Linux Kernel and Driver Development Training

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Latest update: March 29, 2013.
Document updates and sources:
`http://free-electrons.com/doc/training/linux-kernel`
Corrections, suggestions, contributions and translations are welcome!

Rights to copy

Electronic copies of these documents

- Electronic copies of your particular version of the materials are available on:

  `http://free-electrons.com/doc/training/linux-kernel`
- Open the corresponding documents and use them throughout the course to look for explanations given earlier by the instructor.
- You will need these electronic versions because we neither print any index nor any table of contents (quite high environmental cost for little added value)
- For future reference, see the first slide to see where document updates will be available.

Free Electrons: not a training company

Free Electrons is an engineering company, not a training company

- Training is just one of our activities
- Whether they are directly employed by Free Electrons, or whether they are external developers that we know very well, all our trainers are engineers first, with extensive on-the-job experience.
- Free Electrons engineers spend most of their time on technical projects, and share this experience through training sessions and by keeping our training materials up to date.
- All our trainers also spend a lot of time contributing to the user and developer community, by contributing to projects (such as the Linux kernel, Buildroot and Barebox), and/or by sharing technical information (through blog posts, training materials and talks at international conferences)

Free Electrons at a glance

- Created in 2004
- Locations: Orange, Toulouse, Saint Etienne / Lyon (France)
- Serving customers all around the world
  See http://free-electrons.com/company/customers/
- Head count: 7
  Only Free Software enthusiasts!
- Focus: Embedded Linux, Linux kernel, Android Free Software / Open Source for embedded and real-time systems.
- Activities: development, training, consulting, technical support.
- Added value: get the best of the user and development community and the resources it offers.

Free Electrons: what's special

- Engineers recruited in the heart of the embedded Linux developer community.
- We are very familiar with the best solutions the community offers to product developers.
- Contributing as much as possible to the community: code, documentation, knowledge sharing, financial support.
- Our engineers regularly go to the top technical conferences. We know other developers very well.
- Nothing proprietary in Free Electrons. Everything we produce for our company is shared and transparent (in particular training materials and even evaluations from all our training sessions!).

Our main services

- Linux kernel and board support package development, to support new an custom hardware: bootloader, initialization, device drivers, power management...
- Linux kernel mainlining: integrate support for your hardware in the official Linux kernel sources
- Android porting and customization
- System development and building environment. Buildroot, OpenEmbedded and Yocto support.
- System integration: choosing the best components and making a custom system.
- Boot time reduction

Free Electrons on-line resources

- All our training materials:
  http://free-electrons.com/docs/
- Technical blog:
  http://free-electrons.com/blog/
- Quarterly newsletter:
  http://lists.free-electrons.com/mailman/listinfo/newsletter
- News and discussions (LinkedIn):
  http://linkedin.com/groups/Free-Electrons-4501089
- Quick news (Twitter):
  http://twitter.com/free_electrons

Generic course information

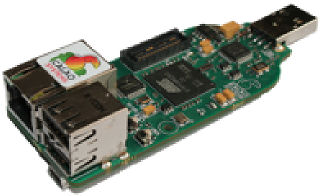
Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Corrections, suggestions, contributions and translations are welcome!

Hardware used in this training session

Calao Systems USB-A9263 Supported in mainstream Linux since version 2.6.27!

- AT91SAM9263 ARM CPU
- 64 MB RAM, 256 MB flash
- 2 USB 2.0 host, 1 USB device
- 100 Mbit Ethernet port
- Powered by USB!
- Serial and JTAG through this USB port.
- Multiple extension boards.
- Approximately 160 EUR (V.A.T. not included)



Participate!

During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- This helps the trainer to detect any explanation that wasn't clear or detailed enough.
- Don't hesitate to share your experience, for example to compare Linux / Android with other operating systems used in your company.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- Your participation can make our session more interactive and make the topics easier to learn.

Practical lab guidelines

During practical labs...

