:
  `alias sound-service-?-0 snd_mixer_oss`

- `modules.dep`

  Module dependencies (see the Loadable kernel modules section)

- `modules.symbols`

  Tells which module a given symbol belongs to.

All the files in this directory are text files.

Don't hesitate to have a look by yourself!

Compiling the kernel in a nutshell

```
make xconfig
make
sudo make install
sudo make modules_install
```

Compiling and booting Linux

Linux device files

Character device files

- Accessed through a sequential flow of individual characters
- Character devices can be identified by their `c` type (`ls -l`):

```
crw-rw---- 1 root uucp   4,  64 Feb 23 2004 /dev/ttyS0
crw--w---- 1 jdoe tty  136,   1 Feb 23 2004 /dev/pts/1
crw------- 1 root root  13,  32 Feb 23 2004 /dev/input/mouse0
crw-rw-rw- 1 root root   1,   3 Feb 23 2004 /dev/null
```

- Example devices: keyboards, mice, parallel port, IrDA, Bluetooth port, consoles, terminals, sound, video...

Block device files

- Accessed through data blocks of a given size. Blocks can be accessed in any order.
- Block devices can be identified by their `b` type (`ls -l`):

```
brw-rw----   1 root disk     3,   1 Feb 23  2004 hda1
brw-rw----   1 jdoe floppy   2,   0 Feb 23  2004 fd0
brw-rw----   1 root disk     7,   0 Feb 23  2004 loop0
brw-rw----   1 root disk     1,   1 Feb 23  2004 ram1
brw-------   1 root root     8,   1 Feb 23  2004 sda1
```

- Example devices: hard or floppy disks, ram disks, loop devices...

Device major and minor numbers

As you could see in the previous examples,
device files have 2 numbers associated to them:

- First number: *major* number
- Second number: *minor* number
- Major and minor numbers are used by the kernel to bind a driver to the device file. Device file names don't matter to the kernel!
- To find out which driver a device file corresponds to,
  or when the device name is too cryptic,
  see `Documentation/devices.txt`.

Device file creation

- Device files are not created when a driver is loaded.
- They have to be created in advance:

```
sudo mknod /dev/<device> [c|b] <major> <minor>
```

- Examples:

```
sudo mknod /dev/ttyS0 c 4 64
sudo mknod /dev/hda1 b 3 1
```

Practical lab – Configuring and compiling

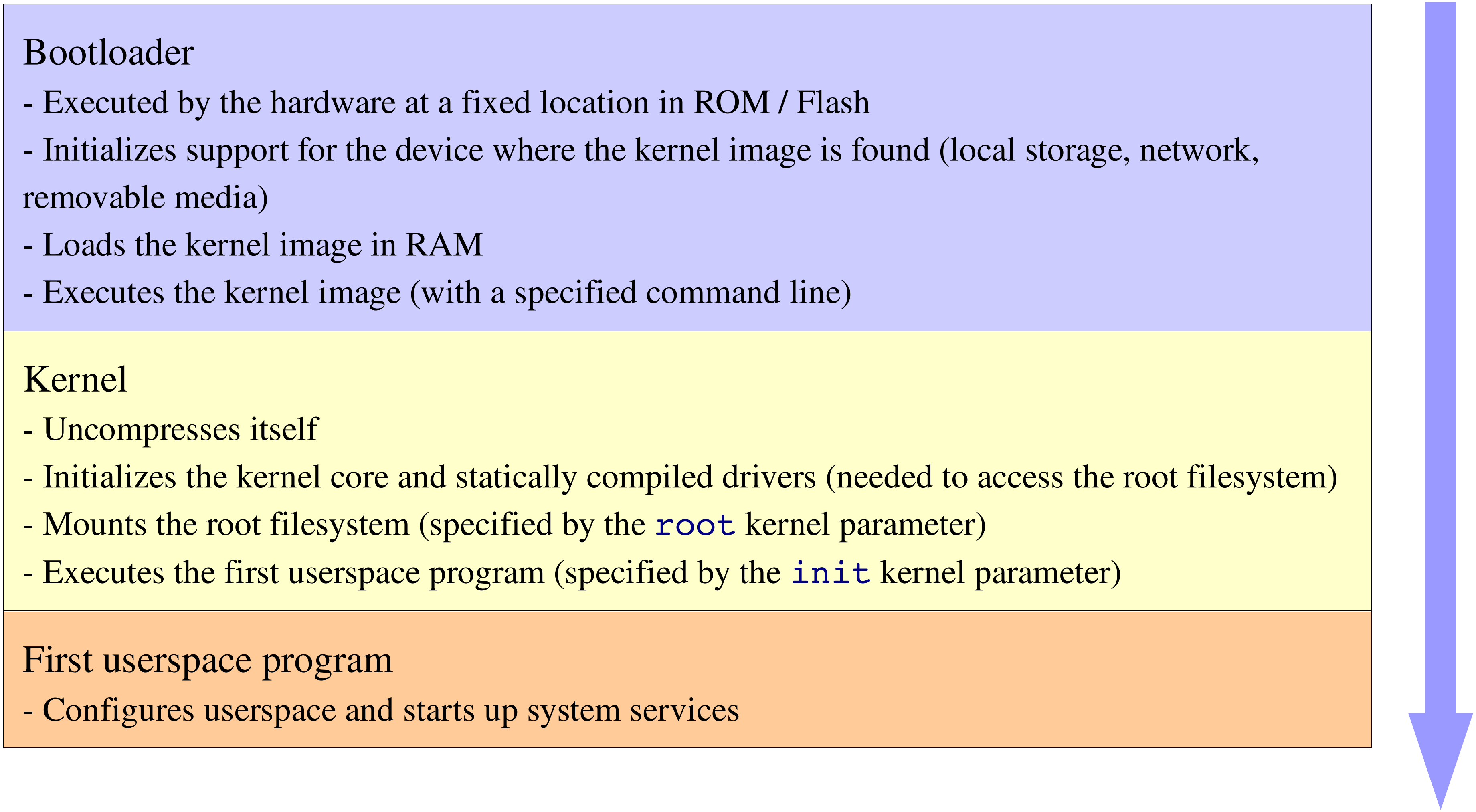
Time to start `Lab 2`!

- Configure your kernel
- Compile it
- Boot it on a virtual PC
- Modify a root filesystem image by adding entries to the `/dev/` directory

Compiling and booting Linux

Overall system startup

Linux 2.4 booting sequence



Bootloader

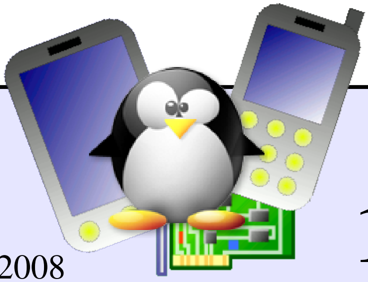
- Executed by the hardware at a fixed location in ROM / Flash
- Initializes support for the device where the kernel image is found (local storage, network, removable media)
- Loads the kernel image in RAM
- Executes the kernel image (with a specified command line)

Kernel

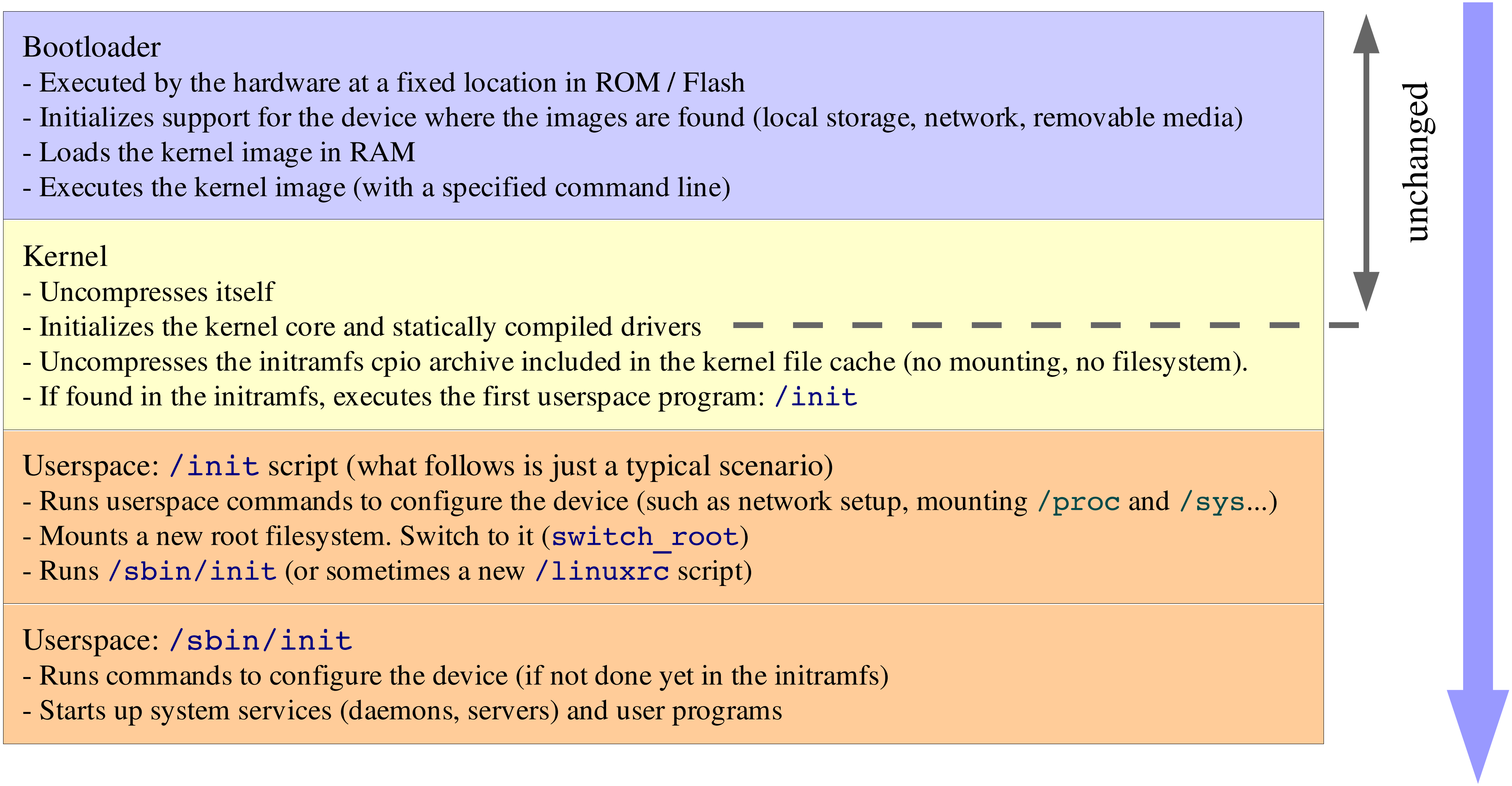
- Uncompresses itself
- Initializes the kernel core and statically compiled drivers (needed to access the root filesystem)
- Mounts the root filesystem (specified by the `root` kernel parameter)
- Executes the first userspace program (specified by the `init` kernel parameter)

First userspace program

- Configures userspace and starts up system services



Linux 2.6 booting sequence



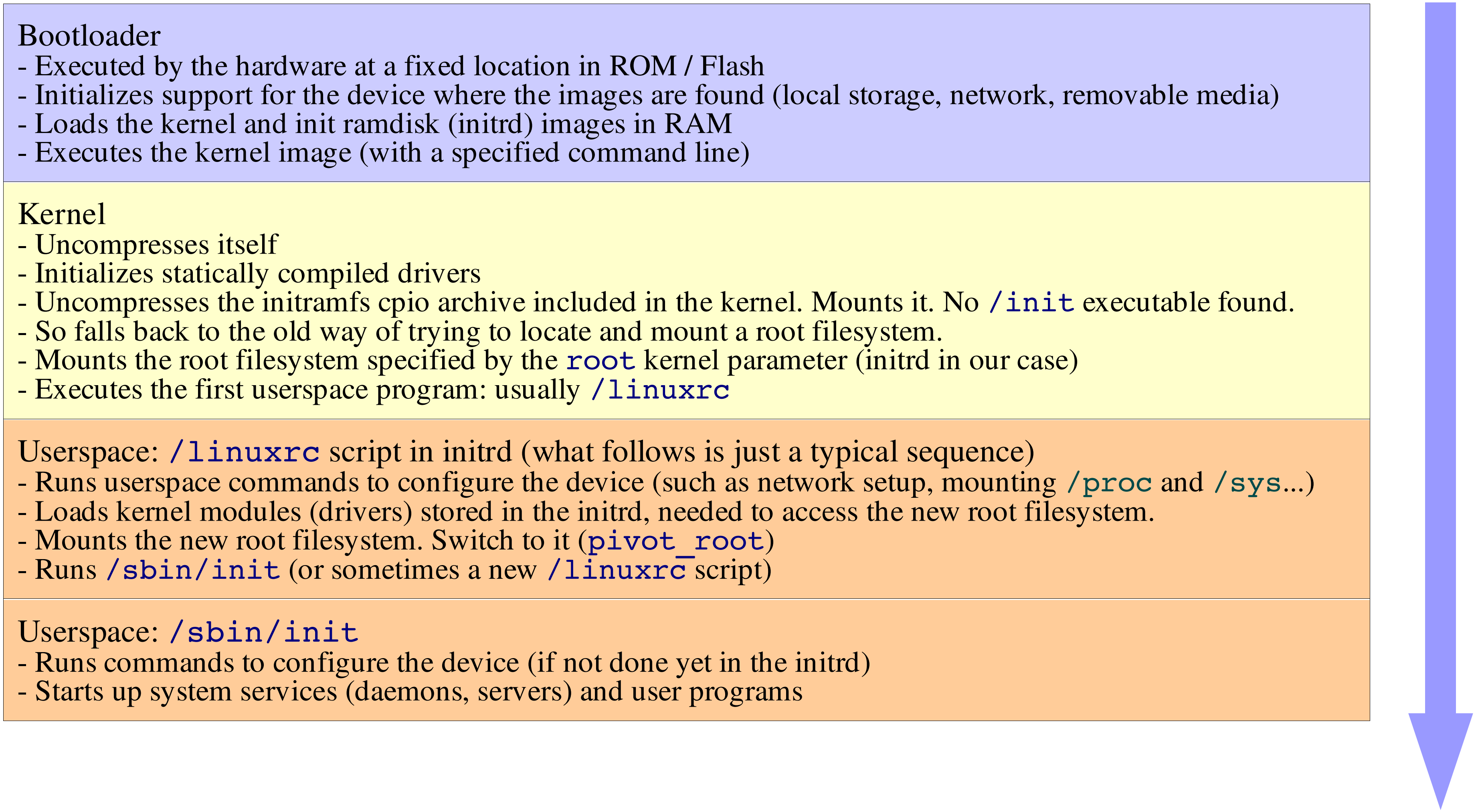
Bootloader
- Executed by the hardware at a fixed location in ROM / Flash
- Initializes support for the device where the images are found (local storage, network, removable media)
- Loads the kernel image in RAM
- Executes the kernel image (with a specified command line)

Kernel
- Uncompresses itself
- Initializes the kernel core and statically compiled drivers
- Uncompresses the initramfs cpio archive included in the kernel file cache (no mounting, no filesystem).
- If found in the initramfs, executes the first userspace program: `/init`

Userspace: `/init` script (what follows is just a typical scenario)
- Runs userspace commands to configure the device (such as network setup, mounting `/proc` and `/sys`...)
- Mounts a new root filesystem. Switch to it (`switch_root`)
- Runs `/sbin/init` (or sometimes a new `/linuxrc` script)

Userspace: `/sbin/init`
- Runs commands to configure the device (if not done yet in the initramfs)
- Starts up system services (daemons, servers) and user programs

Linux 2.6 booting sequence with initrd



Bootloader
- Executed by the hardware at a fixed location in ROM / Flash
- Initializes support for the device where the images are found (local storage, network, removable media)
- Loads the kernel and init ramdisk (initrd) images in RAM
- Executes the kernel image (with a specified command line)

Kernel
- Uncompresses itself
- Initializes statically compiled drivers
- Uncompresses the initramfs cpio archive included in the kernel. Mounts it. No `/init` executable found.
- So falls back to the old way of trying to locate and mount a root filesystem.
- Mounts the root filesystem specified by the `root` kernel parameter (initrd in our case)
- Executes the first userspace program: usually `/linuxrc`

Userspace: `/linuxrc` script in initrd (what follows is just a typical sequence)
- Runs userspace commands to configure the device (such as network setup, mounting `/proc` and `/sys`...)
- Loads kernel modules (drivers) stored in the initrd, needed to access the new root filesystem.
- Mounts the new root filesystem. Switch to it (`pivot_root`)
- Runs `/sbin/init` (or sometimes a new `/linuxrc` script)

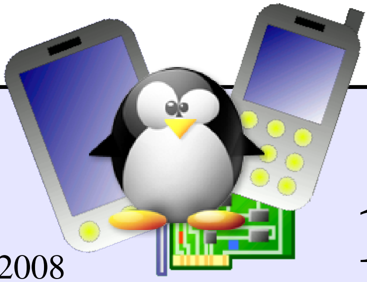
Userspace: `/sbin/init`
- Runs commands to configure the device (if not done yet in the initrd)
- Starts up system services (daemons, servers) and user programs

Linux 2.4 booting sequence drawbacks

Trying to mount the filesystem specified by the `root` kernel parameter is complex:

- Need device and filesystem drivers to be loaded
- Specifying the root filesystem requires ugly black magic device naming (such as `/dev/ram0`, `/dev/hda1`...), while `/` doesn't exist yet!
- Can require a complex initialization to implement within the kernel. Examples: NFS (set up an IP address, connect to the server...), RAID (root filesystem on multiple physical drives)...

In a nutshell: too much complexity in kernel code!



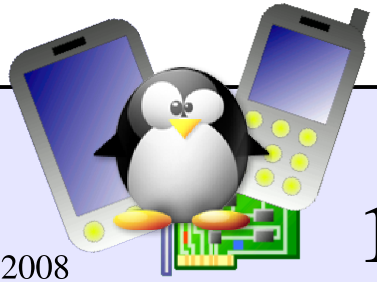
Extra init ramdisk drawbacks

Init ramdisks are implemented as standard block devices

- Need a ramdisk and filesystem driver
- Fixed in size: cannot easily grow in size.
  Any free space cannot be reused by anything else.
- Needs to be created and modified like any block device:
  formatting, mounting, editing, unmounting.
  Root permissions needed.
- Like in any block device, files are first read from the storage,
  and then copied to the file cache.
  Slow and duplication in RAM!!!

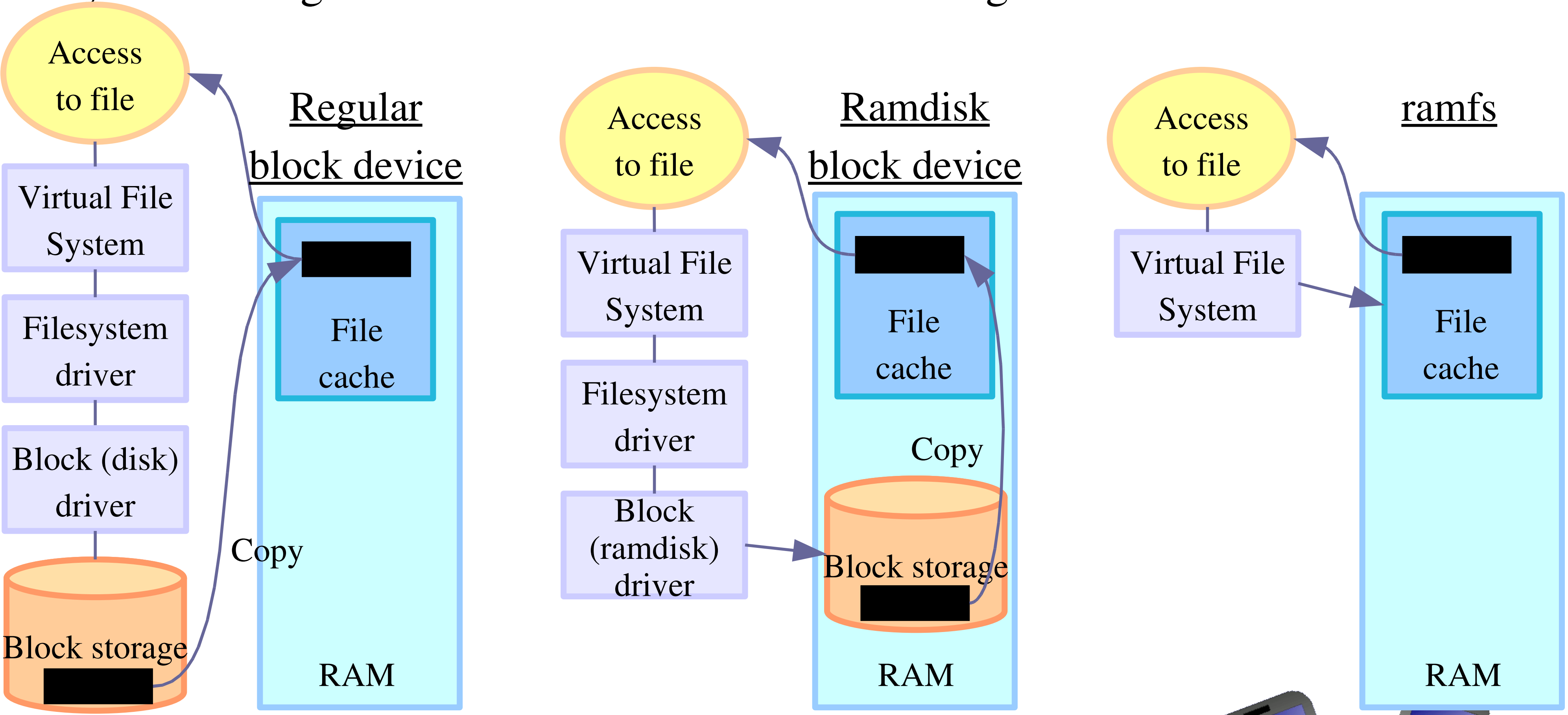
Initramfs features and advantages (1)

- Root file system built in the kernel image (embedded as a compressed `cpio` archive)
- Very easy to create (at kernel build time). No need for root permissions (for `mount` and `mknod`).
- Compared to init ramdisks, just 1 file to handle in the bootloader.
- Always present in the Linux 2.6 kernel (empty by default).
- Just a plain compressed `cpio` archive. Neither needs a block nor a filesystem driver.



Initramfs features and advantages (2)

ramfs: implemented in the file cache. No duplication in RAM, no filesystem layer to manage. Just uses the size of its files. Can grow if needed.



Access to file

Virtual File System

Filesystem driver

Block (disk) driver

Block storage

Copy

Regular block device

File cache

RAM

Access to file

Virtual File System

Filesystem driver

Block (ramdisk) driver

Ramdisk block device

File cache

Copy

Block storage

RAM

Access to file

Virtual File System

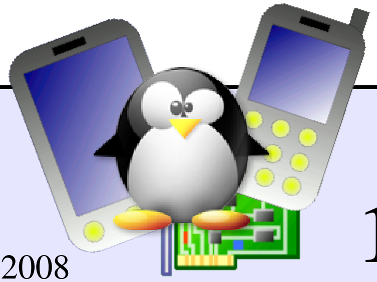
ramfs

File cache

RAM

Initramfs features and advantages (3)

- Loaded by the kernel earlier.
  More initialization code moved to user-space!
- Simpler to mount complex filesystems from flexible userspace scripts rather than from rigid kernel code. More complexity moved out to user-space!
- No more magic naming of the root device.
  `pivot_root` no longer needed.



Initramfs features and advantages (4)

- Possible to add non GPL files (firmware, proprietary drivers) in the filesystem. This is not linking, just file aggregation (not considered as a derived work by the GPL).
- Possibility to remove these files when no longer needed.

More technical details about initramfs:
see `Documentation/filesystems/ramfs-rootfs-initramfs.txt`
and `Documentation/early-userspace/README` in kernel sources.

See also http://www.linuxdevices.com/articles/AT4017834659.html for a nice overview of initramfs (by Rob Landley, the ex-new Busybox maintainer).