- We cannot support more than 8 workstations at once (each with its board and equipment). Having more would make the whole class progress slower, compromising the coverage of the whole training agenda (exception for public sessions: up to 10 people).
- So, if you are more than 8 participants, please form up to 8 working groups.
- Open the electronic copy of your lecture materials, and use it throughout the practical labs to find the slides you need again.
- Don't copy and paste from the PDF slides.
  The slides contain UTF-8 characters that look the same as ASCII ones, but won't be understood by shells or compilers.

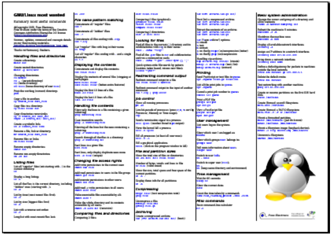
Cooperate!

As in the Free Software and Open Source community, cooperation during practical labs is valuable in this training session:

- If you complete your labs before other people, don't hesitate to help other people and investigate the issues they face. The faster we progress as a group, the more time we have to explore extra topics.
- Explain what you understood to other participants when needed. It also helps to consolidate your knowledge.
- Don't hesitate to report potential bugs to your instructor.
- Don't hesitate to look for solutions on the Internet as well.

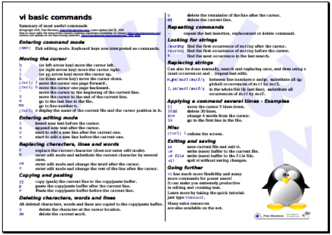
Command memento sheet

- This memento sheet gives command examples for the most typical needs (looking for files, extracting a tar archive...)
- It saves us 1 day of UNIX / Linux command line training.
- Our best tip: in the command line shell, always hit the `Tab` key to complete command names and file paths. This avoids 95% of typing mistakes.
- Get an electronic copy on `http://free-electrons.com/docs/command-line`



vi basic commands

- The vi editor is very useful to make quick changes to files in a embedded target.
- Though not very user friendly at first, vi is very powerful and its main 15 commands are easy to learn and are sufficient for 99% of everyone's needs!
- Get an electronic copy on `http://free-electrons.com/docs/command-line`
- You can also take the quick tutorial by running `vimtutor`. This is a worthy investment!



vi basic commands

Summary of most useful commands

Entering command mode

[ESC] Exit editing mode. Keyboard keys now interpreted as commands.

Moving the cursor

h (or left arrow key) move the cursor left.
l (or right arrow key) move the cursor right.
j (or up arrow key) move the cursor up.
k (or down arrow key) move the cursor down.
[Ctrl] f move the cursor one page forward.
[Ctrl] b move the cursor one page backward.
^ move the cursor to the beginning of the current line.
$ move the cursor to the end of the current line.
G go to the last line in the file.
nG go to line number n.
[Ctrl] g display the name of the current file and the cursor position in it.

Entering editing mode

i insert new text before the cursor.
a append new text after the cursor.
o start to edit a new line after the current one.
O start to edit a new line before the current one.

Replacing characters, lines and words

r replace the current character (does not enter edit mode).
R enter edit mode and substitute the current character by several ones.
cw enter edit mode and change the word after the cursor.
C enter edit mode and change the rest of the line after the cursor.

Copying and pasting

yy copy (yank) the current line to the copy/paste buffer.
p paste the copy/paste buffer after the current line.
P Paste the copy/paste buffer before the current line.

Deleting characters, words and lines

All deleted characters, words and lines are copied to the copy/paste buffer.
x delete the character at the cursor location.
dw delete the current word.
D delete the remainder of the line after the cursor.
dd delete the current line.

Repeating commands

. repeat the last insertion, replacement or delete command.

Looking for strings

/string find the first occurrence of string after the cursor.
?string find the first occurrence of string before the cursor.
n find the next occurrence in the last search.

Replacing strings

Can also be done manually, search and replacing once, and then using n (next occurrence) and . (repeat last edit).

n,p s/str1/str2/g between line numbers n and p, substitute all (g: global) occurrences of str1 by str2.
1,$ s/str1/str2/g in the whole file ($: last line), substitute all occurrences of str1 by str2.

Applying a command several times - Examples

5j move the cursor 5 lines down.
30dd delete 30 lines.
4cw change 4 words from the cursor.
1G go to the first line in the file.

Misc

[Ctrl] l redraw the screen.

Exiting and saving

ZZ save current file and exit vi.
:w write (save) buffer to the current file.
:w file write (save) buffer to the file file.
:q! quit vi without saving changes.

Going further

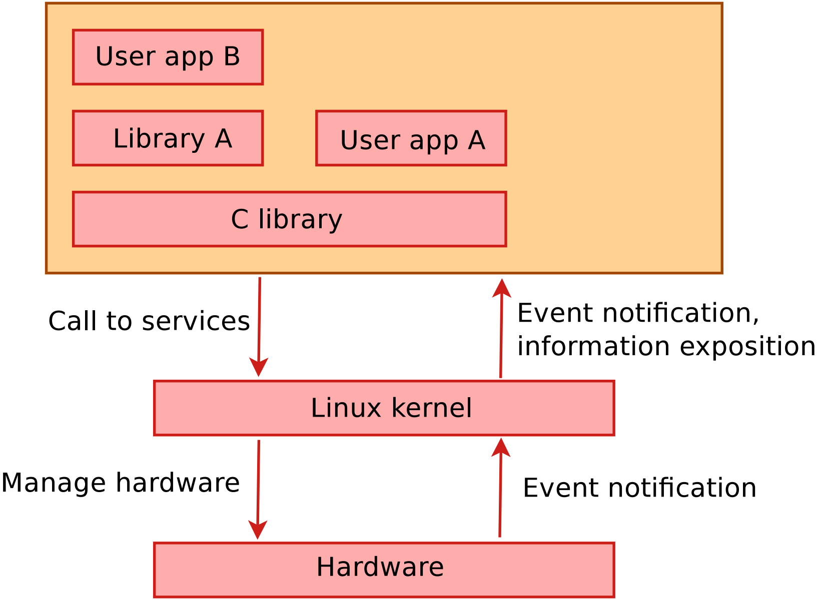
vi has much more flexibility and many more commands for power users!
It can make you extremely productive in editing and creating text.
Learn more by taking the quick tutorial: just type vimtutor.
Many extra resources are also available on the net.

Linux Kernel Introduction

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Linux features

Linux kernel in the system



History

- The Linux kernel is one component of a system, which also requires libraries and applications to provide features to end users.
- The Linux kernel was created as a hobby in 1991 by a Finnish student, Linus Torvalds.
  - Linux quickly started to be used as the kernel for free software operating systems
- Linus Torvalds has been able to create a large and dynamic developer and user community around Linux.
- Nowadays, hundreds of people contribute to each kernel release, individuals or companies big and small.

Linux license

- The whole Linux sources are Free Software released under the GNU General Public License version 2 (GPL v2).
- For the Linux kernel, this basically implies that:
  - When you receive or buy a device with Linux on it, you should receive the Linux sources, with the right to study, modify and redistribute them.
  - When you produce Linux based devices, you must release the sources to the recipient, with the same rights, with no restriction..