How to populate an initramfs

Using `CONFIG_INITRAMFS_SOURCE`
in kernel configuration (`General Setup` section)

- Either give an existing `cpio` archive
- Or give a list of files or directories
  to be added to the archive.
- Or give a text specification file (see next page)

You can build your initramfs with a tiny C library
and the tiny executables it ships:
`klibc`: http://en.wikipedia.org/wiki/Klibc

Initramfs specification file example

major minor

```
dir /dev 755 0 0
nod /dev/console 644 0 0 c 5 1
nod /dev/loop0 644 0 0 b 7 0
dir /bin 755 1000 1000
file /bin/busybox /stuff/initramfs/busybox 755 0 0
slink /bin/sh busybox 777 0 0
dir /proc 755 0 0
dir /sys 755 0 0
dir /mnt 755 0 0
file /init /stuff/initramfs/init.sh 755 0 0
```

permissions

user id group id

No need for `root` user access!

How to handle compressed cpio archives

Useful when you want to build the kernel with a ready-made cpio archive. Better let the kernel do this for you!

- Extracting:

```
gzip -dc initramfs.img | cpio -id
```

- Creating:

```
find <dir> -print -depth | cpio -ov | gzip -c >
initramfs.img
```

How to create an initrd

In case you really need an initrd (why?).

```
sudo mkdir /mnt/initrd
dd if=/dev/zero of=initrd.img bs=1k count=2048
mkfs.ext2 -F initrd.img
sudo mount -o loop initrd.img /mnt/initrd
```

Fill the ramdisk contents: BusyBox, modules, `/linuxrc` script

More details in the Free Software tools for embedded systems training!

```
sudo umount /mnt/initrd
gzip --best -c initrd.img > initrd
```

More details on `Documentation/initrd.txt` in the kernel sources! Also explains pivot rooting.

Booting variants

XIP (Execute In Place)

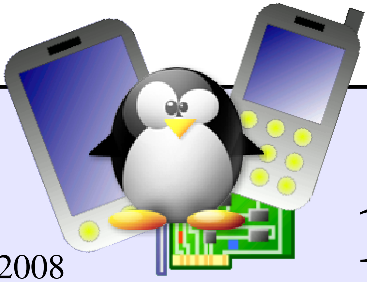
- The kernel image is directly executed from the storage
- Can be faster and save RAM
  However, the kernel image can't be compressed

No initramfs / initrd

- Directly mounting the final root filesystem
  (`root` kernel command line option)

No new root filesystem

- Running the whole system from the initramfs.

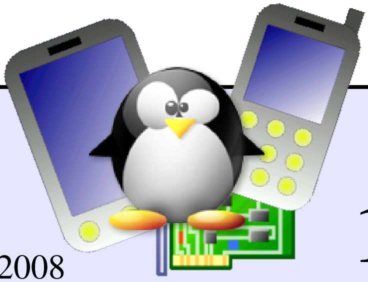


Compiling and booting Linux

Bootloaders

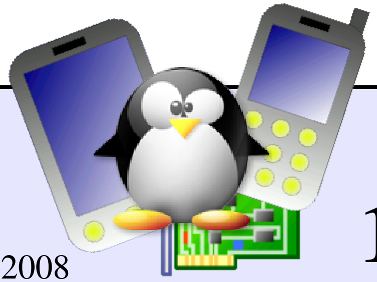
2-stage bootloaders

- At startup, the hardware automatically executes the bootloader from a given location, usually with very little space (such as the boot sector on a PC hard disk)
- Because of this lack of space, 2 stages are implemented:
  - 1st stage: minimum functionality. Just accesses the second stage on a bigger location and executes it.
  - 2nd stage: offers the full bootloader functionality. No limit in what can be implemented. Can even be an operating system itself!



x86 bootloaders

- LILO: LInux LOad. Original Linux bootloader. Now rare.
  http://freshmeat.net/projects/lilo/
  Supports: `x86`
- GRUB: GRand Unified Bootloader from GNU. More powerful.
  http://www.gnu.org/software/grub/
  Supports: `x86`
  See our Grub details annex for details.
- SYSLINUX: Utilities for network and removable media booting
  http://syslinux.zytor.com
  Supports: `x86`



Generic bootloaders

- Das U-Boot: Universal Bootloader from Denx Software

  The most used on `arm`.

  http://www.denx.de/wiki/UBoot/WebHome

  Supports: `arm`, `ppc`, `mips`, `x86`, `m68k`, `nios`...

  See our U-boot details annex for details.

- RedBoot: eCos based bootloader from Red-Hat

  http://sources.redhat.com/redboot/

  Supports: `x86`, `arm`, `ppc`, `mips`, `sh`, `m68k`...

- uMon: MicroMonitor general purpose, multi-OS bootloader

  http://microcross.com/html/micromonitor.html

  Supports: ARM, ColdFire, SH2, m68k, MIPS, PowerPC, Xscale...

Other bootloaders

- LAB: Linux As Bootloader, from Handhelds.org
  http://handhelds.org/cgi-bin/cvsweb.cgi/linux/kernel26/lab/
  Idea: use a trimmed Linux kernel with only features needed in a bootloader (no scheduling, etc.). Reuses flash and filesystem access, LCD interface, without having to implement bootloader specific drivers.
  Supports: `arm` (still experimental)
- And many more: lots of platforms have their own!

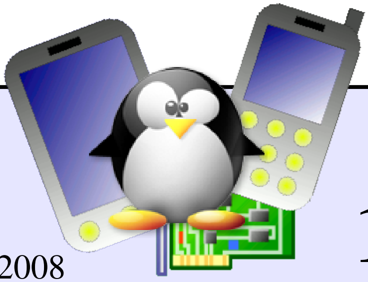
Compiling and booting Linux

Kernel booting

Kernel command line parameters

Like most C programs, the Linux kernel accepts command line arguments

- Kernel command line arguments are part of the bootloader configuration settings.
- Useful to modify the behavior of the kernel at boot time, without having to recompile it.
- Useful to perform advanced kernel and driver initialization, without having to use complex user-space scripts.

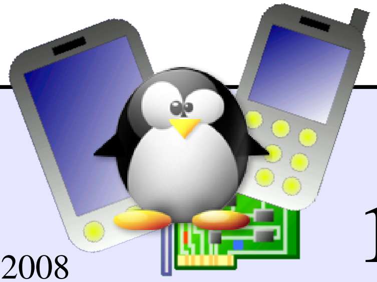


Kernel command line example

HP iPAQ h2200 PDA booting example:

`root=/dev/ram0 \`	Root filesystem (first ramdisk)
`rw \`	Root filesystem mounting mode
`init=/linuxrc \`	First userspace program
`console=ttyS0,115200n8 \`	Console (serial)
`console=tty0 \`	Other console (framebuffer)
`ramdisk_size=8192 \`	Misc parameters...
`cachepolicy=writethrough`	

Hundreds of command line parameters described on
`Documentation/kernel-parameters.txt`



Usefulness of rootfs on NFS

Once networking works, your root filesystem could be a directory on your GNU/Linux development host, exported by NFS (Network File System). This is very convenient for system development:

- Makes it very easy to update files (driver modules in particular) on the root filesystem, without rebooting. Much faster than through the serial port.
- Can have a big root filesystem even if you don't have support for internal or external storage yet.
- The root filesystem can be huge. You can even build native compiler tools and build all the tools you need on the target itself (better to cross-compile though).

NFS boot setup (1)

On the host (NFS server)

- Add the below line to your `/etc/exports` file:

```
/home/rootfs 192.168.0.202(rw,no_root_squash,no_subtree_check)
```

client address NFS server options

- Start or restart your NFS server (Example: Debian, Ubuntu)

```
/etc/init.d/nfs-kernel-server restart
```

NFS boot setup (2)

On the target (NFS client)

- Compile your kernel with `CONFIG_NFS_FS=y`, `CONFIG_IP_PNP=y` (configure IP at boot time) and `CONFIG_ROOT_NFS=y`
- Boot the kernel with the below command line options:

```
root=/dev/nfs
         virtual device
ip=192.168.1.111:192.168.1.110:192.168.1.100:255.255.255.0:at91:eth0
       local IP address    server IP address         gateway      netmask     hostname device
nfsroot=192.168.1.110:/home/nfsroot
           NFS server IP address   Directory on the NFS server
```

First user-space program

- Specified by the `init` kernel command line parameter
  Examples: `init=/bin/sh` or `init=/sbin/init`
- Executed at the end of booting by the kernel
- Takes care of starting all other user-space programs (system services and user programs).
- Gets the `1` process number (pid)
  Parent or ancestor of all user-space programs
  The system won't let you kill it.

/linuxrc

- Program executed by default when booting from an init ramdisk and no `init` parameter is given to the kernel.
- Is most of the time a shell script, based on a very lightweight shell such as `nash` and `busybox sh`
- This script can implement complex tasks: detecting drivers to load, setting up networking, mounting partitions, switching to a new root filesystem...

The init program

- `/sbin/init` is the second default init program
- Takes care of starting system services, and eventually the user interfaces (`sshd`, X server...)
- Also takes care of stopping system services
- Lightweight, partial implementation available through BusyBox.

See the Init runlevels annex section for more details about starting and stopping system services with `init`.

However, simple startup scripts are often sufficient in embedded systems.

Embedded Linux driver development

Compiling and booting Linux

Cross-compiling the kernel

Cross-compiling the kernel

When you compile a Linux kernel for another CPU architecture

- Much faster than compiling natively, when the target system is much slower than your GNU/Linux workstation.
- Much easier as development tools for your GNU/Linux workstation are much easier to find.
- To make the difference with a native compiler, cross-compiler executables are prefixed by the name of the target system, architecture and sometimes library. Examples:

```
mips-linux-gcc
m68k-linux-uclibc-gcc
arm-linux-gnueabi-gcc
```

Specifying a cross-compiler (1)

The CPU architecture and cross-compiler prefix are defined through the `ARCH` and `CROSS_COMPILE` variables in the toplevel `Makefile`.

- The `Makefile` defines `CC = $(CROSS_COMPILE)gcc`
  See comments in `Makefile` for details

- The easiest solution is to modify the `Makefile`.
  Example, ARM platform, cross-compiler: `arm-linux-gcc`

```
ARCH    ?= arm
CROSS_COMPILE  ?= arm-linux-
```

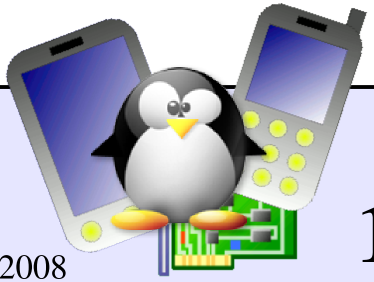
Specifying a cross-compiler (2)

Another solution is to set ARCH and CROSS_COMPILE through the make command line

- Explanation: any variable set through the make command line overrides any setting in the Makefile.
- Examples:

```
make ARCH=sh CROSS_COMPILE=sh-linux- xconfig
make ARCH=sh CROSS_COMPILE=sh-linux-
make ARCH=sh CROSS_COMPILE=sh-linux- modules_install
```

- Big drawback:
  You should never forget these settings when you run make!
  That's error prone and not convenient at all.



Specifying a cross compiler (3)

Another solution: set `ARCH` and `CROSS_COMPILE` as environment variables in your terminal:

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-
```

- Can be set in project specific environments.
- Not hard-coded in the Makefile.
  Do not interfere with patches.
- You don't forget to set them when you run any `make` command.
- Caution: only apply to shells
  in which these variables have been set.

Configuring the kernel

```
make xconfig
```

- Same as in native compiling.
- Don't forget to set the right board / machine type!

Ready-made config files

```
assabet_defconfig     integrator_defconfig  mainstone_defconfig
badge4_defconfig      iq31244_defconfig     mx1ads_defconfig
bast_defconfig        iq80321_defconfig     neponset_defconfig
cerfcube_defconfig    iq80331_defconfig     netwinder_defconfig
clps7500_defconfig    iq80332_defconfig     omap_h2_1610_defconfig
ebsa110_defconfig     ixdp2400_defconfig    omnimeter_defconfig
edb7211_defconfig     ixdp2401_defconfig    pleb_defconfig
enp2611_defconfig     ixdp2800_defconfig    pxa255-idp_defconfig
ep80219_defconfig     ixdp2801_defconfig    rpc_defconfig
epxa10db_defconfig    ixp4xx_defconfig      s3c2410_defconfig
footbridge_defconfig  jornada720_defconfig  shannon_defconfig
fortunet_defconfig    lart_defconfig        shark_defconfig
h3600_defconfig       lpd7a400_defconfig    simpad_defconfig
h7201_defconfig       lpd7a404_defconfig    smdk2410_defconfig
h7202_defconfig       lubbock_defconfig     versatile_defconfig
hackkit_defconfig     lusl7200_defconfig
```

`arch/arm/configs` example

Using ready-made config files

- Default configuration files available for many boards / machines! Check if one exists in `arch/<arch>/configs/` for your target.
- Example: if you found an `acme_defconfig` file, you can run:
  `make acme_defconfig`
- Using `arch/<arch>/configs/` is a very good good way of releasing a default configuration file for a group of users or developers.

Like all `make` commands, you must run `make <machine>_defconfig` in the toplevel source directory.

Cross-compiling setup

Example

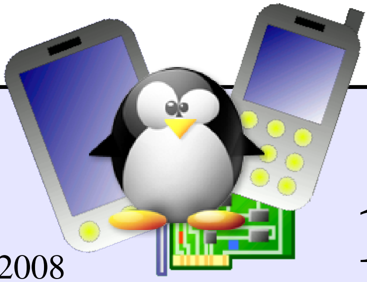
- If you have an ARM cross-compiling toolchain in `/usr/local/arm/3.3.2/`
- You just have to add it to your Unix search path:

```
export PATH=/usr/local/arm/3.3.2/bin:$PATH
```

  (Caution: the scope of this definition is limited to the current shell).

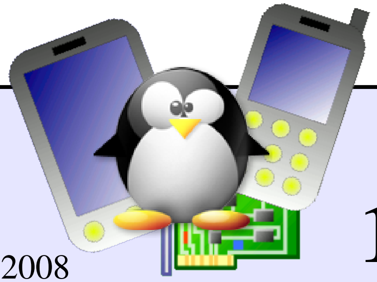
Choosing a toolchain

- See the `Documentation/Changes` file in the sources for details about minimum tool versions requirements.
- More about toolchains: Free Software tools for embedded systems training: http://free-electrons.com/training/devtools/



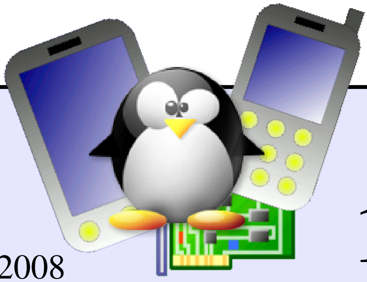
Building the kernel

- Run
  ```
  make
  ```
- Copy
  ```
  arch/<arch>/boot/zImage
  ```
  to the target storage
- You can customize `arch/<arch>/boot/install.sh` so that `make install` does this automatically for you.
- `make INSTALL_MOD_PATH=<dir>/ modules_install`
  and copy `<dir>/` to `/lib/modules/` on the target storage



Cross-compiling summary

- Edit `Makefile`: set `ARCH` and `CROSS_COMPILE`
- Get the default configuration for your machine:
  `make <machine>_defconfig` (if existing in `arch/<arch>/configs`)
- Refine the configuration settings according to your requirements:
  `make xconfig`
- Add the cross-compiler path to your `PATH` environment variable
- Compile the kernel: `make`
- Copy the kernel image from `arch/<arch>/boot/` to the target
- Copy modules to a directory which you replicate on the target:
  `make INSTALL_MOD_PATH=<dir> modules_install`



Practical lab – Cross-compiling

Time to start `Lab 3`!

- Set up a cross-compiling environment
- Configure the kernel `Makefile` accordingly
- Cross-compile the kernel for an `arm` target platform

Driver development

Loadable kernel modules

Loadable kernel modules (1)

- 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.
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).

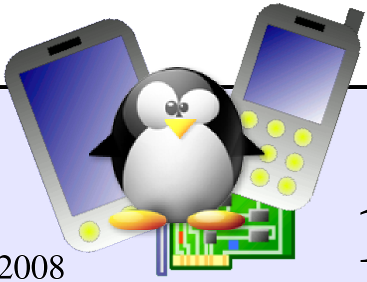
Loadable kernel modules (2)

- Useful to deliver binary-only drivers (bad idea) without having to rebuild the kernel.
- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...

Module dependencies

- Module dependencies stored in `/lib/modules/<version>/modules.dep`
- They are automatically computed during kernel building from module exported symbols. `module2` depends on `module1` if `module2` uses a symbol exported by `module1`.
- 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

```
/* 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 usefulness

- 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`)
- Useful for GNU/Linux distributors for their release policy checks.

Possible module license strings

Available license strings explained in `include/linux/module.h`

- `GPL`

  GNU Public License v2 or later

- `GPL v2`

  GNU Public License v2

- `GPL and additional rights`

- `Dual MIT/GPL`

  GNU Public License v2 or MIT

- `Dual BSD/GPL`

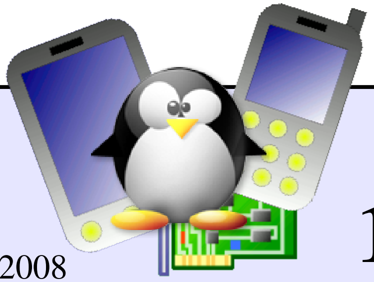
  GNU Public License v2 or BSD

- `Dual MPL/GPL`

  GNU Public License v2 or Mozilla

- `Proprietary`

  Non free products



Compiling a module

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

```
# 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
```

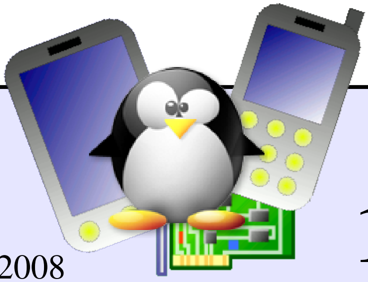
`[Tab]`!
(no spaces)

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

Kernel log

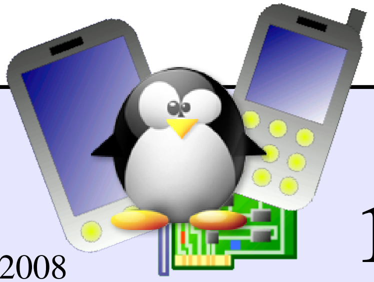
- Of course, the kernel doesn't store its log into a file!
  Files belong to user space.
- The kernel keeps `printk` messages in a circular buffer
  (so that doesn't consume more memory with many messages)
- Kernel log messages can be accessed from user space through system calls, or through `/proc/kmsg`
- Kernel log messages are also displayed in the system console.



Accessing the kernel log

Many ways are available!

- Watch the system console
- `syslogd` / `klogd`
  Daemon gathering kernel messages
  in `/var/log/messages`
  Follow changes by running:
  `tail -f /var/log/messages`
  Caution: this file grows!
  Use `logrotate` to control this
- `cat /proc/kmsg`
  Waits for kernel messages and displays them.
  Useful when none of the above user space programs are available (tiny system)
- `dmesg` ("**d**iagnostic **mes**sa**g**e")
  Found in all systems
  Displays the kernel log buffer



Using the module

- Load the module:

```
sudo insmod ./hello.ko
```

- You will see the following in the kernel log:

```
Good morrow
to this fair assembly
```

- Now remove the module:

```
sudo rmmod hello
```

- You will see:

```
Alas, poor world, what treasure
hast thou lost!
```

Understanding module loading issues

- When loading a module fails, `insmod` often doesn't give you enough details!
- Details are available in the kernel log.
- Example:

```
> sudo insmod ./intr_monitor.ko
insmod: error inserting './intr_monitor.ko': -1
Device or resource busy
> dmesg
[17549774.552000] Failed to register handler for
irq channel 2
```

Module utilities (1)

- `modinfo <module_name>`
  `modinfo <module_path>.ko`
  Gets information about a module: parameters, license, description and dependencies.
  Very useful before deciding to load a module or not.
- `sudo insmod <module_path>.ko`
  Tries to load the given module.

Module utilities (2)

- `sudo modprobe <module_name>`

  Most common usage of `modprobe`: tries to load all the modules the given module depends on, and then this module. Lots of other options are available.

- `lsmod`

  Displays the list of loaded modules

  Compare its output with the contents of `/proc/modules`!

Module utilities (3)

- `sudo rmmod <module_name>`

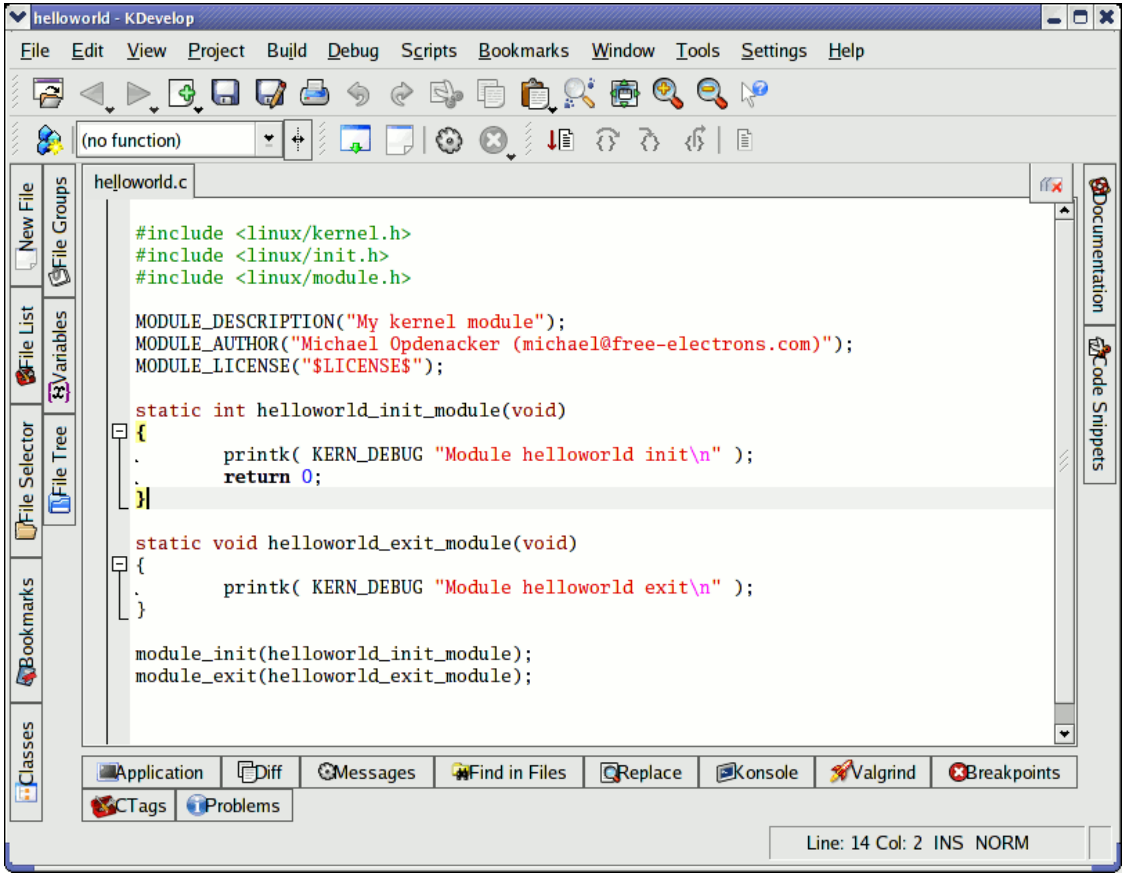
  Tries to remove the given module

- `sudo modprobe -r <module_name>`

  Tries to remove the given module and all dependent modules (which are no longer needed after the module removal)

Create your modules with kdevelop

http://kdevelop.org - Available in most distros.