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, and gcc support
- 32 bit architectures (arch/ subdirectories)
  Examples: arm, avr32, blackfin, m68k, microblaze, mips, score, sparc, um
- 64 bit architectures:
  Examples: alpha, arm64, ia64, sparc64, tile
- 32/64 bit architectures
  Examples: powerpc, x86, sh
- Find details in kernel sources: arch/<arch>/Kconfig, arch/<arch>/README, or Documentation/<arch>/

System calls

- The main interface between the kernel and userspace is the set of system calls
- About 300 system calls that provide the main kernel services
  - File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- This interface is stable over time: only new system calls can be added by the kernel developers
- This system call interface is wrapped by the C library, and userspace applications usually never make a system call directly but rather use the corresponding C library function

Virtual filesystems

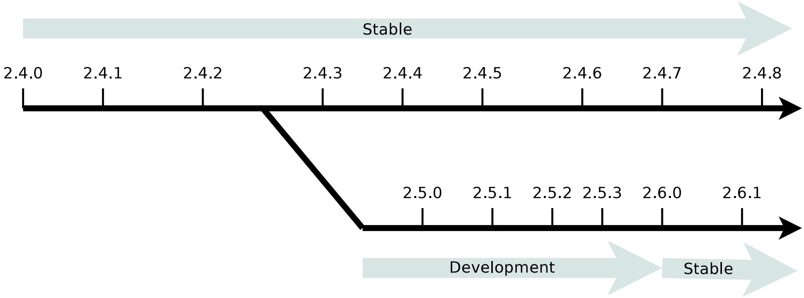
- Linux makes system and kernel information available in user-space through virtual filesystems.
- Virtual filesystems allow applications to see directories and files that do not exist on any real storage: they are created on the fly by the kernel
- The two most important virtual filesystems are
  - `proc`, usually mounted on `/proc`:
    Operating system related information (processes, memory management parameters...)
  - `sysfs`, usually mounted on `/sys`:
    Representation of the system as a set of devices and buses. Information about these devices.

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.x, 2.0.x, 2.2.x, 2.4.x`
- One development branch to integrate new functionalities and major changes
  - Identified by an odd middle number
  - Examples: `2.1.x, 2.3.x, 2.5.x`
  - 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)


Stable
2.4.0
2.4.1
2.4.2
2.4.3
2.4.4
2.4.5
2.4.6
2.4.7
2.4.8
2.5.0
2.5.1
2.5.2
2.5.3
2.6.0
2.6.1
Development
Stable

Changes since Linux 2.6 (1)

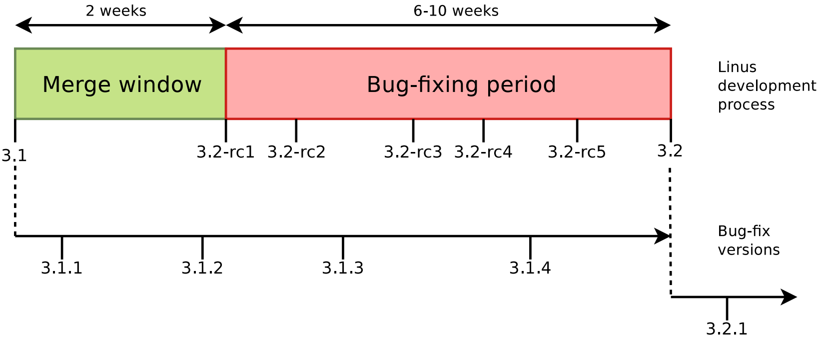
- Since `2.6.0`, kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make major changes in existing subsystems.
- So far, there was no need to create a new development branch (such as `2.7`), which would massively break compatibility with the stable branch.
- 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 6 to 10 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



More stability for the kernel source tree

- Issue: bug and security fixes only released for most recent stable kernel versions.
- Some people need to have a recent kernel, but with long term support for security updates.
- You could get long term support from a commercial embedded Linux provider.
- You could reuse sources for the kernel used in Ubuntu Long Term Support releases (5 years of free security updates).
- The `http://kernel.org` front page shows which versions will be supported for some time (up to 2 or 3 years), and which ones won't be supported any more ("EOL: End Of Life")

stable:	**3.7.1**
mainline:	**3.7**
stable:	**3.6.11 (EOL)**
stable:	**3.5.7 (EOL)**
stable:	**3.4.24**
stable:	**3.2.35**
stable:	**3.0.57**
stable:	**2.6.34.13**
stable:	**2.6.32.60**
linux-next:	**next-20121220**

New 3.x branch

- From 2003 to 2011, the official kernel versions were named `2.6.x`.
- Linux `3.0` was released in July 2011
- There is no change to the development model, only a change to the numbering scheme
  - Official kernel versions will be named `3.x` (`3.0, 3.1, 3.2`, etc.)
  - Stabilized versions will be named `3.x.y` (`3.0.2, 3.4.3`, etc.)
  - It effectively only removes a digit compared to the previous numbering scheme

What's new in each Linux release?