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

MODULE_DESCRIPTION("My kernel module");
MODULE_AUTHOR("Michael Opdenacker (michael@free-electrons.com)");
MODULE_LICENSE("$LICENSE$");

static int helloworld_init_module(void)
{
        printk( KERN_DEBUG "Module helloworld init\n" );
        return 0;
}

static void helloworld_exit_module(void)
{
        printk( KERN_DEBUG "Module helloworld exit\n" );
}

module_init(helloworld_init_module);
module_exit(helloworld_exit_module);
```

- Makes it easy to create a module code skeleton from a ready-made template.
- Can also be used to compile your module.

Driver development

Module parameters

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`:

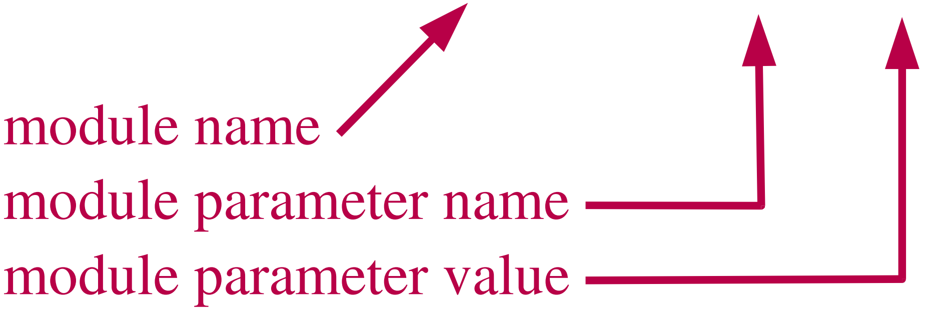
```
sudo insmod ./hello_param.ko howmany=2 whom=universe
```

- 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:

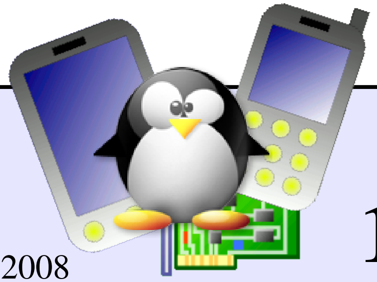
```
options hello_param.howmany=2 hello_param.whom=universe
```



module name

module parameter name

module 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
```

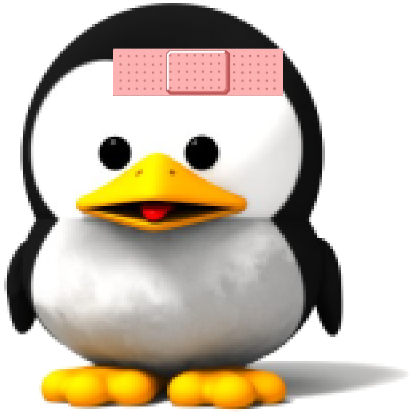
- `Run make xconfig` and see your new options!
- Run `make` and your new files are compiled!
- See `Documentation/kbuild/` for details

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 -Nurp 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.



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

Practical lab – Writing modules

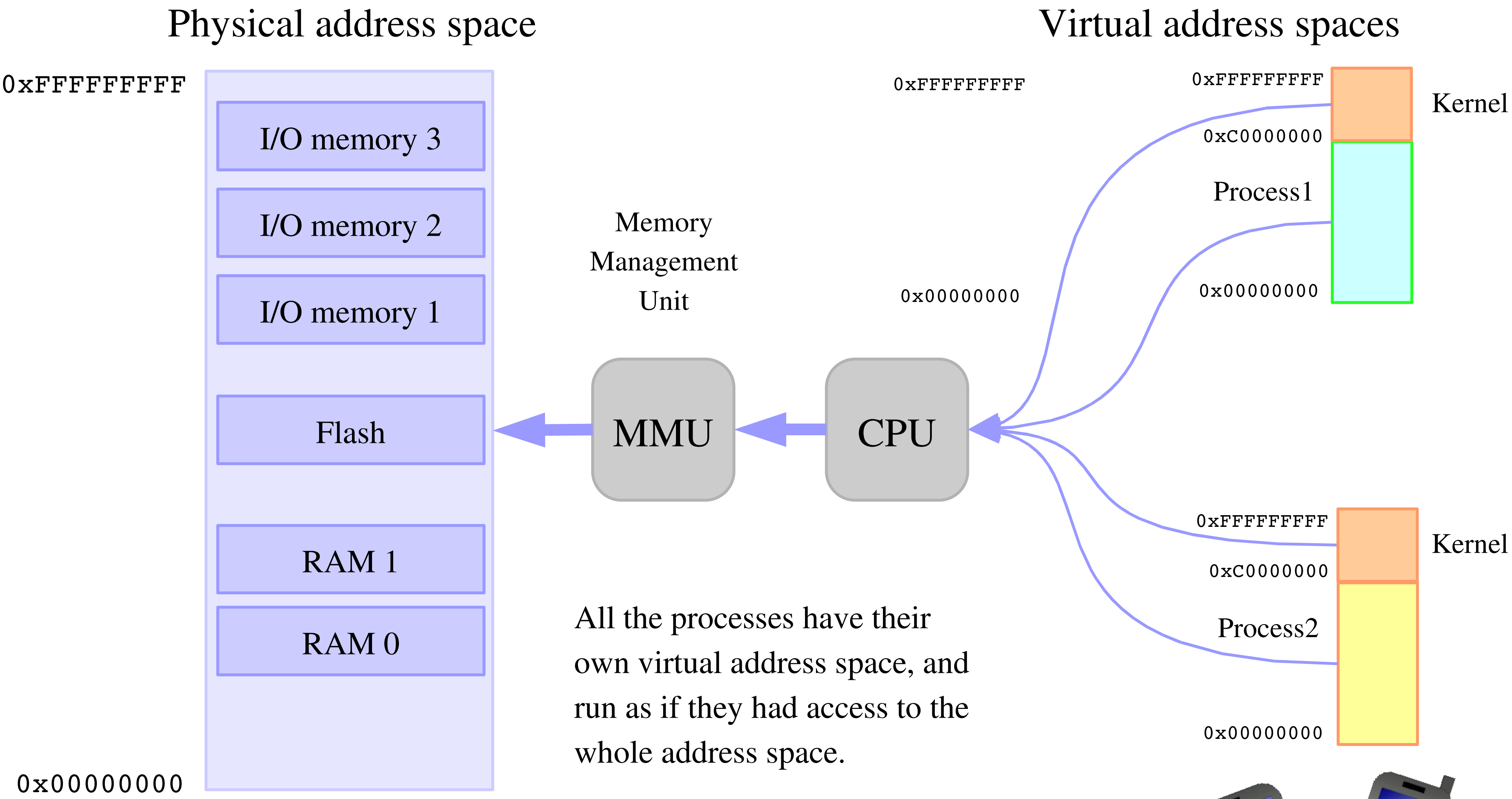
Time to start `Lab 4`!

- Write a kernel module with parameters
- Setup the environment to compile it
- Access kernel internals
- Add a `/proc` interface
- Add the module sources to the kernel source tree
- Create a kernel source patch

Driver development

Memory management

Physical and virtual memory



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

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 (see next page)
- `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 allocation: 32 or 64 bytes (page size dependent).

Main kmalloc flags (1)

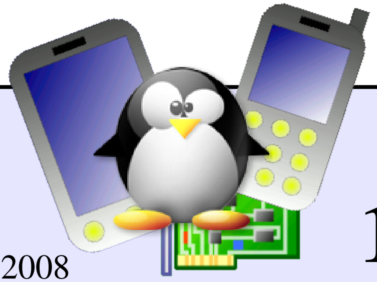
Defined in `include/linux/gfp.h` (GFP: __get_free_pages)

- `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.

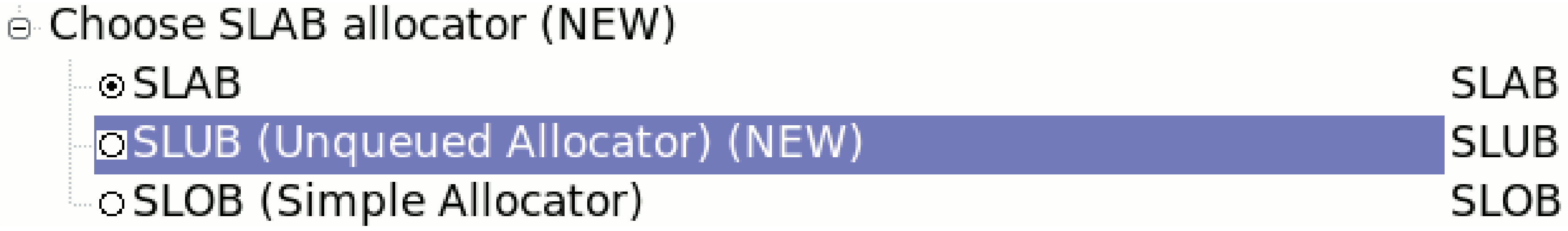
-
  ```
  static inline void *kcalloc(
       size_t n, size_t size, gfp_t flags);
  ```
  Allocates memory for an array of `n` elements of size `size`, and zeroes its contents.

-
  ```
  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.

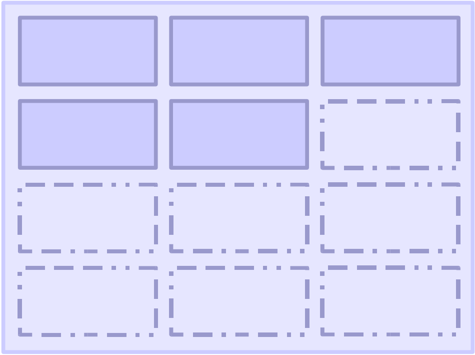


⊖ Choose SLAB allocator (NEW)

● SLAB	SLAB
○ SLUB (Unqueued Allocator) (NEW)	SLUB
○ 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.

See our Slab caches and memory pools annex for details.

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$_2$(`<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`

Memory management - Summary

Small allocations

- `kmalloc`, `kzalloc` (and `kfree`!)
- Slab caches and memory pools

Bigger allocations

- `__get_free_page[s]`, `get_zeroed_page`, `free_page[s]`
- `vmalloc`, `vfree`

Libc like memory utilities

- `memset`, `memcopy`, `memmove`...

Driver development

I/O memory and ports

Requesting I/O ports

`/proc/ioports` example (x86)

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

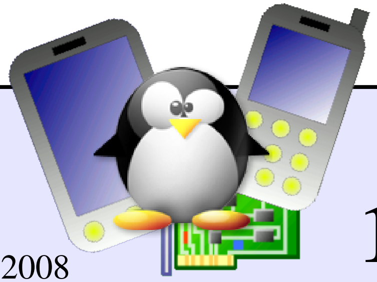
-
  ```
  struct resource *request_region(
      unsigned long start,
      unsigned long len,
      char *name);
  ```

  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`



Reading / writing on I/O ports

The implementation of the below functions

and the exact `unsigned` type can vary from platform to platform!

bytes

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

words

```
unsigned inw(unsigned port);
void outw(unsigned char byte, unsigned port);
```

"long" integers

```
unsigned inl(unsigned port);
void outl(unsigned char byte, unsigned port);
```

Reading / writing strings on I/O ports

Often more efficient than the corresponding C loop,
if the processor supports such operations!

byte strings

```c
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

word strings

```c
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

long strings

```c
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

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
...
```

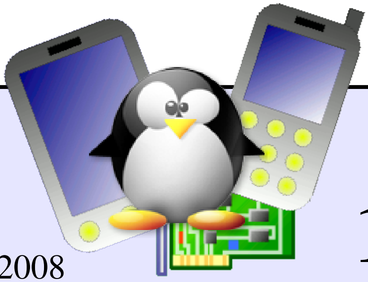
- Equivalent functions with the same interface

-
  ```
  struct resource * request_mem_region(
      unsigned long start,
      unsigned long len,
      char *name);
  ```

-
  ```
  void release_mem_region(
      unsigned long start,
      unsigned long len);
  ```

Choosing I/O ranges

- I/O port and memory ranges can be passed as module parameters. An easy way to define those parameters is through `/etc/modprobe.conf`.
- Modules can also try to find free ranges by themselves (making multiple calls to `request_region` or `request_mem_region`.

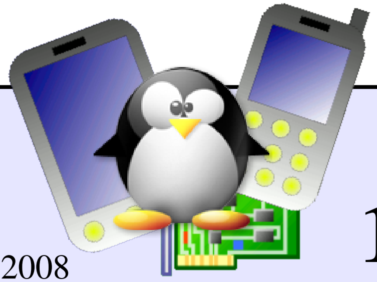


Mapping I/O memory in virtual memory

- 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!



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!

Hardware dependent

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

Safe on all architectures!

Accessing I/O memory

- Directly reading from or writing to addresses returned by `ioremap` ("pointer dereferencing") may not work on some architectures.

- Use the below functions instead. They are always portable and safe:

  `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:

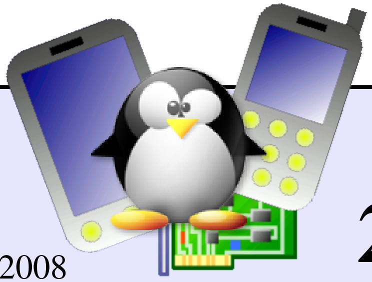
```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
```

- 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);
```

/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.



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! 😉

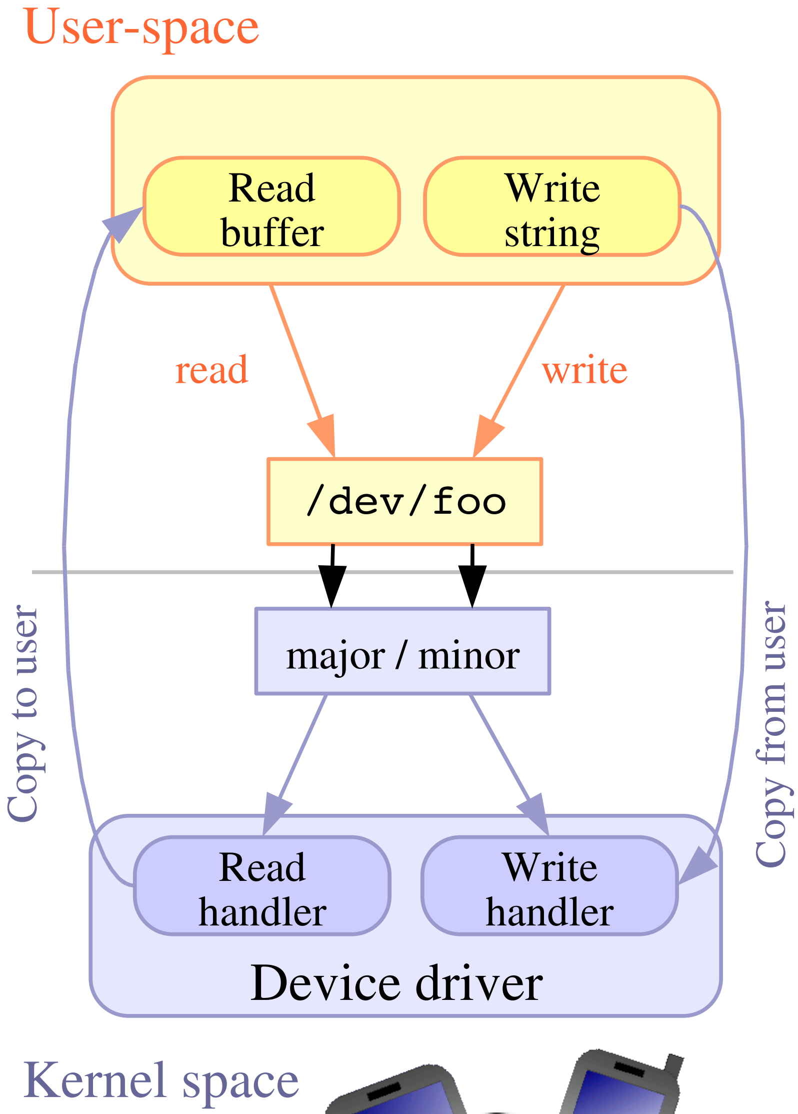
Creating a character driver

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

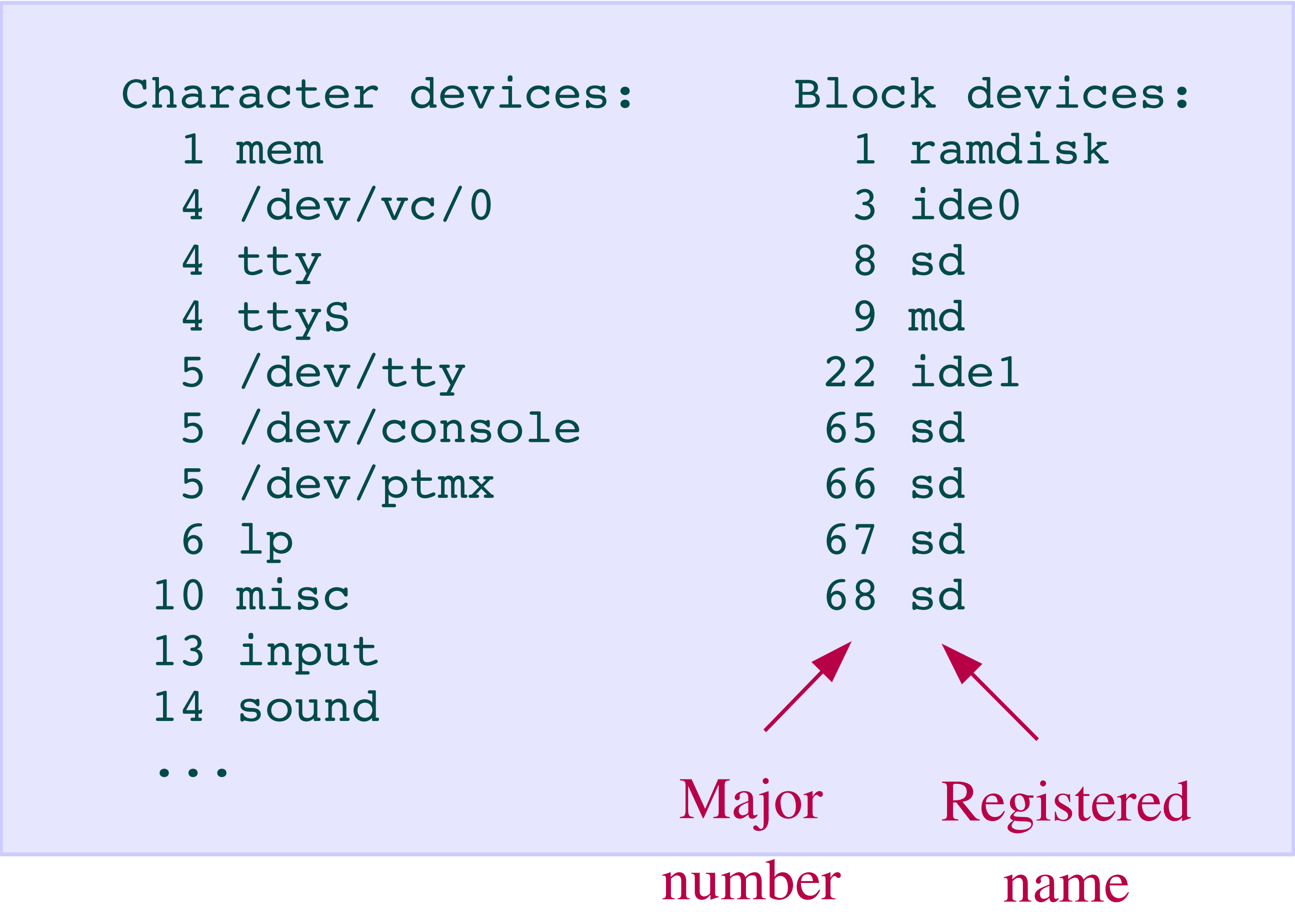
- Need to register one or more device numbers (major / minor pairs), depending on the number of devices managed by the driver.
- Need to find free ones!

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

Major number

Registered name

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 create the device number :

```
MKDEV(int major, int minor);
```

- Macro to extract the minor and major numbers:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

Allocating fixed device numbers

```
#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

```
if (register_chrdev_region(MKDEV(202, 128),
                           acme_count, "acme")) {
  printk(KERN_ERR "Failed to allocate device number\n");
  ...
```

Dynamic allocation of device numbers

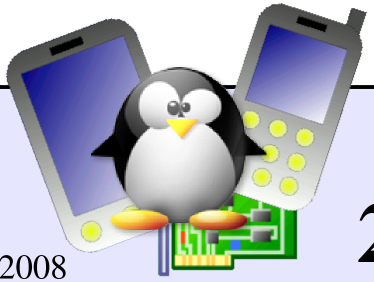
Safer: have the kernel allocate free numbers for you!

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

int alloc_chrdev_region(
        dev_t *dev,                /* Output: starting device number */
        unsigned baseminor,        /* Starting minor number, usually 0 */
        unsigned count,            /* Number of device numbers */
        const char *name);         /* Registered name */
```

Returns `0` if the allocation was successful.

Example

```
if (alloc_chrdev_region(&acme_dev, 0, acme_count, “acme”)) {
   printk(KERN_ERR “Failed to allocate device number\n”);
   ...
```

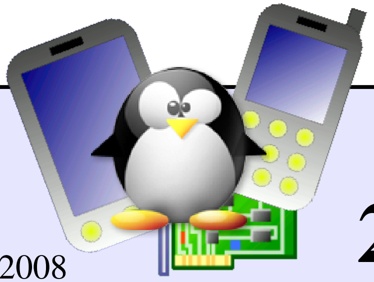


Creating device files

- Issue: you can no longer create /dev entries in advance!
  You have to create them on the fly
  after loading the driver according to the allocated major number.
- Trick: the script loading the module can then use /proc/devices:

```
module=foo; name=foo; device=foo
rm -f /dev/$device
insmod $module.ko
major=`awk "\\$2==\"$name\" {print \\$1}" /proc/devices`
mknod /dev/$device c $major 0
```

- Better: use udev to create the device file automatically.
  See our udev and hotplug section.



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.

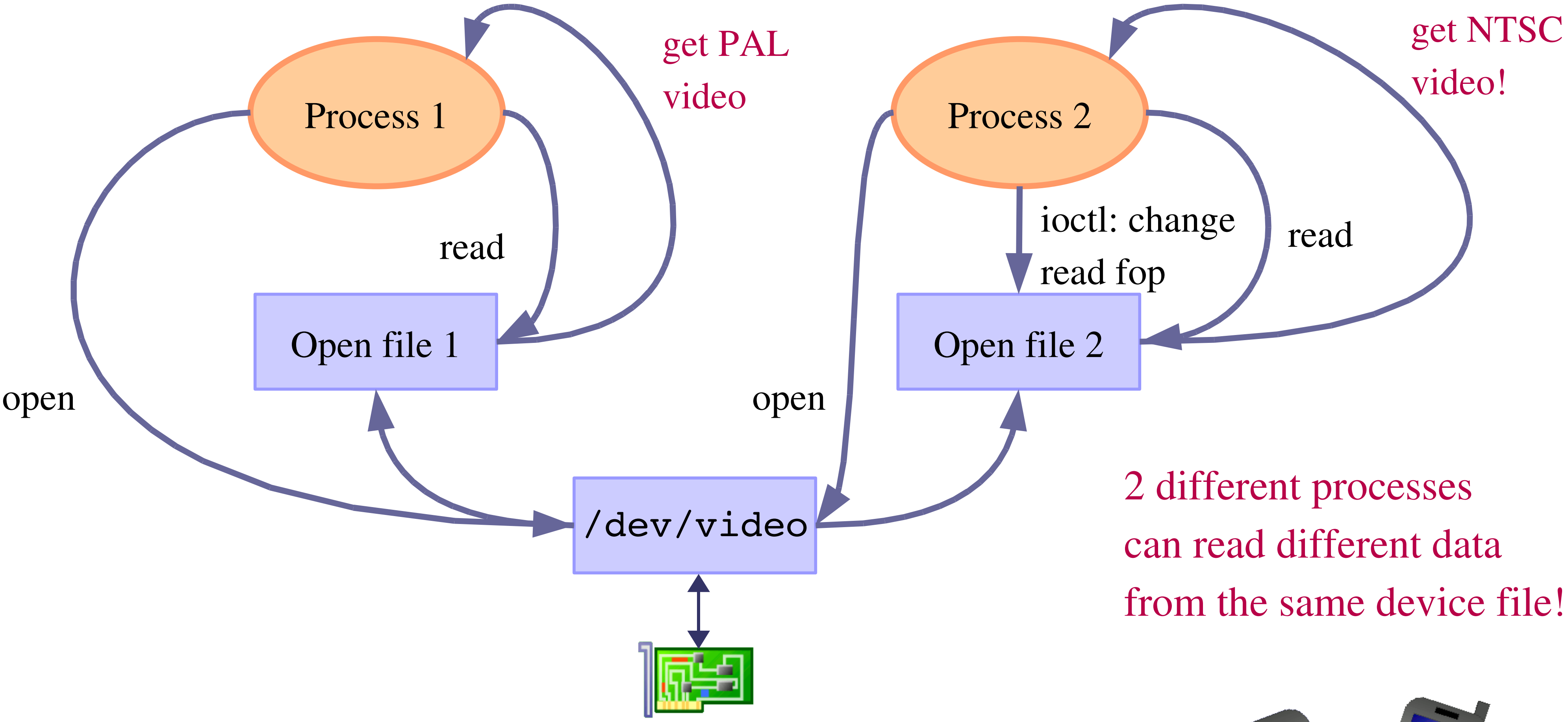
File operations (3)

- `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 specific to each open file!

Using the possibility to redefine file operations for each open file.



2 different processes can read different data from the same device file!

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

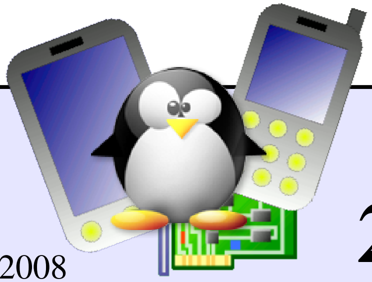
file operations definition example (3)

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;

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
};
```

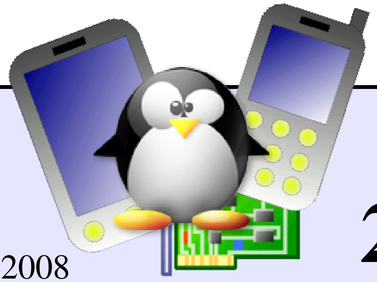
Char driver example summary (2)

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 (alloc_chrdev_region(&acme_dev, 0,
                     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, get major and minor numbers with `alloc_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
- In `/proc/devices`, find the major number it uses.
- Create the device file with this major number

The device file is ready to use!

System user

- Open the device file, read, write, or send ioctl's to it.

User-space

Kernel

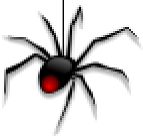
- Executes the corresponding file operations

Kernel

Practical lab – Character drivers

Time to start `Lab 5`!

- Write simple `file_operations`, for a character device, including `ioctl` controls
- Register the character device
- Use the `kmalloc` and `kfree` utilities
- Exchange data with userspace



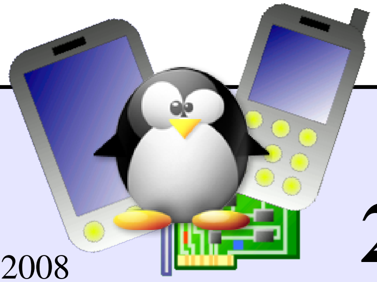
Driver development

Debugging

Usefulness of a serial port

For people porting Linux on consumer devices (no development board)

- Most processors feature a serial port interface (usually very well supported by Linux). Just need this interface to be connected to the outside.
- Easy way of getting the first messages of an early kernel version, even before it boots. A minimum kernel with only serial port support is enough.
- Once the kernel is fixed and has completed booting, possible to access a serial console and issue commands.
- The serial port can also be used to transfer files to the target.



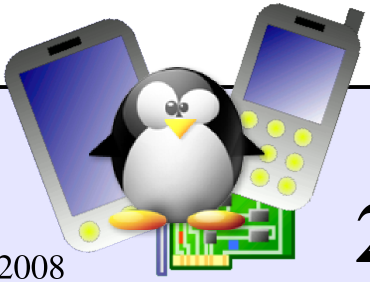
When you don't have a serial port

On the host

- Not an issue. You can get a USB to serial converter. Usually very well supported on Linux and roughly costs $20. The device appears as `/dev/ttyUSB0` on the host.

On the target

- Check whether you have an IrDA port. It's usually a serial port too.
- You may also try to manually hook-up the processor serial interface (check the electrical specifications first!)



Debugging with printk

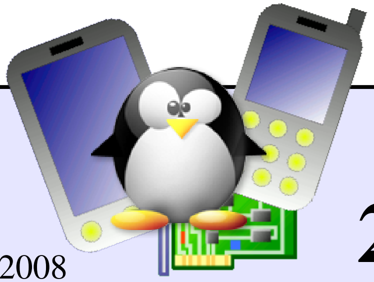
- Universal debugging technique used since the beginning of programming (first found in cavemen drawings)
- Printed or not in the console or `/var/log/messages` according to the priority. This is controlled by the `loglevel` kernel parameter, or through `/proc/sys/kernel/printk` (see `Documentation/sysctl/kernel.txt`)
- 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 (1)

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!



Debugging with /proc or /sys (2)

Examples

- `cat /proc/acme/stats` (dummy example)
  Displays statistics about your acme driver.

- `cat /proc/acme/globals` (dummy example)
  Displays values of global variables used by your driver.

- `echo 600000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed`
  Adjusts the speed of the CPU (controlled by the cpufreq driver).

Debugfs

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

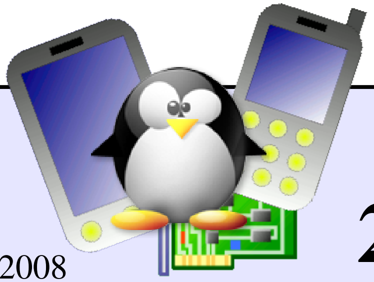
- Kernel configuration: `DEBUG_FS`
  `Kernel hacking -> Debug Filesystem`
- Much simpler to code than an interface in `/proc` or `/sys`.
  The debugging interface disappears when Debugfs is configured out.
- You can mount it as follows:
  ```
  sudo mount -t debugfs none /mnt/debugfs
  ```
- First described 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.



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)

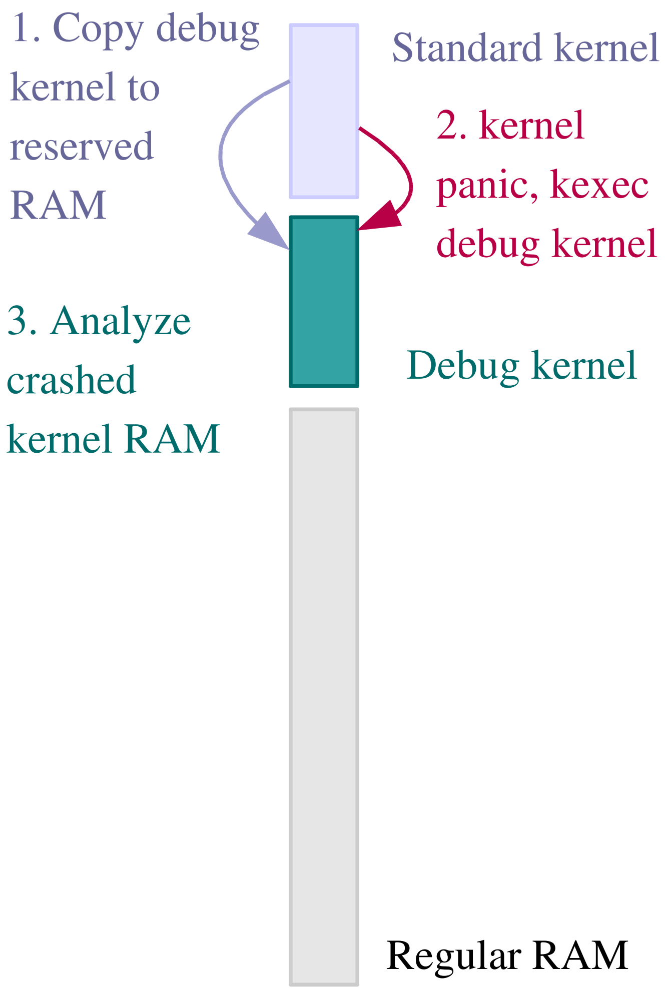
kgdb kernel patch

http://kgdb.linsyssoft.com/

- The execution of the patched kernel is fully controlled by `gdb` from another machine, connected through a serial line.
- Can do almost everything, including inserting breakpoints in interrupt handlers.
- Supported architectures (May 2007 status: version 2.4) `i386`, `x86_64`, `ia64`, `ppc`, `arm` and `mips`.

Kernel crash analysis with kexec

- `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.


1. Copy debug kernel to reserved RAM
Standard kernel
2. kernel panic, kexec debug kernel
3. Analyze crashed kernel RAM
Debug kernel
Regular RAM

Debugging with SystemTap

http://sourceware.org/systemtap/

- Infrastructure to add instrumentation to a running kernel:
  trace functions, read and write variables, follow pointers, gather statistics...
- Eliminates the need to modify the kernel sources to add one's own instrumentation to investigated a functional or performance problem.
- Uses a simple scripting language.
  Several example scripts and probe points are available.
- Based on the Kprobes instrumentation infrastructure.
  See `Documentation/kprobes.txt` in kernel sources.
  Linux 2.6.20: supported on most popular CPUs.
  `arm` and `mips` patches available from http://elinux.org/Patch_Archive

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)

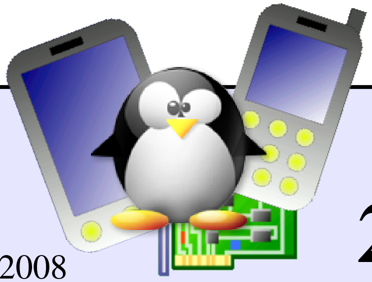
```
#! /usr/bin/env stap

# 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

Other debugging techniques

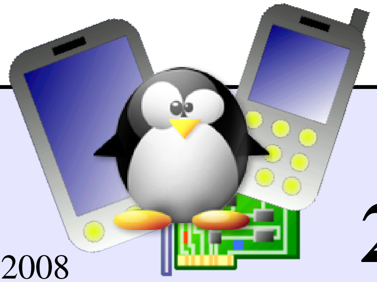
- Use a hardware debugger (JTAG, BDM..)

  If supported by your board and if you have the equipment.



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 (References section)!



Driver development

Processes and scheduling

Processes

A process is an instance of a running program

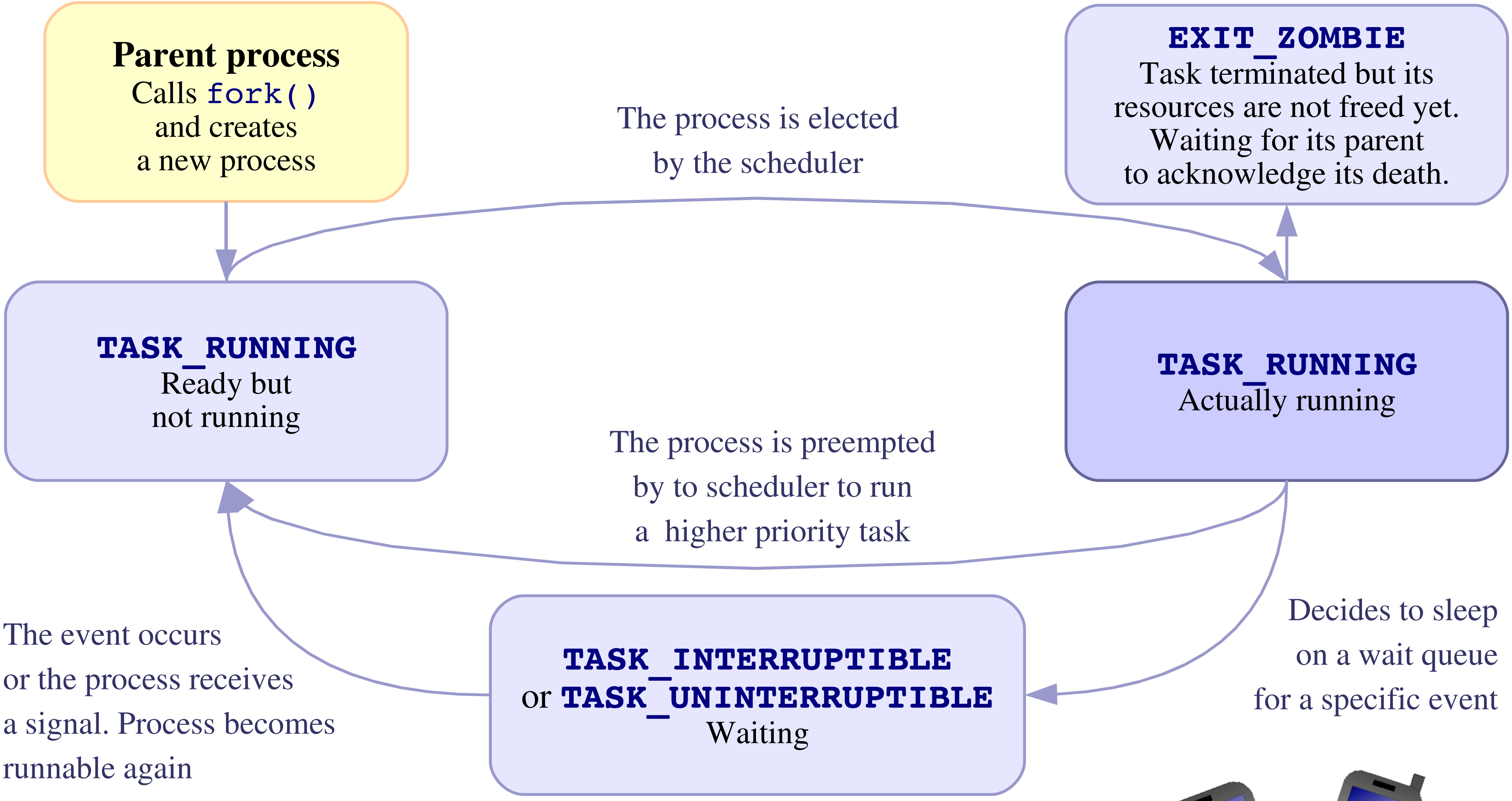
- Multiple instances of the same program can be running. Program code (“text section”) memory is shared.
- Each process has its own data section, address space, processor state, open files and pending signals.
- The kernel has a separate data structure for each process.

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



Parent process
Calls `fork()`
and creates
a new process

The process is elected
by the scheduler

`EXIT_ZOMBIE`
Task terminated but its
resources are not freed yet.
Waiting for its parent
to acknowledge its death.

`TASK_RUNNING`
Ready but
not running

`TASK_RUNNING`
Actually running

The process is preempted
by to scheduler to run
a higher priority task

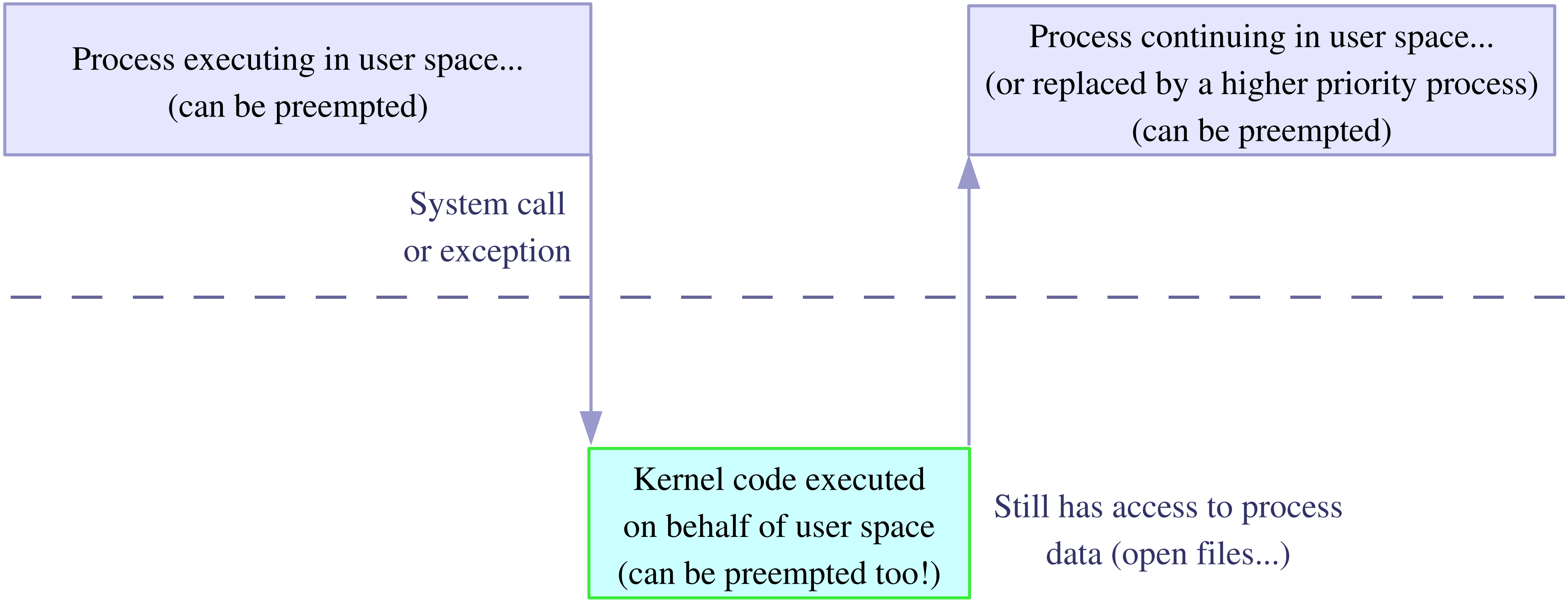
The event occurs
or the process receives
a signal. Process becomes
runnable again

`TASK_INTERRUPTIBLE`
or `TASK_UNINTERRUPTIBLE`
Waiting

Decides to sleep
on a wait queue
for a specific event

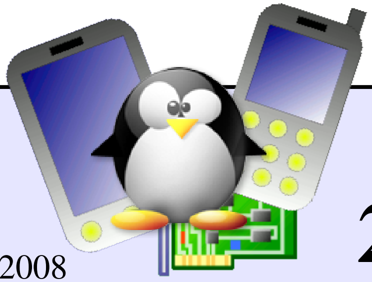
Process context

User space programs and system calls are scheduled together



Kernel threads

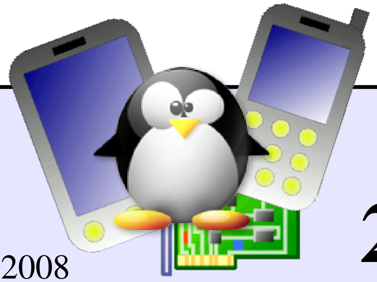
- The kernel does not only react from user-space (system calls, exceptions) or hardware events (interrupts). It also runs its own processes.
- Kernel threads are standard processes scheduled and preempted in the same way (you can view them with `top` or `ps`!) They just have no special address space 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>`



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...

Timer frequency

Timer interrupts are raised every `HZ` th of second (= 1 *jiffy*)

- `HZ` is now configurable (in `Processor type and features`): `100`, `250` (`i386` default), `300` or `1000` (architecture dependent)
  See `kernel/Kconfig.hz`.
- Compromise between system responsiveness and global throughput.
- Caution: not any value can be used. Constraints apply!

Another idea is to completely turn off CPU timer interrupts when the system is idle (“dynamic tick”). This capability is available since 2.6.21, together with high resolution timers.

See our http://free-electrons.com/articles/realtime presentation for details.

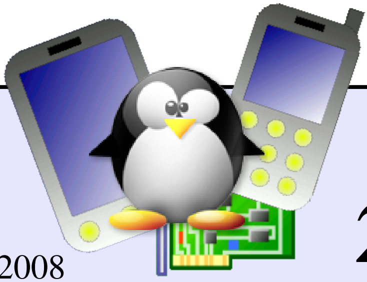
Timeslices

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.



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.

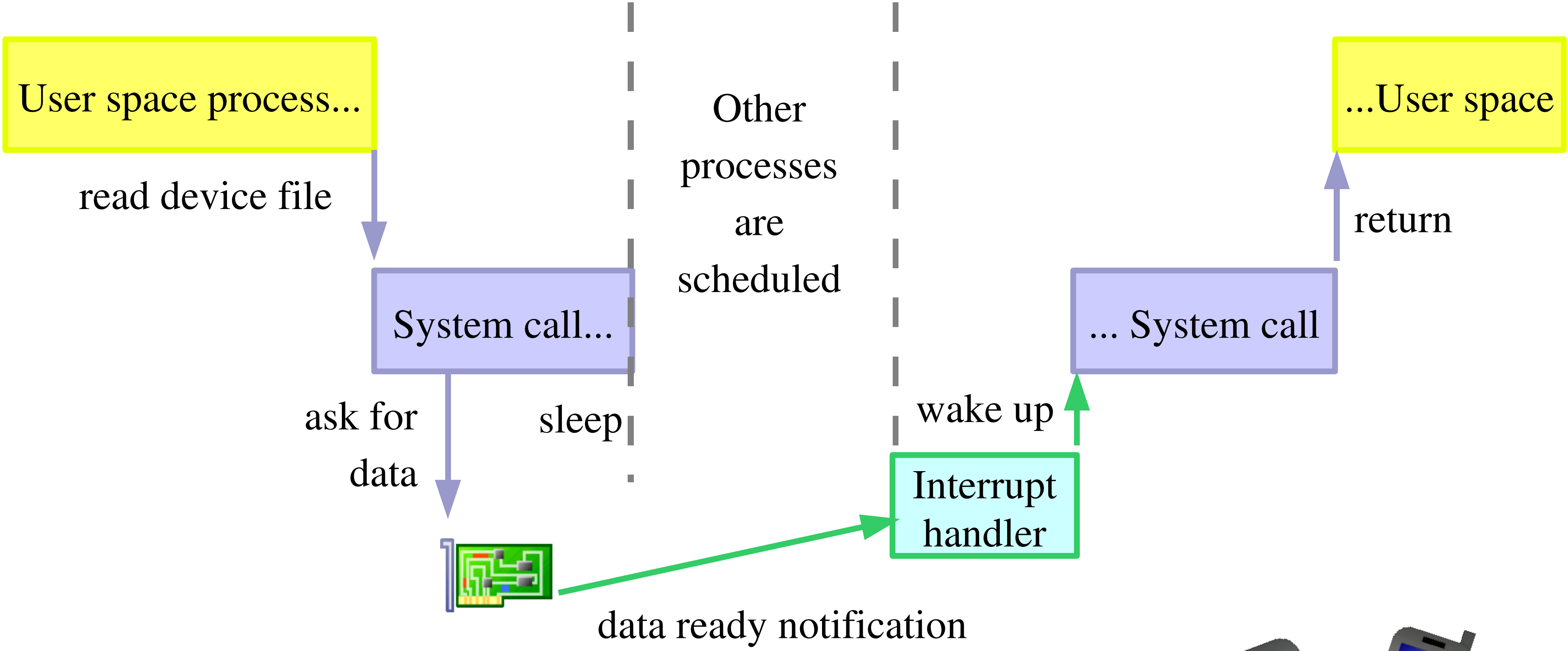
Scheduling also happens when kernel code explicitly runs `schedule()` or executes an action that sleeps.

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 given C expression is true.

  Caution: can't be interrupted (i.e. by killing the client process in user-space)

- `wait_event_interruptible(queue, condition);`

  Can be interrupted

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

  Sleeps and automatically wakes up after the given timeout.

- `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);`
  Does the same job. Usually called when processes waited using `wait_event_interruptible`.

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.

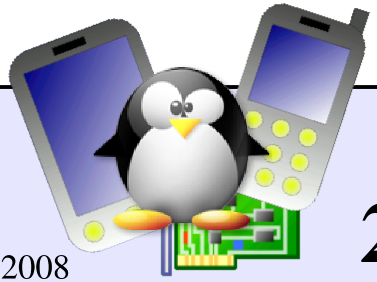
This way, several processes can be woken up at the same time.

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.



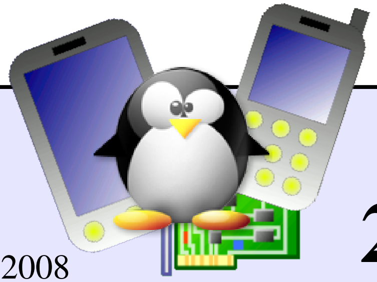
Registering an interrupt handler (1)

Defined in `include/linux/interrupt.h`

-
  ```
  int request_irq(                          Returns 0 if successful
      unsigned int irq,                     Requested irq channel
      irqreturn_t handler,                  Interrupt handler
      unsigned long irq_flags,              Option mask (see next page)
      const char * devname,                 Registered name
      void *dev_id);                        Pointer to some handler data
                         Cannot be NULL and must be unique for shared irqs!
  ```

- `void free_irq( unsigned int irq, void *dev_id);`

- `dev_id` cannot be NULL and must be unique for shared irqs.
  Otherwise, on a shared interrupt line,
  `free_irq` wouldn't know which handler to free.



Registering an interrupt handler (2)

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

- `IRQF_DISABLED`
  "Quick" interrupt handler. Run with all interrupts disabled on the current cpu (instead of just the current line). For latency reasons, should only be used when needed!
- `IRQF_SHARED`
  Run with interrupts disabled only on the current irq line and on the local cpu.
  The interrupt channel can be shared by several devices.
  Requires a hardware status register telling whether an IRQ was raised or not.
- `IRQF_SAMPLE_RANDOM`
  Interrupts can be used to contribute to the system entropy pool used by `/dev/random` and `/dev/urandom`. Useful to generate good random numbers. Don't use this if the interrupt behavior of your device is predictable!

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

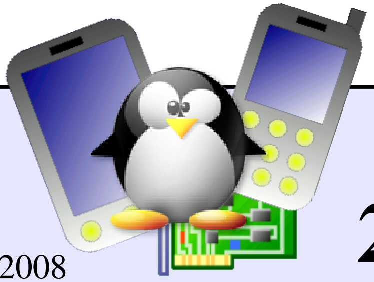
```
cat /proc/stat | grep intr
```

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

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

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 */
                 0                      /* data */
);
```

- 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)

Disabling interrupts

May be useful in regular driver code...

- Can be useful to ensure that an interrupt handler will not preempt your code (including kernel preemption)
- Disabling interrupts on the local CPU:

  ```
  unsigned long flags;
  local_irq_save(flags);      // Interrupts disabled
  ...
  local_irq_restore(flags); // Interrupts restored to their previous state.
  ```

  Note: must be run from within the same function!

Masking out an interrupt line

Useful to disable interrupts **on a particular line**

- `void disable_irq (unsigned int irq);`
  Disables the `irq` line for all processors in the system.
  Waits for all currently executing handlers to complete.
- `void disable_irq_nosync (unsigned int irq);`
  Same, except it doesn't wait for handlers to complete.
- `void enable_irq (unsigned int irq);`
  Restores interrupts on the `irq` line.
- `void synchronize_irq (unsigned int irq);`
  Waits for `irq` handlers to complete (if any).

Checking interrupt status

Can be useful for code which can be run from both process or interrupt context, to know whether it is allowed or not to call code that may sleep.

- `irqs_disabled()`

  Tests whether local interrupt delivery is disabled.

- `in_interrupt()`

  Tests whether code is running in interrupt context

- `in_irq()`

  Tests whether code is running in an interrupt handler.

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 crash 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

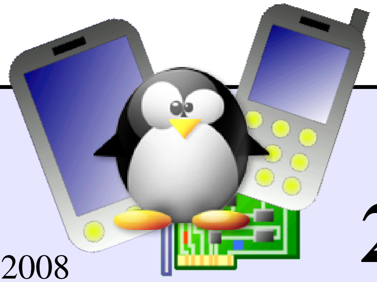
- Called when an interrupt is raised.
- Acknowledge the interrupt
- If needed, schedule a tasklet taking care of handling data. Otherwise, wake up processes waiting for the data.

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.



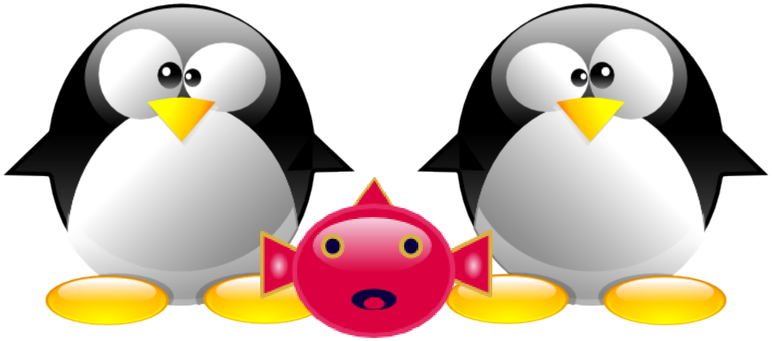
Practical lab – Interrupts

Time to start `Lab 6`!

- Implement a simple interrupt handler
- Register this handler on a shared interrupt line on your GNU/Linux host.
- See how Linux handles shared interrupt lines.

Driver development

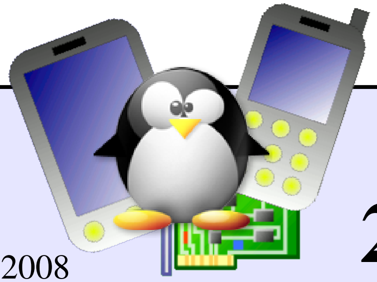
Concurrent access to resources



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.

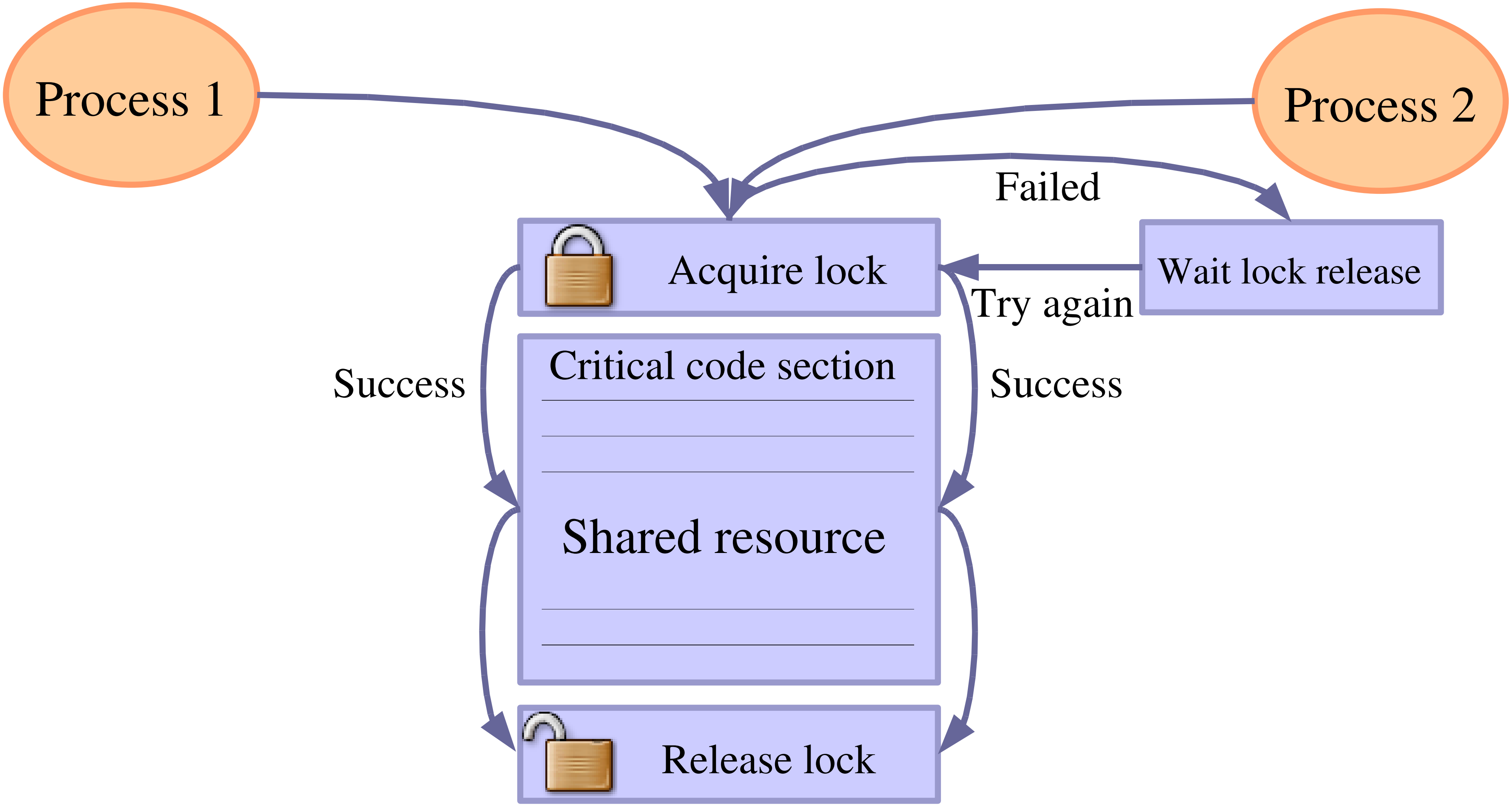


Avoiding concurrency issues

- 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:

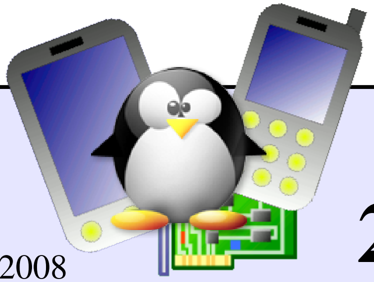
```
#include <linux/mutex.h>
```

- 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_interruptible (struct mutex *lock);`
  Same, but can be interrupted. If interrupted, returns a non zero value and doesn't hold the lock. Test the return value!!!
- `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. Make sure you do it as quickly as possible!

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);
```

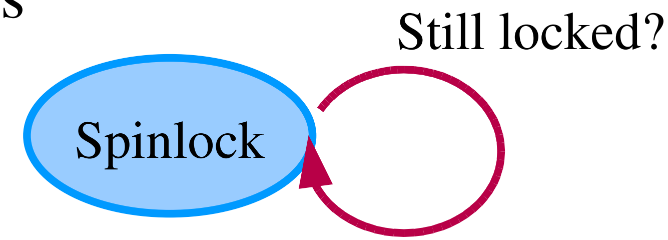
Well suited for rare writes, holding the semaphore briefly. Otherwise, readers get *starved*, waiting too long for the semaphore to be released.

When to use mutexes or semaphores

- Before and after accessing shared resources
- In situations when sleeping is allowed.
  Semaphores and mutexes must only be used in process context (managed by the scheduler), and not in interrupt context (managed by the CPU, sleeping not supported).

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 (1)

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).

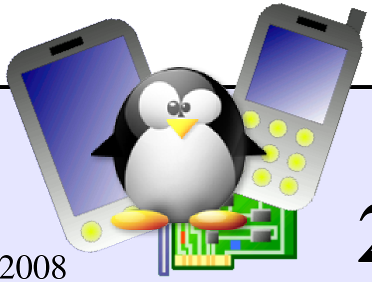
-
  ```
  void spin_lock_irqsave / spin_unlock_irqrestore
           (spinlock_t *lock, unsigned long flags);
  ```
  Disables / restores IRQs on the local CPU.

  Typically used when the lock can be accessed in both process and interrupt context, to prevent preemption by interrupts.

Using spinlocks (2)

- `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.

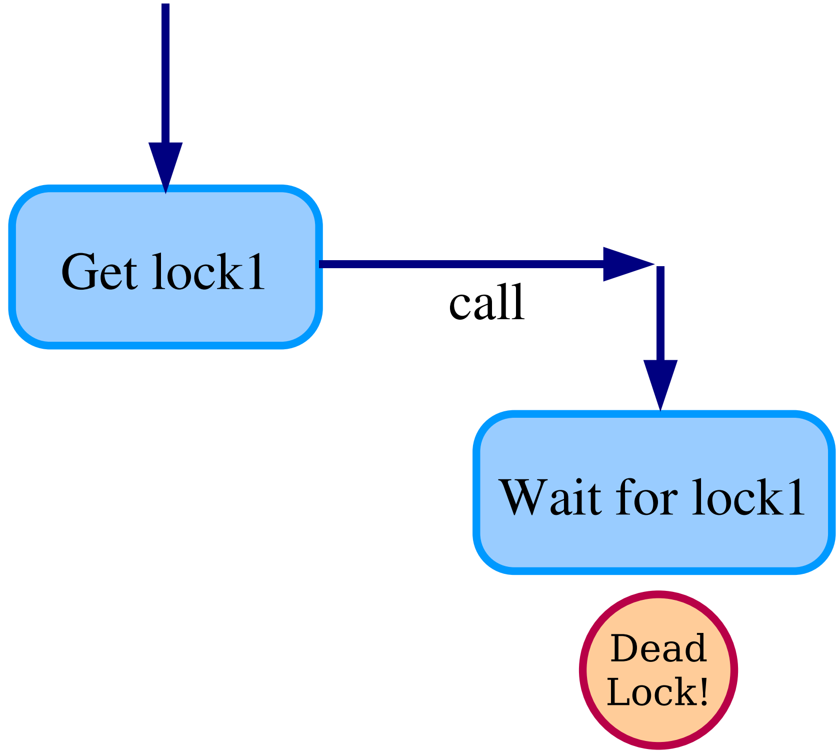
Note that reader / writer spinlocks also exist.



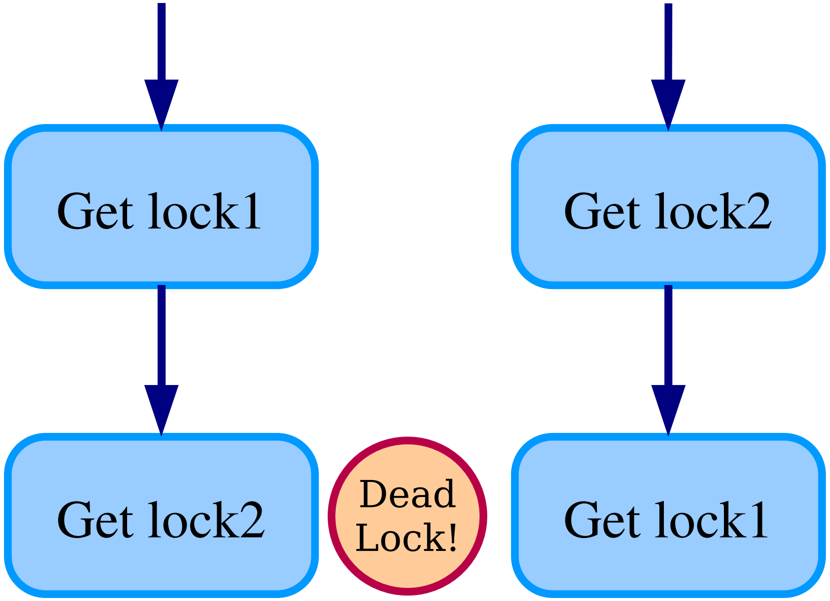
Deadlock situations

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

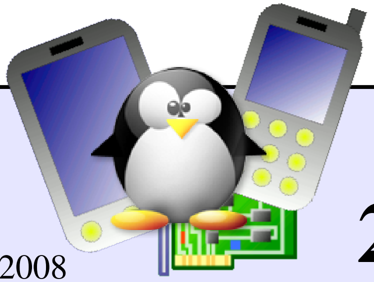
- Adds instrumentation to kernel locking code
- Detect violations of locking rules during system life, such as:
  - Locks acquired in different order (keeps track of locking sequences and compares them).
  - Spinlocks acquired in interrupt handlers and also in process context when interrupts are enabled.
- 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:

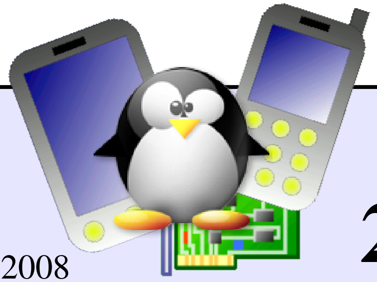
```
void atomic_inc (atomic_t *v);
void atomic_dec (atomic_t *v);
void atomic_add (int i, atomic_t *v);
void atomic_sub (int i, atomic_t *v);
```

- Simular functions testing the result:

```
int atomic_inc_and_test (...);
int atomic_dec_and_test (...);
int atomic_sub_and_test (...);
```

- Functions returning the new value:

```
int atomic_inc_and_return (...);
int atomic_dec_and_return (...);
int atomic_add_and_return (...);
int atomic_sub_and_return (...);
```



Atomic bit operations

- Supply very fast, atomic operations
- On most platforms, apply to an `unsigned long` type.

  Apply to a `void` type on a few others.
- Set, clear, toggle a given bit:

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

- Test bit value:

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

- Test and modify (return the previous value):

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

Driver development

mmap

mmap (1)

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

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

mmap (2)

Particularly useful when the file is a device file!

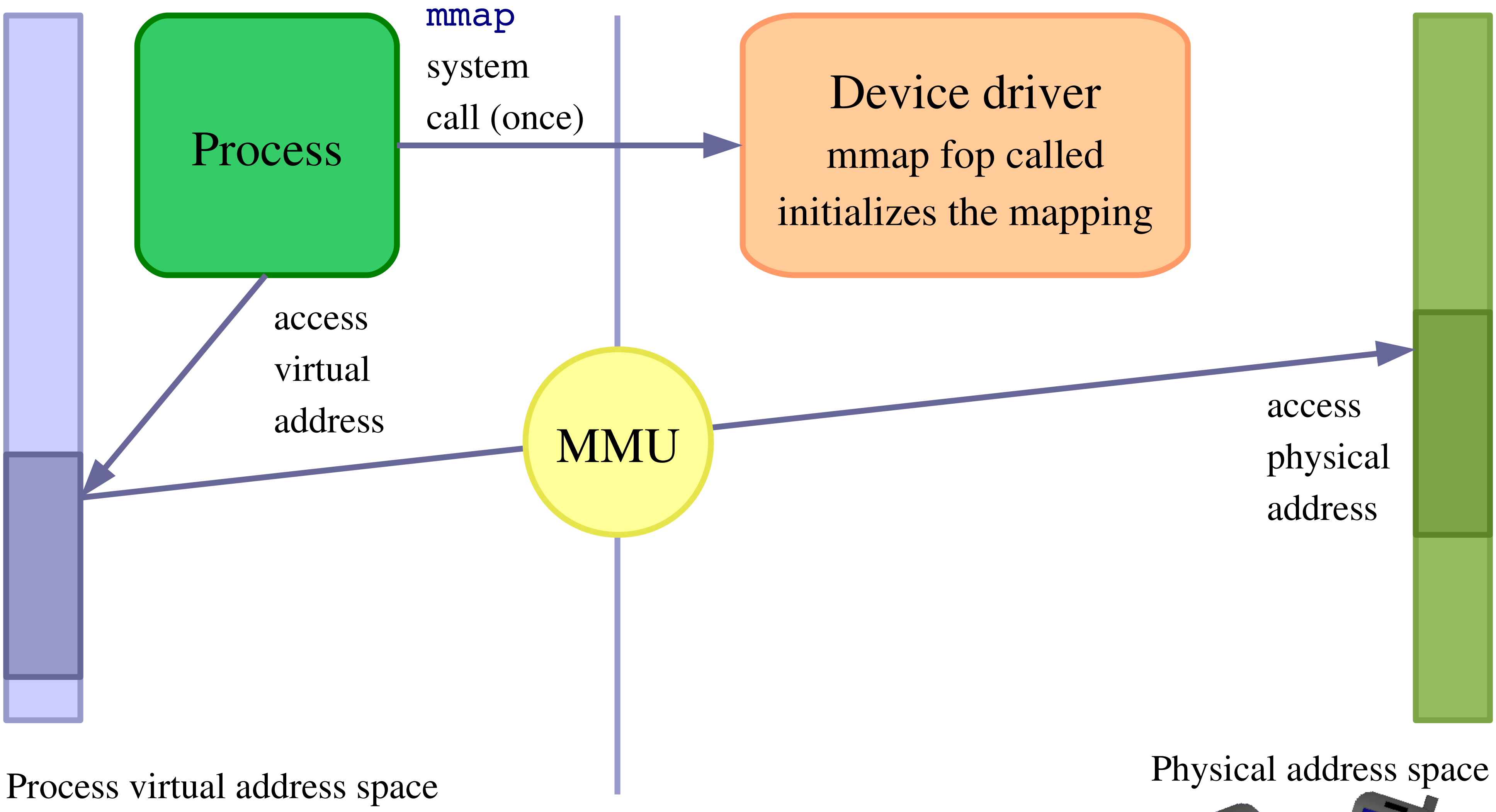
Allows to access device I/O memory and ports without having to go through (expensive) `read`, `write` or `ioctl` calls!

X server example (maps excerpt)

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

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

mmap overview



How to implement mmap - User space

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

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

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

How to implement mmap - Kernel space

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

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

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

remap_pfn_range()

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

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

Simple mmap implementation

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

   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)

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

DMA

DMA memory constraints

- Need to use contiguous memory in physical space.
- Can use any memory allocated by `kmalloc` (up to 128 KB) or `__get_free_pages` (up to 8MB).
- Can use block I/O and networking buffers, designed to support DMA.
- Can not use `vmalloc` memory (would have to setup DMA on each individual page).

Reserving memory for DMA

To make sure you've got enough RAM for big DMA transfers...

Example assuming you have 32 MB of RAM, and need 2 MB for DMA:

- Boot your kernel with `mem=30`

  The kernel will just use the first 30 MB of RAM.

- Driver code can now reclaim the 2 MB left:

```
dmabuf = ioremap (
            0x1e00000,        /* Start: 30 MB */
            0x200000          /* Size: 2 MB */
            );
```

Memory synchronization issues

Memory caching could interfere with DMA

- Before DMA to device:
  Need to make sure that all writes to DMA buffer are committed.
- After DMA from device:
  Before drivers read from DMA buffer, need to make sure that memory caches are flushed.
- Bidirectional DMA
  Need to flush caches before and after the DMA transfer.

Linux DMA API

The kernel DMA utilities can take care of:

- Either allocating a buffer in a cache coherent area,
- Or make sure caches are flushed when required,
- Managing the DMA mappings and IOMMU (if any).
- See `Documentation/DMA-API.txt`
  for details about the Linux DMA generic API.
- Most subsystems (such as PCI or USB) supply their own DMA API, derived from the generic one. May be sufficient for most needs.

Limited DMA address range?

- By default, the kernel assumes that your device can DMA to any 32 bit address. Not true for all devices!
- To tell the kernel that it can only handle 24 bit addresses:

```
if (dma_set_mask (dev,           /* device structure */
                  0x00ffffff     /* 24 bits */
                  ))
    use_dma = 1;                 /* Able to use DMA */
else
    use_dma = 0;        /* Will have to do without DMA */
```

Coherent or streaming DMA mappings

Coherent mappings

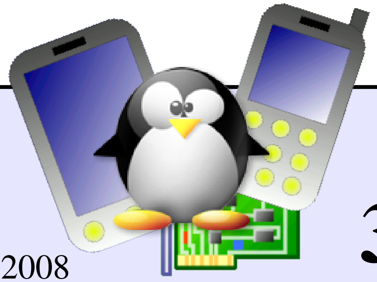
The kernel allocates a suitable buffer and sets the mapping for the driver.

- Can simultaneously be accessed by the CPU and device.
- So, has to be in a cache coherent memory area.
- Usually allocated for the whole time the module is loaded.
- Can be expensive to setup and use on some platforms.

Streaming mappings

The kernel just sets the mapping for a buffer provided by the driver.

- Use a buffer already allocated by the driver.
- Mapping set up for each transfer. Keeps DMA registers free on the hardware.
- Some optimizations also available.
- The recommended solution.



Allocating coherent mappings

The kernel takes care of both the buffer allocation and mapping:

```
include <asm/dma-mapping.h>

void *                          /* Output: buffer address */
  dma_alloc_coherent(
     struct device *dev,        /* device structure */
     size_t size,               /* Needed buffer size in bytes */
     dma_addr_t *handle,        /* Output: DMA bus address */
     gfp_t gfp                  /* Standard GFP flags */
  );

void dma_free_coherent(struct device *dev,
  size_t size, void *cpu_addr, dma_addr_t handle);
```

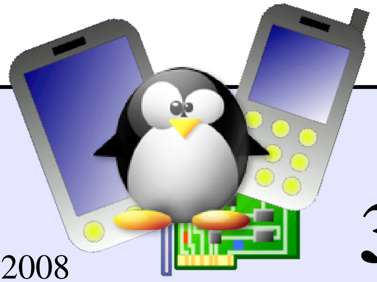
DMA pools (1)

- `dma_alloc_coherent` usually allocates buffers with `__get_free_pages` (minimum: 1 page).
- You can use DMA pools to allocate smaller coherent mappings:

```
<include linux/dmapool.h>
```

- Create a DMA pool:

```
struct dma_pool *
dma_pool_create (
    const char *name,         /* Name string */
    struct device *dev,       /* device structure */
    size_t size,              /* Size of pool buffers */
    size_t align,             /* Hardware alignment (bytes) */
    size_t allocation         /* Address boundaries not to be crossed */
);
```



DMA pools (2)

- Allocate from pool

```
void * dma_pool_alloc (
   struct dma_pool *pool,
   gfp_t mem_flags,
   dma_addr_t *handle
);
```

Note

DMA pools only used by USB core and 2 SCSI drivers

- Free buffer from pool

```
void dma_pool_free (
              struct dma_pool *pool,
              void *vaddr,
              dma_addr_t dma);
```

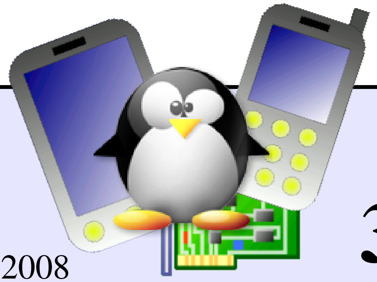
- Destroy the pool (free all buffers first!)

```
void dma_pool_destroy (struct dma_pool *pool);
```

Setting up streaming mappings

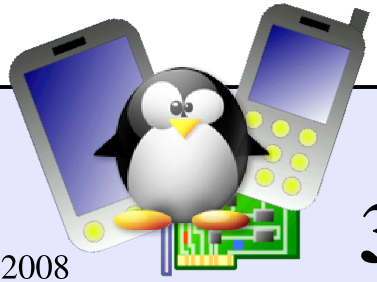
Works on buffers already allocated by the driver

```
<include linux/dmapool.h>

dma_addr_t dma_map_single(
  struct device *,                    /* device structure */
  void *,                             /* input: buffer to use */
  size_t,                             /* buffer size */
  enum dma_data_direction  /* Either DMA_BIDIRECTIONAL,
                    DMA_TO_DEVICE or DMA_FROM_DEVICE  */
  );

void dma_unmap_single(struct device *dev, dma_addr_t
  handle, size_t size, enum dma_data_direction dir);
```



DMA streaming mapping notes

- When the mapping is active: only the device should access the buffer (potential cache issues otherwise).
- The CPU can access the buffer only after unmapping!
  Use locking to prevent CPU access to the buffer.
- Another reason: if required, this API can create an intermediate *bounce buffer* (used if the given buffer is not usable for DMA).
- The Linux API also supports scatter / gather DMA streaming mappings.



DMA summary

Most drivers can use the specific API provided by their subsystem: USB, PCI, SCSI... Otherwise they can use the Linux generic API:

Coherent mappings

- DMA buffer allocated by the kernel
- Set up for the whole module life
- Can be expensive. Not recommended.
- Let both the CPU and device access the buffer at the same time.
- Main functions:
  dma_alloc_coherent
  dma_free_coherent

Streaming mappings

- DMA buffer allocated by the driver
- Set up for each transfer
- Cheaper. Saves DMA registers.
- Only the device can access the buffer when the mapping is active.
- Main functions:
  dma_map_single
  dma_unmap_single

Driver development

New Device Model

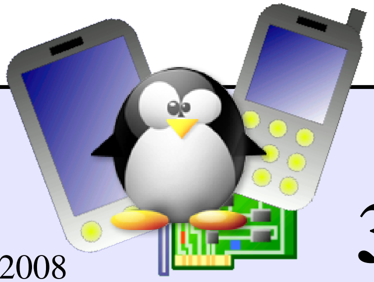
Device Model features (1)

- Originally created to make power management simpler
  Now goes much beyond.
- Used to represent the architecture and state of the system
- Has a representation in userspace: sysfs
  Now the preferred interface with userspace (instead of `/proc`)
- Easy to implement thanks to the device interface:
  `include/linux/device.h`

Device model features (2)

Allows to view the system for several points of view:

- From devices existing in the system: their power state, the bus they are attached to, and the driver responsible for them.
- From the system bus structure: which bus is connected to which bus (e.g. USB bus controller on the PCI bus), existing devices and devices potentially accepted (with their drivers)
- From the various kinds ("classes") of devices: `input`, `net`, `sound`... Existing devices for each class. Convenient to find all the input devices without actually knowing how they are physically connected.



sysfs

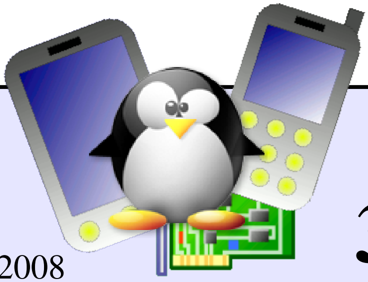
- Userspace representation of the Device Model.
- Configure it with
  `CONFIG_SYSFS=y` (Filesystems -> Pseudo filesystems)
- Mount it with

  ```
  sudo mount -t sysfs none /sys
  ```
- Spend time exploring `/sys` on your workstation!

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. Used by `udev` (see later).
- `systool` - A utility built upon `libsysfs` that lists devices by bus, class, and topology.



Device Model references

- Very useful and clear documentation in the kernel sources!
- `Documentation/driver-model/`
- `Documentation/filesystems/sysfs.txt`

Driver development

udev and hotplug

/dev issues and limitations

- On Red Hat 9, 18000 entries in `/dev`!
  All entries for all possible devices
  had to be created at system installation.

- Needed an authority to assign major numbers
  http://lanana.org/: Linux Assigned Names and Numbers Authority

- Not enough numbers in 2.4, limits extended in 2.6.

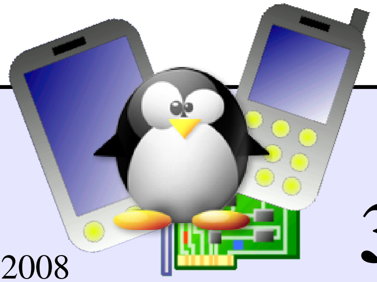
- Userspace neither knew what devices were present in the system,
  nor which real device corresponded to each `/dev` entry.

devfs solution and limitations

devfs: a first solution implemented in Linux 2.3.

- Only showed present devices
- But used different names as in `/dev`, causing issues in scripts.
- But no flexibility in device names, unlike with `/dev/`, e.g. the 1st IDE disk device had to be called either `/dev/hda` or `/dev/ide/hd/c0b0t0u0`.
- But didn't allow dynamic major and minor number allocation.
- But required to store the device naming policy in kernel memory. Kept forever in kernel RAM even when no longer needed.

devfs was completely removed in Linux 2.6.18.



The udev solution

Takes advantage of sysfs introduced by Linux 2.6.

- Created by Greg Kroah Hartman, a huge contributor. Other key contributors: Kay Sievers, Dan Stekloff.
- Entirely in user space.
- Automatically creates / removes device entries in `/dev/` according to inserted / removed devices.
- Major and minor device transmitted by the kernel.
- Requires no change to driver code.
- Fast: written in C Small size: `udevd` version 108: 61 KB in Ubuntu 7.04

hotplug history

udev was first implemented through the hotplug infrastructure:

- Introduced in Linux 2.4. Pioneered by USB.
- Whenever a device was inserted or removed, the kernel executed `/sbin/hotplug` to notify user space programs.
- For each subsystem (USB, PCI...), `/sbin/hotplug` then ran scripts (*agents*) taking care of identifying the hardware and inserting/removing the right driver modules.
- Linux 2.6: much easier device identification thanks to sysfs.
- udev was one of the agents run by `/sbin/hotplug`.

udev issues with hotplug

- sysfs timing issues.
- Out of order execution of hotplug processes.
- Out of memory issues when too many processes are run in a very short time.

Eventually, udev took over several parts of the hotplug infrastructure and completely replaced it.

Starting udev (1)

- At the very beginning of user-space startup, mount the `/dev/` directory as a tmpfs filesystem: `sudo mount -t tmpfs udev /dev`
- `/dev/` is populated with static devices available in `/lib/udev/devices/` :

Ubuntu 6.10 example:

```
crw------- 1 root root   5, 1 2007-01-31 04:18 console
lrwxrwxrwx 1 root root     11 2007-01-31 04:18 core -> /proc/kcore
lrwxrwxrwx 1 root root     13 2007-01-31 04:18 fd -> /proc/self/fd
crw-r----- 1 root kmem   1, 2 2007-01-31 04:18 kmem
brw------- 1 root root   7, 0 2007-01-31 04:18 loop0
lrwxrwxrwx 1 root root     13 2007-01-31 04:18 MAKEDEV -> /sbin/MAKEDEV
drwxr-xr-x 2 root root   4096 2007-01-31 04:18 net
crw------- 1 root root   1, 3 2007-01-31 04:18 null
crw------- 1 root root 108, 0 2007-01-31 04:18 ppp
drwxr-xr-x 2 root root   4096 2006-10-16 14:39 pts
drwxr-xr-x 2 root root   4096 2006-10-16 14:39 shm
lrwxrwxrwx 1 root root     24 2007-01-31 04:18 sndstat -> /proc/asound/oss/sndstat
lrwxrwxrwx 1 root root     15 2007-01-31 04:18 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root     15 2007-01-31 04:18 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root     15 2007-01-31 04:18 stdout -> /proc/self/fd/1
```

Starting udev (2)

- The `udevd` daemon is started.
  It listens to *uevents* from the driver core,
  which are sent whenever devices are inserted or removed.
- The `udevd` daemon reads and parses all the rules found in `/etc/udev/rules.d/` and keeps them in memory.
- Whenever rules are added, removed or modified, `udevd` receives an *inotify* event and updates its ruleset in memory.
- When an event is received, `udevd` starts a process to:
  - try to match the event against udev rules,
  - create / remove device files,
  - and run programs (to load / remove a driver, to notify user space...)

The ***inotify*** mechanism lets userspace programs subscribe to notifications of filesystem changes. Possibility to watch individual files or directories.

Event queue management

- `udevd` takes care of processing events in the right order.
  This is useful to process events after the ones then depend on (example: partition events need the parent block device event processing to be complete, to access its information in the udev database).

- `udevd` also limits the number of processes it starts. When the limit is exceeded, only events carrying the `TIMEOUT` key are immediately processed.

- The `/etc/.udev/queue/` directory represents currently running or queued events. It contains symbolic links to the corresponding sysfs devices. The directory is removed after removing the last link.

- Event processes which failed are represented by `/etc/.udev/failed/` . Symbolic links in this directory are removed when an event for the same device is successfully processed.

netlink sockets

Kernel netlink sockets are used to carry uevents. Advantages:

- They are asynchronous. Messages are queued. The receiver can choose to process messages at its best convenience.
- Other userspace - kernelspace communication means are synchronous: system calls, ioctls, `/proc/` and `/sys`.
- System calls have to be compiled statically into the kernel. They cannot be added by module-based device drivers.
- Multicasting is available. Several applications can be notified.

See http://www.linuxjournal.com/article/7356
for a very nice description of netlink sockets.

uevent message example

Example inserting a USB mouse

```
recv(4,                                      // socket id
     "add@/class/input/input9/mouse2\0       // message
     ACTION=add\0                            // action type
     DEVPATH=/class/input/input9/mouse2\0    // path in /sys
     SUBSYSTEM=input\0                       // subsystem (class)
     SEQNUM=1064\0                           // sequence number
     PHYSDEVPATH=/devices/pci0000:00/0000:00:1d.1/usb2/2-2/2-2:1.0\0
                                             // device path in /sys

     PHYSDEVBUS=usb\0                        // bus
     PHYSDEVDRIVER=usbhid\0                  // driver
     MAJOR=13\0                              // major number
     MINOR=34\0",                            // minor number
     2048,                                   // message buffer size
     0)                                      // flags
= 221                                        // actual message size
```

udev rules

When a udev rule matching event information is found, it can be used:

- To define the name and path of a device file.
- To define the owner, group and permissions of a device file.
- To execute a specified program.

Rule files are processed in lexical order.

udev naming capabilities

Device names can be defined

- from a label or serial number,
- from a bus device number,
- from a location on the bus topology,
- from a kernel name,
- from the output of a program.

See http://www.reactivated.net/writing_udev_rules.html for a very complete description. See also `man udev`.

udev naming rule examples

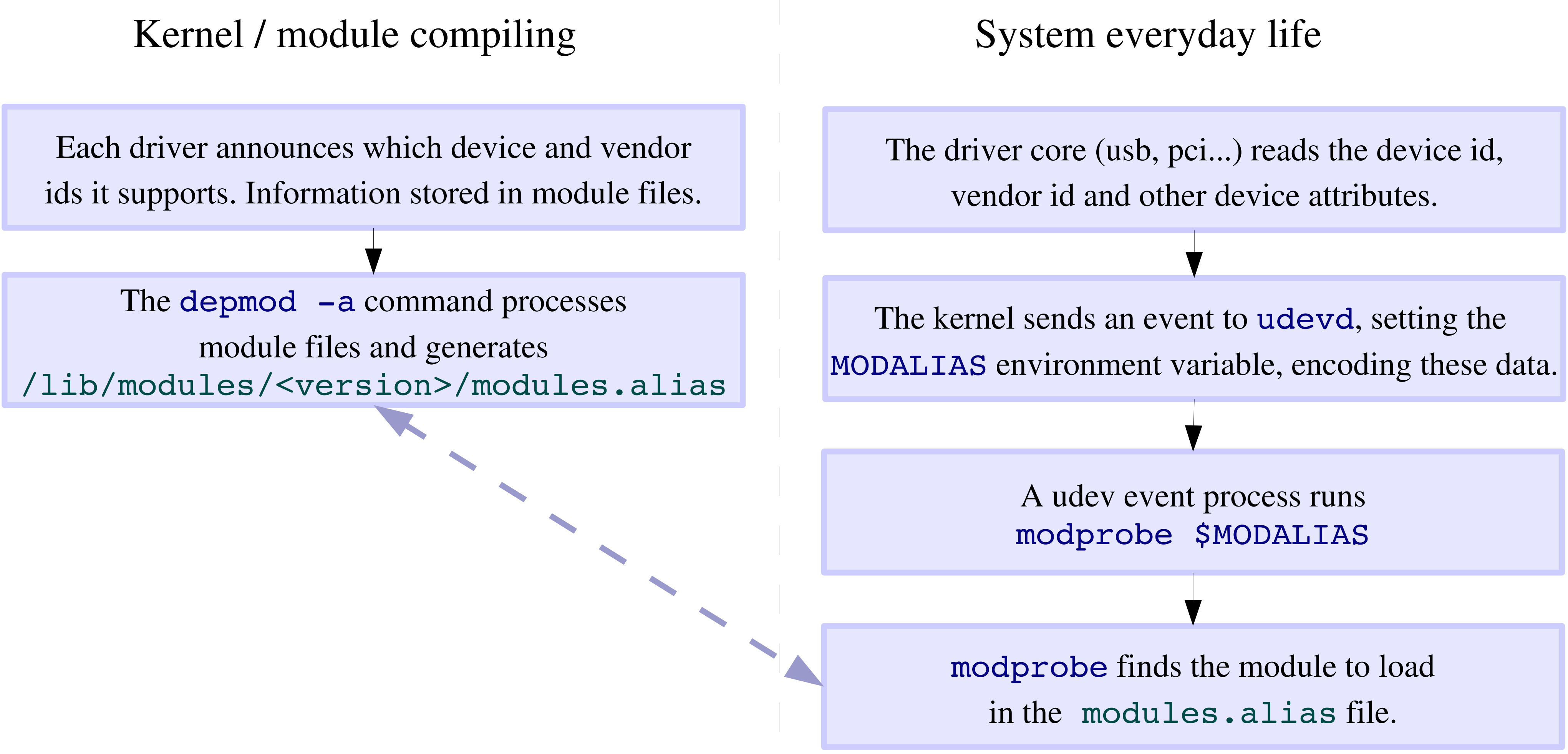
```
# Naming testing the output of a program
BUS=="scsi", PROGRAM="/sbin/scsi_id", RESULT=="OEM 0815", NAME="disk1"

# USB printer to be called lp_color
BUS=="usb", SYSFS{serial}=="W09090207101241330", NAME="lp_color"

# SCSI disk with a specific vendor and model number will be called boot
BUS=="scsi", SYSFS{vendor}=="IBM", SYSFS{model}=="ST336", NAME="boot%n"

# sound card with PCI bus id 00:0b.0 to be called dsp
BUS=="pci", ID=="00:0b.0", NAME="dsp"

# USB mouse at third port of the second hub to be called mouse1
BUS=="usb", PLACE=="2.3", NAME="mouse1"

# ttyUSB1 should always be called pda with two additional symlinks
KERNEL=="ttyUSB1", NAME="pda", SYMLINK="palmtop handheld"

# multiple USB webcams with symlinks to be called webcam0, webcam1, ...
BUS=="usb", SYSFS{model}=="XV3", NAME="video%n", SYMLINK="webcam%n"
```

udev permission rule examples

Excerpts from /etc/udev/rules.d/40-permissions.rules

```
# Block devices
SUBSYSTEM!="block", GOTO="block_end"
SYSFS{removable}!="1",                  GROUP="disk"
SYSFS{removable}=="1",                  GROUP="floppy"
BUS=="usb",                             GROUP="plugdev"
BUS=="ieee1394",                        GROUP="plugdev"
LABEL="block_end"

# Other devices, by name

KERNEL=="null",                         MODE="0666"
KERNEL=="zero",                         MODE="0666"
KERNEL=="full",                         MODE="0666"
```

Identifying device driver modules



Kernel / module compiling

Each driver announces which device and vendor ids it supports. Information stored in module files.

The `depmod -a` command processes module files and generates `/lib/modules/<version>/modules.alias`

System everyday life

The driver core (usb, pci...) reads the device id, vendor id and other device attributes.

The kernel sends an event to `udevd`, setting the `MODALIAS` environment variable, encoding these data.

A udev event process runs `modprobe $MODALIAS`

`modprobe` finds the module to load in the `modules.alias` file.

Module aliases

- `MODALIAS` environment variable example (USB mouse):
  MODALIAS=usb:**v**046D**p**C03E**d**2000**dc**00**dsc**00**dp**00ic03isc01ip02
- Matching line in `/lib/modules/<version>/modules.alias`:
  `alias usb:v*p*d*dc*dsc*dp*ic03isc01ip02* usbmouse`

udev modprobe rule examples

Even module loading is done with udev!

Excerpts from `/etc/udev/rules.d/90-modprobe.rules`

```
ACTION!="add", GOTO="modprobe_end"


SUBSYSTEM!="ide", GOTO="ide_end"
IMPORT{program}="ide_media --export $devpath"
ENV{IDE_MEDIA}=="cdrom",   RUN+="/sbin/modprobe -Qba ide-cd"
ENV{IDE_MEDIA}=="disk",    RUN+="/sbin/modprobe -Qba ide-disk"
ENV{IDE_MEDIA}=="floppy", RUN+="/sbin/modprobe -Qba ide-floppy"
ENV{IDE_MEDIA}=="tape",   RUN+="/sbin/modprobe -Qba ide-tape"
LABEL="ide_end"

SUBSYSTEM=="input", PROGRAM="/sbin/grepmap --udev", \
           RUN+="/sbin/modprobe -Qba $result"

# Load drivers that match kernel-supplied alias

ENV{MODALIAS}=="?*", RUN+="/sbin/modprobe -Q $env{MODALIAS}"
```

Coldplugging

- Issue: loosing all device events happening during kernel initialization, because udev is not ready yet.
- Solution: after starting `udevd`, have the kernel emit uevents for all devices present in `/sys`.
- This can be done by the `udevtrigger` utility.
- Strong benefit: completely transparent for userspace. Legacy and removable devices handled and named in exactly the same way.

Debugging events - udevmonitor (1)

udevmonitor visualizes the driver core events and the udev event processes.

Example event sequence connecting a USB mouse:

```
UEVENT[1170452995.094476] add@/devices/pci0000:00/0000:00:1d.7/usb4/4-3/4-3.2
UEVENT[1170452995.094569] add@/devices/pci0000:00/0000:00:1d.7/usb4/4-3/4-3.2/4-3.2:1.0
UEVENT[1170452995.098337] add@/class/input/input28
UEVENT[1170452995.098618] add@/class/input/input28/mouse2
UEVENT[1170452995.098868] add@/class/input/input28/event4
UEVENT[1170452995.099110] add@/class/input/input28/ts2
UEVENT[1170452995.099353] add@/class/usb_device/usbdev4.30
UDEV  [1170452995.165185] add@/devices/pci0000:00/0000:00:1d.7/usb4/4-3/4-3.2
UDEV  [1170452995.274128] add@/devices/pci0000:00/0000:00:1d.7/usb4/4-3/4-3.2/4-3.2:1.0
UDEV  [1170452995.375726] add@/class/usb_device/usbdev4.30
UDEV  [1170452995.415638] add@/class/input/input28
UDEV  [1170452995.504164] add@/class/input/input28/mouse2
UDEV  [1170452995.525087] add@/class/input/input28/event4
UDEV  [1170452995.568758] add@/class/input/input28/ts2
```

It gives time information measured in microseconds.

You can measure time elapsed between the uevent (UEVENT line), and the completion of the corresponding udev process (matching UDEV line).

Debugging events - udevmonitor (2)

`udevmonitor --env` shows the complete event environment for each line.

```
UDEV  [1170453642.595297] add@/devices/pci0000:00/0000:00:1d.7/usb4/4-3/4-3.2/4-3.2:1.0
UDEV_LOG=3
ACTION=add
DEVPATH=/devices/pci0000:00/0000:00:1d.7/usb4/4-3/4-3.2/4-3.2:1.0
SUBSYSTEM=usb
SEQNUM=3417
PHYSDEVBUS=usb
DEVICE=/proc/bus/usb/004/031
PRODUCT=46d/c03d/2000
TYPE=0/0/0
INTERFACE=3/1/2
MODALIAS=usb:v046DpC03Dd2000dc00dsc00dp00ic03isc01ip02
UDEVD_EVENT=1
```

Misc udev utilities

- `udevinfo`

  Lets users query the udev database.

- `udevtest <sysfs_device_path>`

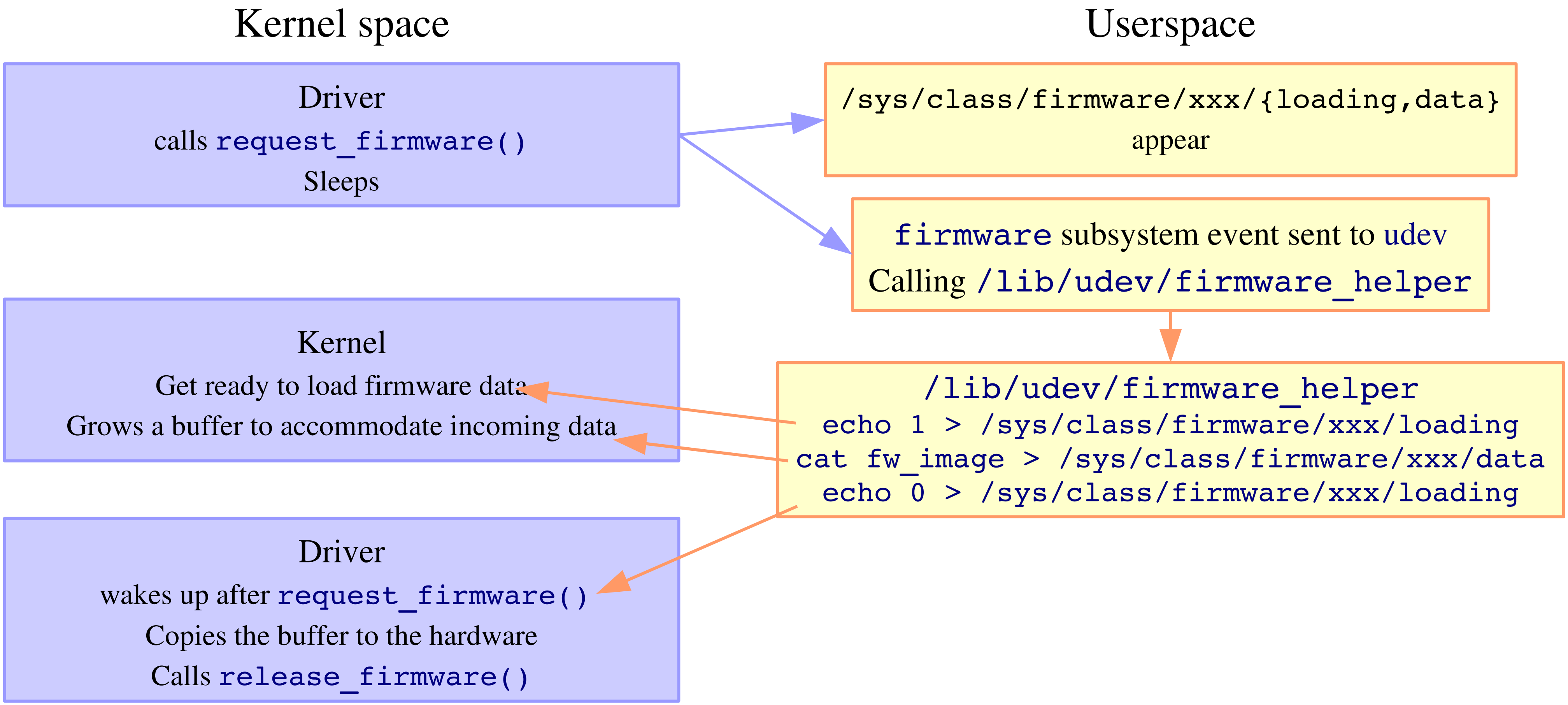
  Simulates a `udev` run to test the configured rules.

Firmware hotplugging

Also implemented with udev!

- Firmware data are kept outside device drivers
  - May not be legal or free enough to distribute
  - Firmware in kernel code would occupy memory permanently, even if just used once.
- Kernel configuration: needs to be set in `CONFIG_FW_LOADER` (Device Drivers -> Generic Driver Options -> hotplug firmware loading support)

Firmware hotplugging implementation



Kernel space

Userspace

Driver
calls `request_firmware()`
Sleeps

`/sys/class/firmware/xxx/{loading,data}`
appear

`firmware` subsystem event sent to `udev`
Calling `/lib/udev/firmware_helper`

```
/lib/udev/firmware_helper
echo 1 > /sys/class/firmware/xxx/loading
cat fw_image > /sys/class/firmware/xxx/data
echo 0 > /sys/class/firmware/xxx/loading
```

Kernel
Get ready to load firmware data
Grows a buffer to accommodate incoming data

Driver
wakes up after `request_firmware()`
Copies the buffer to the hardware
Calls `release_firmware()`

See `Documentation/firmware_class/` for a nice overview

udev files

- `/etc/udev/udev.conf`

  udev configuration file.

  Mainly used to configure syslog reporting priorities.

  Example setting: `udev_log="err"`

- `/etc/udev/rules.d/*.rules`

  udev event matching rules.

- `/lib/udev/devices/*`

  static `/dev` content (such as `/dev/console`, `/dev/null`...).

- `/lib/udev/*`

  helper programs called from udev rules.

- `/dev/*`

  Created device files.

Kernel configuration for udev

Created for 2.6.19

Caution: no documentation found, and not tested yet on a minimalistic system.

Some settings may still be missing.

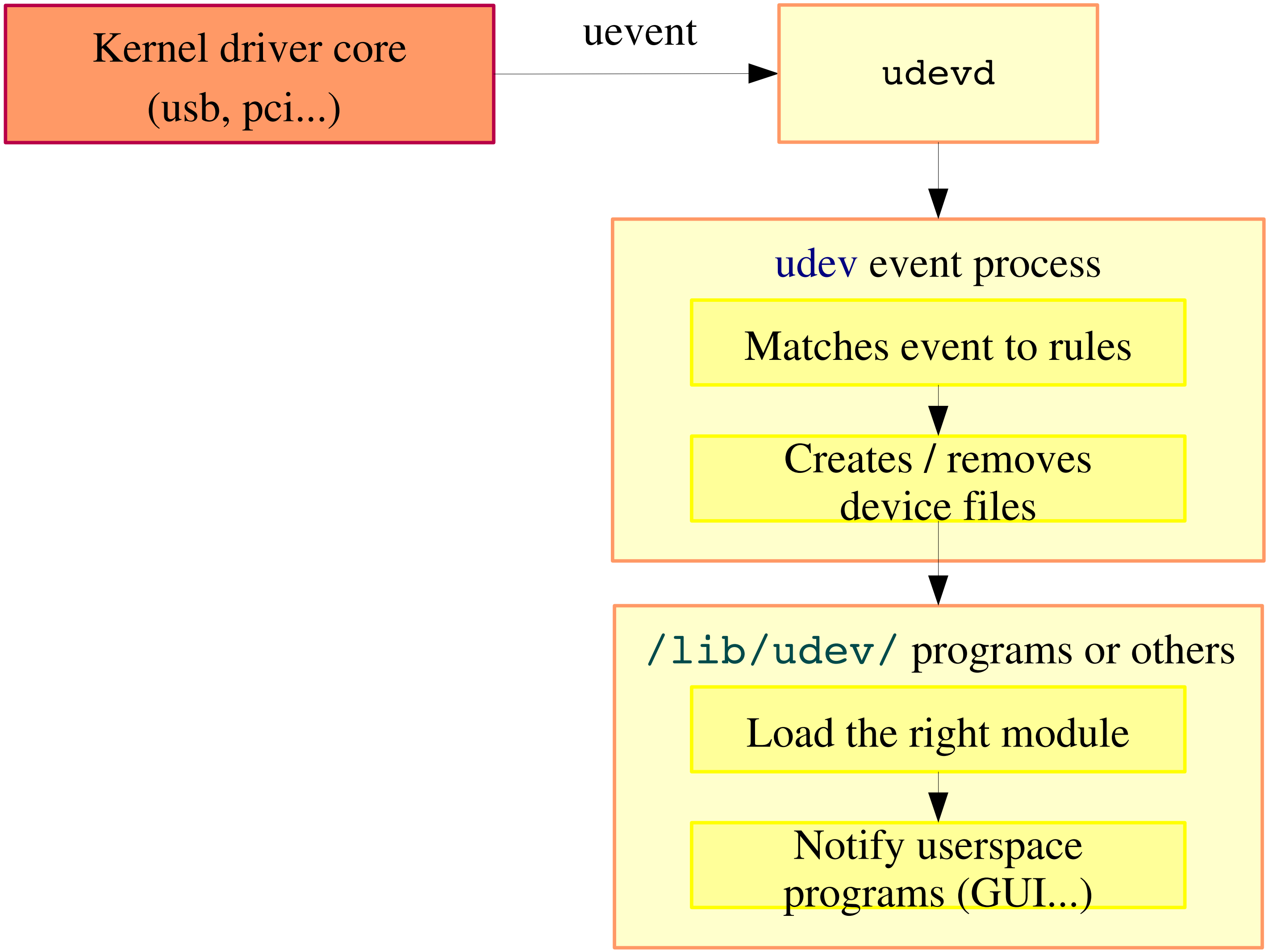
Subsystems and device drivers (USB, PCI, PCMCIA...) should be added too!

```
# General setup
CONFIG_HOTPLUG=y
# Networking, networking options
CONFIG_NET=y
CONFIG_UNIX=y
CONFIG_NETFILTER_NETLINK=y
CONFIG_NETFILTER_NETLINK_QUEUE=y
# Pseudo filesystems
CONFIG_PROC_FS=y
CONFIG_SYSFS=y
CONFIG_TMPFS=y
CONFIG_RAMFS=y
```

Unix domain sockets

Needed to manage /dev

udev summary - typical operation



Kernel driver core (usb, pci...)

uevent

udevd

udev event process

Matches event to rules

Creates / removes device files

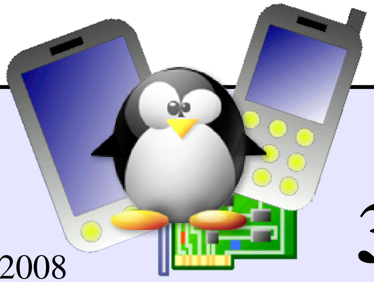
/lib/udev/ programs or others

Load the right module

Notify userspace programs (GUI...)

udev resources

- Home page
  http://kernel.org/pub/linux/utils/kernel/hotplug/udev.html
- Sources
  http://kernel.org/pub/linux/utils/kernel/hotplug/
- Recent state of udev, by Kay Sievers (very good article):
  http://vrfy.org/log/recent-state-of-udev.html
- The udev manual page:
  `man udev`



Embedded Linux driver development

Advice and resources

Getting help and contributions

Solving issues

- If you face an issue, and it doesn't look specific to your work but rather to the tools you are using, it is very likely that someone else already faced it.
- Search the Internet for similar error reports.
- You have great chances of finding a solution or workaround, or at least an explanation for your issue.
- Otherwise, reporting the issue is up to you!

Getting help

- If you have a support contract, ask your vendor.
- Otherwise, don't hesitate to share your questions and issues
  - Either contact the Linux mailing list for your architecture (like linux-arm-kernel or linuxsh-dev...).
  - Or contact the mailing list for the subsystem you're dealing with (linux-usb-devel, linux-mtd...). Don't ask the maintainer directly!
  - Most mailing lists come with a FAQ page. Make sure you read it before contacting the mailing list.
  - Useful IRC resources are available too
    (for example on http://kernelnewbies.org).
  - Refrain from contacting the Linux Kernel mailing list, unless you're an experienced developer and need advice.

Getting contributions

Applies if your project can interest other people: developing a driver or filesystem, porting Linux on a new processor, board or device available on the market...

External contributors can help you a lot by

- Testing
- Writing documentation
- Making suggestions
- Even writing code

Encouraging contributions

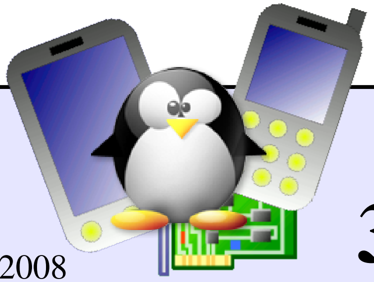
- Open your development process: mailing list, Wiki, public CVS read access
- Let everyone contribute according to their skills and interests.
- Release early, release often
- Take feedback and suggestions into account
- Recognize contributions
- Make sure status and documentation are up to date
- Publicize your work and progress to broader audiences

Advice and resources

Bug report and patch submission

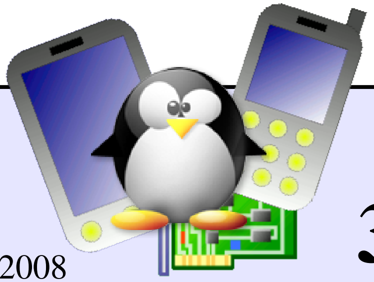
Reporting Linux bugs

- First make sure you're using the latest version
- Make sure you investigate the issue as much as you can: see `Documentation/BUG-HUNTING`
- Make sure the bug has not been reported yet. A bug tracking system (http://bugzilla.kernel.org/) exists but very few kernel developers use it. Best to use web search engines (accessing public mailing list archives)
- If the subsystem you report a bug on has a mailing list, use it. Otherwise, contact the official maintainer (see the `MAINTAINERS` file). Always give as many useful details as possible.



How to submit patches or drivers

- Don't merge patches addressing different issues
- You should identify and contact the official maintainer for the files to patch.
- See `Documentation/SubmittingPatches` for details. For trivial patches, you can copy the Trivial Patch Monkey.
- See also http://kernelnewbies.org/UpstreamMerge for very helpful advice to have your code merged upstream (by Rik van Riel).
- Special subsystems:
  - ARM platform: it's best to submit your ARM patches to Russell King's patch system: http://www.arm.linux.org.uk/developer/patches/



How to become a kernel developer?

Greg Kroah-Hartman gathered useful references and advice for people interested in contributing to kernel development:

`Documentation/HOWTO`

Do not miss this very useful document!

Advice and resources

References

Specific training materials

Free Electrons is working on dedicated training materials for specific device / driver types:

- Linux USB drivers
  http://free-electrons.com/articles/linux-usb

More will be available in the next months: block, network, input, audio, graphics...

Don't hesitate to ask us to create the ones you need for a training session!

Information sites (1)

Linux Weekly News

http://lwn.net/

- The weekly digest off all Linux and free software information sources
- In depth technical discussions about the kernel
- Subscribe to finance the editors ($5 / month)
- Articles available for non subscribers after 1 week.

Information sites (2)

KernelTrap

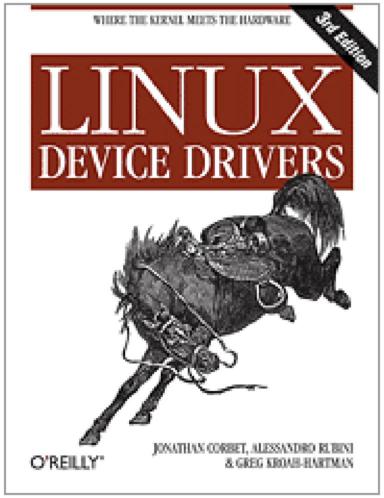
http://kerneltrap.org/

- Forum website for kernel developers
- News, articles, whitepapers, discussions, polls, interviews
- Perfect if a digest is not enough!

Useful reading (1)

Linux Device Drivers, 3rd edition, Feb 2005 ★ ★ ★

- By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, O'Reilly
  http://www.oreilly.com/catalog/linuxdrive3/

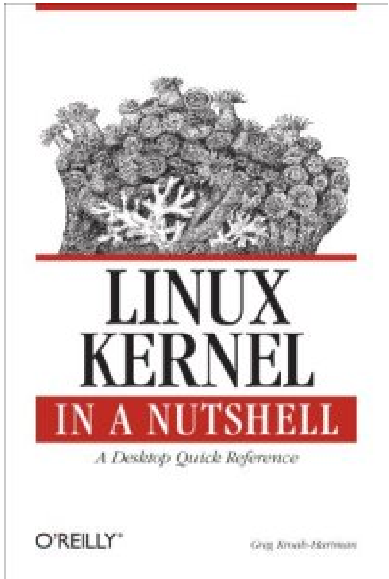

WHERE THE KERNEL MEETS THE HARDWARE
3rd Edition
LINUX
DEVICE DRIVERS
O'REILLY
JONATHAN CORBET, ALESSANDRO RUBINI
& GREG KROAH-HARTMAN

- Freely available on-line!
  Great companion to the printed book
  for easy electronic searches!
  http://lwn.net/Kernel/LDD3/ (1 PDF file per chapter)
  http://free-electrons.com/community/kernel/ldd3/ (single PDF file)

A must-have book for Linux device driver writers!

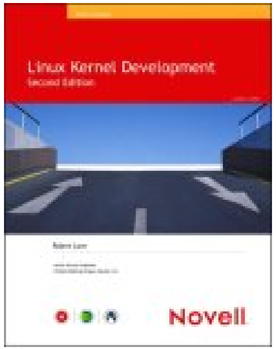
Useful reading (2)

Linux Kernel in a Nutshell, Dec 2006

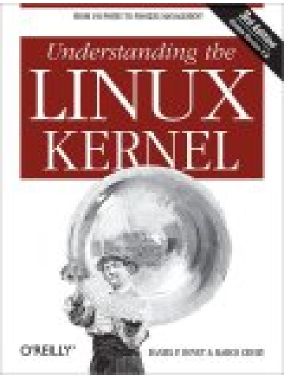


- By Greg Kroah-Hartman, O'Reilly
  http://www.kroah.com/lkn/
- A good reference book and guide on configuring, compiling and managing the Linux kernel sources.
- Freely available on-line!
  Great companion to the printed book for easy electronic searches!
  Available as single PDF file on
  http://free-electrons.com/community/kernel/lkn/

Useful reading (3)



- Linux Kernel Development, 2nd Edition, Jan 2005 ★★★
  Robert Love, Novell Press
  http://free-electrons.com/redirect/lkd2-book.html
  A very synthetic and pleasant way to learn about kernel subsystems (beyond the needs of device driver writers)



- Understanding the Linux Kernel, 3rd edition, Nov 2005 ★★
  Daniel P. Bovet, Marco Cesati, O'Reilly
  http://oreilly.com/catalog/understandlk/
  An extensive review of Linux kernel internals, covering Linux 2.6 at last.
  Unfortunately, only covers the PC architecture.

Useful on-line resources

- Linux kernel mailing list FAQ
  http://www.tux.org/lkml/
  Complete Linux kernel FAQ
  Read this before asking a question to the mailing list

- Kernel Newbies
  http://kernelnewbies.org/
  Glossary, articles, presentations, HOWTOs,
  recommended reading, useful tools for people
  getting familiar with Linux kernel or driver development.

- Kernel glossary:
  http://kernelnewbies.org/KernelGlossary

Embedded Linux Wiki

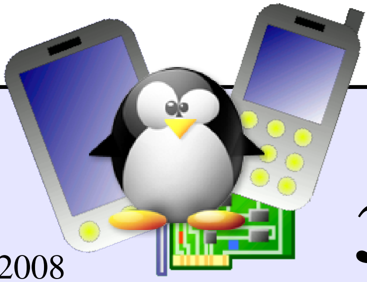
The embedded Linux Wiki contains loads of useful resources for embedded systems developers:

- Many HOWTO documents of all kinds, covering topics like system size, boot time, multimedia, power management, toolchains...
- Kernel patches not available in mainstream yet (e.g. Linux Tiny)
- Community resource: hacker interviews, book reviews, event coverage...
- Is open to everyone. Contributions are welcome!

http://elinux.org

ARM resources

- ARM Linux project: http://www.arm.linux.org.uk/
  - Developer documentation: http://www.arm.linux.org.uk/developer/
  - arm-linux-kernel mailing list: http://lists.arm.linux.org.uk/mailman/listinfo/linux-arm-kernel
  - FAQ: http://www.arm.linux.org.uk/armlinux/mlfaq.php
  - How to post kernel fixes: http://www.arm.uk.linux.org/developer/patches/
- ARMLinux @ Simtec: http://armlinux.simtec.co.uk/
  A few useful resources: FAQ, documentation and Who's who!
- ARM Limited: http://www.linux-arm.com/
  Wiki with links to useful developer resources



International conferences (1)

Useful conferences featuring Linux kernel presentations

- Ottawa Linux Symposium (July): http://linuxsymposium.org/
  Lots of kernel topics by major kernel hackers.
  Freely available proceedings.

- Fosdem: http://fosdem.org (Brussels, February)
  For developers. Kernel presentations from well-known kernel hackers.

- Embedded Linux Conference: http://embeddedlinuxconference.com/
  Organized by the CE Linux Forum: California (San Jose, April), in Europe (November). Frequent technical sessions in Japan. Very interesting kernel topics for embedded systems developers. Presentation slides freely available.

International conferences (2)

- linux.conf.au: http://conf.linux.org.au/ (Australia / New Zealand) Features a few presentations by key kernel hackers.

Don't miss our free conference videos on http://free-electrons.com/community/videos/conferences/!

Advice and resources

Last advice

Use the Source, Luke!

Many resources and tricks on the Internet find you will, but solutions to all technical issues only in the Source lie.

Thanks to LucasArts

Annexes

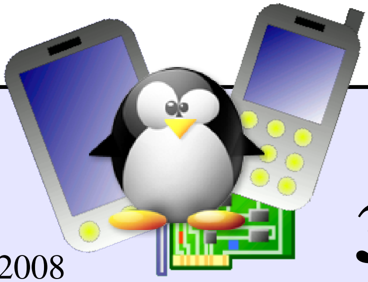
Quiz answers

Quiz answers

- Interrupt handling

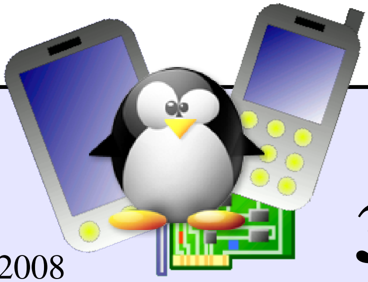
  Q: Why did the kernel segfault at module unload (forgetting to unregister a handler in a shared interrupt line)?

  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!



Annexes

Kernel sources

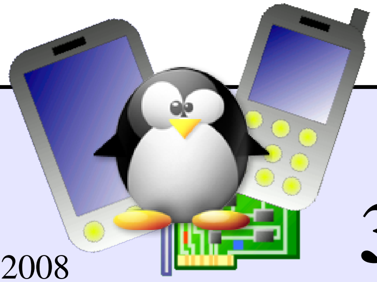


Checking the integrity of sources

Kernel source integrity can be checked through OpenPGP digital signatures.
Full details on http://www.kernel.org/signature.html

- Details about GnuPG: http://www.gnupg.org/gph/en/manual.html
- Import the public GnuPG key of kernel developers:

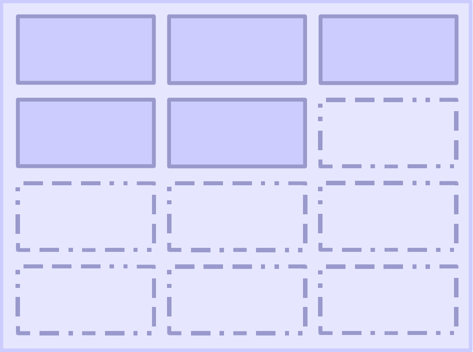
  -
    ```
    gpg --keyserver pgp.mit.edu --recv-keys 0x517D0F0E
    ```
  - If blocked by your firewall, look for `0x517D0F0E` on http://pgp.mit.edu/, copy and paste the key to a `linuxkey.txt` file:
    ```
    gpg --import linuxkey.txt
    ```
- Download the signature file corresponding to your kernel version (at the same location), and run the signature check:
  ```
  gpg --verify linux-2.6.11.12.tar.bz2.sign
  ```



Annexes

Slab caches and memory pools

Slab caches



Also called *lookaside caches*

- *Slab caches*: Objects that can hold any number of memory areas of the same size.
- Optimum use of available RAM and reduced fragmentation.
- Mainly used in Linux core subsystems: filesystems (open files, inode and file caches...), networking... Live stats on `/proc/slabinfo`.
- May be useful in device drivers too, though not used so often. Linux 2.6: used by USB and SCSI drivers.

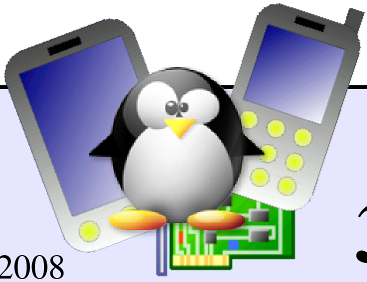
Slab cache API (1)

- `#include <linux/slab.h>`
- Creating a cache:

```
cache = kmem_cache_create (
    name,           /* Name for /proc/slabinfo */
    size,           /* Cache object size */
    align,          /* Cache alignment */
    flags,          /* Options: DMA, debugging, tracing... */
    constructor,    /* Optional, called after each allocation */
    destructor);    /* Optional, called before each release */
```

- Example: `drivers/usb/host/uhci-hcd.c`

```
uhci_up_cachep = kmem_cache_create(
    "uhci_urb_priv", sizeof(struct urb_priv),
    0, 0, NULL, NULL);
```



Slab cache API (2)

Since Linux 2.6.22, a macro can simplify cache creation in most cases:

-
  ```
  #define KMEM_CACHE(__struct, __flags)\
     kmem_cache_create(#__struct,\
          sizeof(struct __struct),\
          __alignof__(struct __struct),\
          (__flags), NULL, NULL)
  ```
- Example: `kernel/pid.c`

  ```
  pid_cachep = KMEM_CACHE(pid, SLAB_PANIC);
  ```

Slab cache API (3)

- Allocating from the cache:

  `object = kmem_cache_alloc (cache, flags);`

  or `object = kmem_cache_zalloc (cache, flags);`

- Freeing an object:

  `kmem_cache_free (cache, object);`

- Destroying the whole cache:

  `kmem_cache_destroy (cache);`

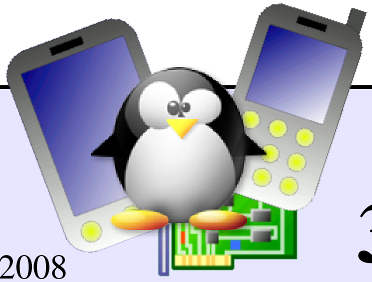
More details and an example in the Linux Device Drivers book:

http://lwn.net/images/pdf/LDD3/ch08.pdf

Memory pools

Useful for memory allocations that cannot fail

- Kind of lookaside cache trying to keep a minimum number of pre-allocated objects ahead of time.
- Use with care: otherwise can result in a lot of unused memory that cannot be reclaimed! Use other solutions whenever possible.



Memory pool API (1)

- `#include <linux/mempool.h>`
- Mempool creation:

```
mempool = mempool_create (
   min_nr,
   alloc_function,
   free_function,
   pool_data);
```

Memory pool API (2)

- Allocating objects:

```
object = mempool_alloc (pool, flags);
```

- Freeing objects:

```
mempool_free (object, pool);
```

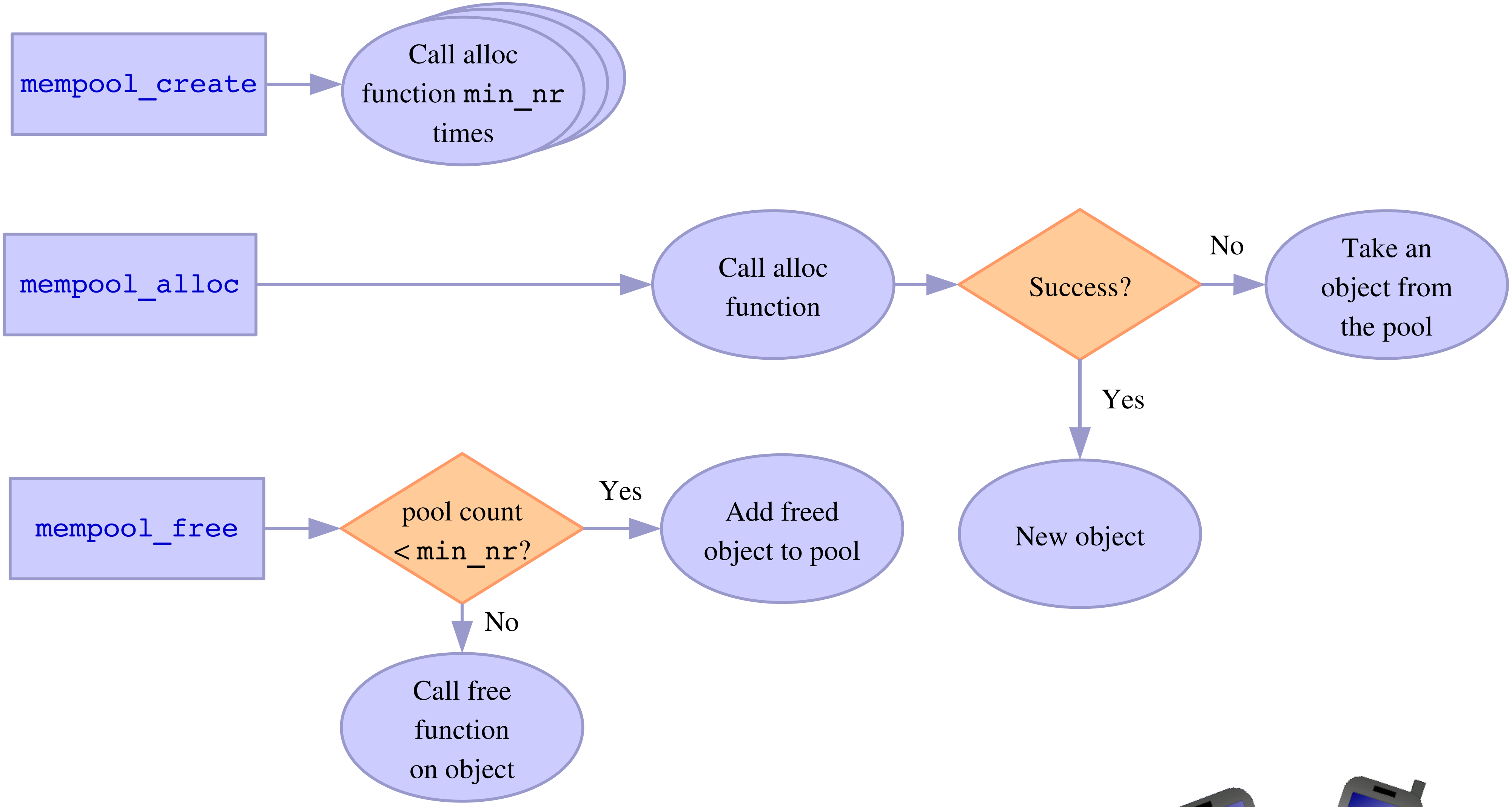
- Resizing the pool:

```
status = mempool_resize (
            pool, new_min_nr, flags);
```

- Destroying the pool (caution: free all objects first!):

```
mempool_destroy (pool);
```

Memory pool implementation



Memory pools using slab caches

- Idea: use slab cache functions to allocate and free objects.
- The `mempool_alloc_slab` and `mempool_free_slab` functions supply a link with slab cache routines.
- So, you will find many code examples looking like:

```
cache = kmem_cache_create (...);
pool = mempool_create (
                     min_nr,
                     mempool_alloc_slab,
                     mempool_free_slab,
                     cache);
```

There's a shorthand pool creation function for this case:

```
pool = mempool_create_slab_pool(min_nr, cache);
```

Annexes

U-boot details

Postprocessing kernel image for U-boot

The U-boot bootloader needs extra information to be added to the kernel and initrd image files.

- `mkimage` postprocessing utility provided in U-boot sources
- Kernel image postprocessing:
  `make uImage`

Postprocessing initrd image for U-boot

```
mkimage
  -n initrd \                 Name
  -A arm \                    Architecture
  -O linux \                  Operating System
  -T ramdisk \                Type
  -C gzip \                   Compression
  -d rd-ext2.gz \             Input file
  uInitrd                     Output file
```

Compiling Das U-boot

- Get the U-boot sources from http://www.denx.de/wiki/UBoot
- In the U-boot source directory:
  Find the name of the config file for your board in `include/configs`
  (for example: `omap1710h3.h`)
- Configure U-boot:
  `make omap1710h3_config` (`.h` replaced by `_config`)
- If needed, change the cross-compiler prefix in `Makefile`:

  ```
  ifeq ($(ARCH),arm)
  CROSS_COMPILE = arm-linux-
  endif
  ```

- Compile:

  ```
  make
  ```

Compiling U-boot mkimage

If you just need `mkimage` and U-boot is already installed on your board:

- `mkimage` is completely architecture and board independent.
- Configure U-boot sources for any board on any architecture (see previous slide).
- Compile:
  `make` (or `make -k` if you have minor failures)
- Install `mkimage`:

  ```
  cp tools/mkimage /usr/local/bin/
  ```

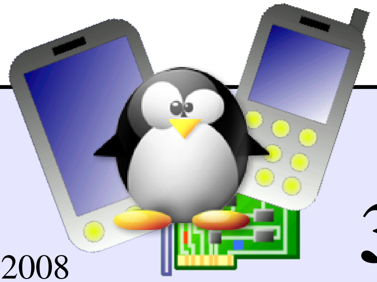
Configuring tftp (1)

Often in development: downloading a kernel image from the network.

Instructions for xinetd based systems (Fedora Core, Red Hat...)

- Install the `tftp-server` package if needed
- Remove `disable = yes` in `/etc/xinetd.d/tftp`
- Copy your image files to the `/tftpboot/` directory (or to the location specified in `/etc/xinetd.d/tftp`)
- You may have to disable SELinux in `/etc/selinux/config`
- Restart xinetd:

```
/etc/init.d/xinetd restart
```



Configuring tftp (2)

On GNU/Linux systems based on Debian: Ubuntu, Knoppix
(already set up in KernelKit)

- Install the `tftpd-hpa` package if needed
- Set `RUN_DAEMON="yes"`
  in `/etc/default/tftpd-hpa`
- Copy your images to `/var/lib/tftpboot`
- `/etc/hosts.allow`:
  `Replace ALL : ALL@ALL : DENY by ALL : ALL@ALL : ALLOW`
- `/etc/hosts.deny`:
  Comment out `ALL: PARANOID`
- Restart the server:
  `/etc/init.d/tftpd-hpa restart`

U-boot prompt

- Connect the target to the host through a serial console
- Power-up the board.

  On the serial console, you will see something like:

```
U-Boot 1.1.2 (Aug  3 2004 - 17:31:20)
RAM Configuration:
Bank #0: 00000000  8 MB
Flash:  2 MB
In:    serial
Out:   serial
Err:   serial
u-boot #
```

Board information

```
u-boot # bdinfo
DRAM bank = 0x00000000
-> start = 0x00000000
-> size = 0x00800000
ethaddr = 00:40:95:36:35:33
ip_addr = 10.0.0.11
baudrate = 19200 bps
```

Environment variables (1)

```
u-boot # printenv
baudrate=19200
ethaddr=00:40:95:36:35:33
netmask=255.255.255.0
ipaddr=10.0.0.11
serverip=10.0.0.1
stdin=serial
stdout=serial
stderr=serial

u-boot # setenv serverip 10.0.0.2

u-boot # printenv serverip
serverip=10.0.0.2
```

Network settings
For TFTP
and NFS

Environment variables (2)

- Environment variable changes can be stored to flash using the `saveenv` command.
- You can even create small shell scripts stored in environment variables:

```
setenv myscript 'tftp 0x21400000 uImage ;
bootm 0x21400000'
```

- You can then execute the script:

```
run myscript
```

- More elaborate scripting is available with script files, to be processed with `mkimage`.

Network commands

```
u-boot # tftp 8000 u-boot.bin
From server 10.0.0.1; our IP address is
10.0.0.11
Filename 'u-boot.bin'.
Load address: 0x8000
Loading: ###################
done
Bytes transferred = 95032 (17338 hex)
```

The address and size of the downloaded file are stored in the `fileaddr` and `filesize` environment variables.

Flash commands (1)

```
u-boot # flinfo
Bank # 1: AMD Am29LV160DB 16KB,2x8KB,32KB,31x64KB
Size: 2048 KB in 35 Sectors
Sector Start Addresses:
S00 @ 0x01000000 ! S01 @ 0x01004000 !
S02 @ 0x01006000 ! S03 @ 0x01008000 !
S04 @ 0x01010000 ! S05 @ 0x01020000 !
S06 @ 0x01030000 S07 @ 0x01040000
...
S32 @ 0x011D0000 S33 @ 0x011E0000
S34 @ 0x011F0000
```

Protected sectors

Flash commands (2)

```
u-boot # protect off 1:0-4
Un-Protect Flash Sectors 0-4 in Bank # 1

u-boot # erase 1:0-4
Erase Flash Sectors 0-4 in Bank # 1
Erasing Sector 0 @ 0x01000000 ... done
Erasing Sector 1 @ 0x01004000 ... done
Erasing Sector 2 @ 0x01006000 ... done
Erasing Sector 3 @ 0x01008000 ... done
Erasing Sector 4 @ 0x01010000 ... done
```

Flash commands (3)

Storing a file in flash

- Downloading from the network:

```
u-boot # tftp 8000 u-boot.bin
```

- Copy to flash (`0x01000000`: first sector)

```
u-boot # cp.b ${fileaddr} 1000000 ${filesize}
Copy to Flash... ................ done
```

- Restore flash sector protection:

```
u-boot # protect on 1:0-4
Protect Flash Sectors 0-5 in Bank # 1
```

boot commands

- Specify kernel boot parameters:

```
u-boot # setenv bootargs mem=64M \
console=ttyS0,115200 init=/sbin/init \
root=/dev/mtdblock0
```

Continues on the same line

- Execute the kernel from a given physical address (RAM or flash):

```
bootm 0x01030000
```

Useful links

- U-boot home page:
  http://www.denx.de/wiki/UBoot/WebHome
- Very nice overview about U-boot
  (which helped to create this section):
  http://linuxdevices.com/articles/AT5085702347.html
- The U-boot manual:
  http://www.denx.de/wiki/view/DULG/UBoot

Back to the bootloaders section.

Annexes

Grub details

Grub features (1)

- Many features and a lot of flexibility!
- Supports booting many operating systems:
  Linux, Hurd, *BSD, Windows, DOS, OS/2...
- Support for different boot devices: hard disk (of course), cdrom (El Torito), network (tftp)
- Support for many filesystems (unlike LILO, it doesn't need to store the physical location of each kernel):
  ext2/3, xfs, jfs, reiserfs, dos, fat16, fat32...
- Configuration file: unlike LILO, no need to update the MBR after making changes to the configuration file.

Grub features (2)

- Support for many network cards
  (reusing drivers from the Etherboot bootloader).
- Menu interface for regular users.
  Advanced command line interface for advanced users.
- Remote control from a serial console.
- Supports multiple executable formats:
  ELF by also a.out variants.
- Can uncompress compressed files
- Small: possible to remove features and drivers
  which are not used (`./configure --help`).
  Without recompiling: remove unused filesystem stages.

Grub size

Example from grub 0.97-1ubuntu9 (Ubuntu Dapper):

- Stage 1:
  `/lib/grub/i386-pc/stage1`: 512 bytes
- Stage 1.5:
  `/lib/grub/i386-pc/e2fs_stage1_5`: 7508 bytes
- Stage 2:
  `/lib/grub/i386-pc/stage2`: 105428 bytes

Total: only 113448 bytes!

Installing grub (1)

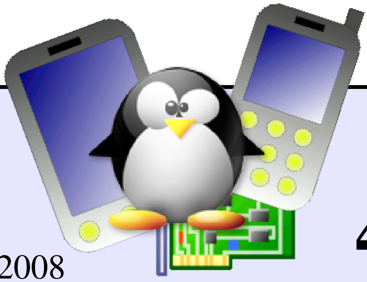
Install Grub on an embedded target with a blank disk.

- Do it from a GNU/Linux host with Grub installed.
- Access the disk for the embedded target as external storage:
  - Compact Flash disk: use a USB CF card reader.
  - Hard disk drive: use a USB hard disk drive enclosure.
- Create a partition on this disk (useful, but not mandatory):
  `fdisk /dev/sda` (type `m` for a menu of commands)
- Format and mount this partition:

```
mkfs.ext3 /dev/sda1
sudo mount /dev/sda1 /mnt/sda1
```

Installing grub (2)

- Install Grub:
  `grub-install --root-directory=/mnt/sda1 /dev/sda`
  - `/dev/sda`: the physical disk. Grub is installed on its Master Boot Record.
  - `/mnt/sda1`: the directory under which `grub-install` creates a `boot/` directory containing the upper stage and configuration file. Of course, you could have used another partition.
- Grub now needs a kernel to boot. Copy a kernel image to `/mnt/sda1/boot/` (for example) and describe this kernel in `/mnt/sda1/boot/grub/menu.lst`.
- Once you also copied root filesystem files, you can put your storage device back to the embedded target and boot from it.



Naming files

- Grub names partitions as follows: `(hd`*`n`*`,`*`p`*`)`
  n: *n*th disk on the system
  p: *p*th partition on this disk
- Files are described with the partition they belong to.
  Example: `(hd0,2)/boot/vmlinuz-2.6.18`
- You can specify a default partition with the `root` command:
  Example:

```
root (hd0,0)
kernel /boot/vmlinuz-2.6.18
```

Sample configuration file

/boot/grub/menu.lst

```
default 0
timeout 10

title      Ubuntu, kernel 2.6.15-27-386
root       (hd0,2)
kernel     /boot/vmlinuz-2.6.15-27-386 root=/dev/hda3 ro quiet splash
initrd     /boot/initrd.img-2.6.15-27-386
boot

title      Ubuntu, kernel 2.6.15-27-386 (recovery mode)
root       (hd0,2)
kernel     /boot/vmlinuz-2.6.15-27-386 root=/dev/hda3 ro single
initrd     /boot/initrd.img-2.6.15-27-386
boot
```

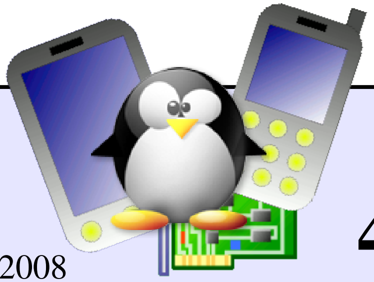
Network support

Grub can use the network in several ways

- Grub running from disk (floppy, hard drive, cdrom), and downloading kernel images from a tftp server on the network.
- Diskless system:
  - A first stage bootloader (typically Etherboot) is booted from ROM.
  - It then downloads a second stage from Grub: `pxegrub` for a PXE ROM, or `nbgrub` for a NBI loader).
  - Grub can then get kernel images from the network.

Grub security (1)

- Caution: the Grub shell can be used to display any of your files!
- Example:
  - Boot your system
  - Type the `c` command to enter command line mode.
  - `find /etc/passwd`
    Grub displays all partitions containing such a file.
  - `cat (hd0,2)/etc/passwd`
    You can see the names of users on the system!
    Of course, you can access any file. Permissions are ignored.



Grub security (2)

- Interactive commands can be protected with a password. Otherwise, people would even be able to view the contents of files from the Grub shell!
- You can also protect menu entries with a password. Useful to restrict failsafe modes to admin users.

Grub resources

- Grub home page:
  http://www.gnu.org/software/grub/
- Grub manual:
  http://www.gnu.org/software/grub/manual/

Back to the bootloaders section.

Annexes

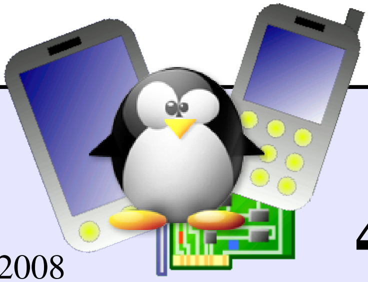
Init runlevels

System V init runlevels (1)

- Introduced by System V Unix
  Much more flexible than in BSD
- Make it possible to start or stop different services for each runlevel
- Correspond to the argument given to `/sbin/init`.
- Runlevels defined in `/etc/inittab`.

`/etc/initab` excerpt:

```
id:5:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
```



System V init runlevels (2)

Standard levels

- `init 0`

  Halt the system
- `init 1`

  Single user mode for maintenance
- `init 6`

  Reboot the system
- `init S`

  Single user mode for maintenance. Mounting only `/`. Often identical to `1`

Customizable levels: `2`, `3`, `4`, `5`

- `init 3`

  Often multi-user mode, with only command-line login
- `init 5`

  Often multi-user mode, with graphical login

init scripts

According to `/etc/inittab` settings, `init <n>` runs:

- First `/etc/rc.d/rc.sysinit` for all runlevels
- Then scripts in `/etc/rc<n>.d/`
- Starting services (`1`, `3`, `5`, `S`):
  runs `S*` scripts with the `start` option
- Killing services (`0`, `6`):
  runs `K*` scripts with the `stop` option
- Scripts are run in file name lexical order
  Just use `ls -l` to find out the order!

/etc/init.d

- Repository for all available init scripts
- /etc/rc<n>.d/ only contains links to the /etc/init.d/ scripts needed for runlevel n
- /etc/rc1.d/ example (from Fedora Core 3)

```
K01yum -> ../init.d/yum
K02cups-config-daemon -> ../init.d/cups-
config-daemon
K02haldaemon -> ../init.d/haldaemon
K02NetworkManager ->
../init.d/NetworkManager
K03messagebus -> ../init.d/messagebus
K03rhnsd -> ../init.d/rhnsd
K05anacron -> ../init.d/anacron
K05atd -> ../init.d/atd
```

```
S00single -> ../init.d/single
S01sysstat -> ../init.d/sysstat
S06cpuspeed -> ../init.d/cpuspeed
```

Handling init scripts by hand

Simply call the `/etc/init.d` scripts!

-
  ```
  sudo /etc/init.d/sshd start
  Starting sshd:                      [  OK  ]
  ```

-
  ```
  sudo /etc/init.d/nfs stop
  Shutting down NFS mountd:           [FAILED]
  Shutting down NFS daemon:
  [FAILED]Shutting down NFS quotas:
  [FAILED]
  Shutting down NFS services:         [  OK  ]
  ```

-
  ```
  sudo /etc/init.d/pcmcia status
  cardmgr (pid 3721) is running...
  ```

-
  ```
  sudo /etc/init.d/httpd restart
  Stopping httpd:                     [  OK  ]
  Starting httpd:                     [  OK  ]
  ```

Init runlevels - Useful links

Back to the slide about the init program.

Training labs

Training labs are also available from the same location:

http://free-electrons.com/training/drivers

They are a useful complement to consolidate what you learned from this training. They don't tell *how* to do the exercises. However, they only rely on notions and tools introduced by the lectures.

If you happen to be stuck with an exercise, this proves that you missed something in the lectures and have to go back to the slides to find what you're looking for.

Related documents

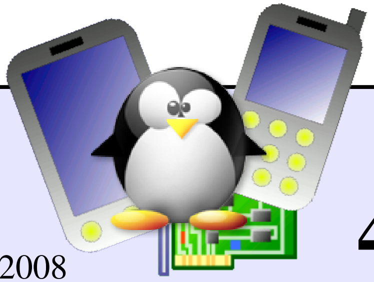
All the technical presentations and training materials created and used by Free Electrons, available under a free documentation license (more than 1500 pages!).

http://free-electrons.com/training

- Introduction to Unix and GNU/Linux
- Embedded Linux kernel and driver development
- Free Software tools for embedded Linux systems
- Audio in embedded Linux systems
- Multimedia in embedded Linux systems

http://free-electrons.com/articles

- Advantages of Free Software in embedded systems
- Embedded Linux optimizations
- Embedded Linux from Scratch... in 40 min!
- Linux USB drivers
- Real-time in embedded Linux systems
- Introduction to uClinux
- Linux on TI OMAP processors
- Free Software development tools
- Java in embedded Linux systems
- Introduction to GNU/Linux and Free Software
- Linux and ecology
- What's new in Linux 2.6?
- How to port Linux on a new PDA



How to help

If you support this work, you can help ...

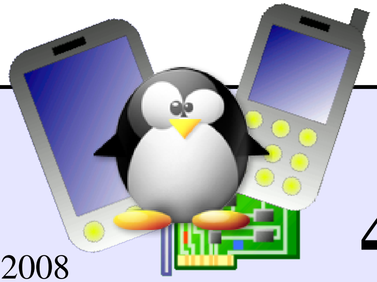
- By sending corrections, suggestions, contributions and translations
- By asking your organization to order training sessions performed by the author of these documents (see http://free-electrons.com/training)
- By speaking about it to your friends, colleagues and local Free Software community.
- By adding links to our on-line materials on your website, to increase their visibility in search engine results.

Thanks

- To the OpenOffice.org project, for their presentation and word processor tools which satisfied all my needs
- To http://openclipart.org project contributors for their nice public domain clipart
- To the Handhelds.org community, for giving me so much help and so many opportunities to help.
- To the members of the whole Free Software and Open Source community, for sharing the best of themselves: their work, their knowledge, their friendship.
- To Bill Gates, for leaving us with so much room for innovation!

To people who helped, sent corrections or suggestions:

Vanessa Conchodon, Stéphane Rubino, Samuli Jarvinen, Phil Blundell, Jeffery Huang, Mohit Mehta, Matti Aaltonen, **Robert P.J. Day**



Free Electrons services

Embedded Linux Training

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

Custom Development

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

Consulting

Help in decision making
System architecture
Identification of suitable technologies
Managing licensing requirements
System design and performance review

Technical Support

Development tool and application support
Issue investigation and solution follow-up with mainstream developers
Help getting started

http://free-electrons.com

Free Electrons
Free Software for Embedded Systems