- The official list of changes for each Linux release is just a huge list of individual patches!

```
commit aa6e52a35d388e730f4df0ec2ec48294590cc459
Author: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
Date:   Wed Jul 13 11:29:17 2011 +0200

    at91: at91-ohci: support overcurrent notification

    Several USB power switches (AIC1526 or MIC2026) have a digital output
    that is used to notify that an overcurrent situation is taking
    place. This digital outputs are typically connected to GPIO inputs of
    the processor and can be used to be notified of those overcurrent
    situations.

    Therefore, we add a new overcurrent_pin[] array in the at91_usbh_data
    structure so that boards can tell the AT91 OHCI driver which pins are
    used for the overcurrent notification, and an overcurrent_supported
    boolean to tell the driver whether overcurrent is supported or not.

    The code has been largely borrowed from ohci-da8xx.c and
    ohci-s3c2410.c.

    Signed-off-by: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
    Signed-off-by: Nicolas Ferre <nicolas.ferre@atmel.com>
```

  - Very difficult to find out the key changes and to get the global picture out of individual changes.
- Fortunately, there are some useful resources available
  - http://wiki.kernelnewbies.org/LinuxChanges
  - http://lwn.net
  - http://linuxfr.org, for French readers

Practical lab - Training Setup

Prepare your lab environment

- Download the lab archive
- Enforce correct permissions

Embedded Linux Kernel Usage

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Corrections, suggestions, contributions and translations are welcome!

Linux kernel sources

Location of kernel sources

- The official version of the Linux kernel, as released by Linus Torvalds is available at `http://www.kernel.org`
  - This version follows the well-defined development model of the kernel
  - However, it may not contain the latest development from a specific area, due to the organization of the development model and because features in development might not be ready for mainline inclusion
- Many kernel sub-communities maintain their own kernel, with usually newer but less stable features
  - Architecture communities (ARM, MIPS, PowerPC, etc.), device drivers communities (I2C, SPI, USB, PCI, network, etc.), other communities (real-time, etc.)
  - They generally don't release official versions, only development trees are available

Linux kernel size (1)

- Linux 3.1 sources:
  Raw size: 434 MB (39,400 files, approx 14,800,000 lines)
  `gzip` compressed tar archive: 93 MB
  `bzip2` compressed tar archive: 74 MB (better)
  `xz` compressed tar archive: 62 MB (best)
- Minimum Linux 2.6.29 compiled kernel size with `CONFIG_EMBEDDED`, for a kernel that boots a QEMU PC (IDE hard drive, ext2 filesystem, ELF executable support):
  532 KB (compressed), 1325 KB (raw)
- 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)

As of kernel version 3.2.

- drivers/: 53.65%
- arch/: 20.78%
- fs/: 6.88%
- sound/: 5.04%
- net/: 4.33%
- include/: 3.80%
- firmware/: 1.46%
- kernel/: 1.10%
- tools/: 0.56%
- mm/: 0.53%
- scripts/: 0.44%
- security/: 0.40%
- crypto/: 0.38%
- lib/: 0.30%
- block/: 0.13%
- ipc/: 0.04%
- virt/: 0.03%
- init/: 0.03%
- samples/: 0.02%
- usr/: 0%

Getting Linux sources

- **Full tarballs**
  - Contain the complete kernel sources: long to download and uncompress, but must be done at least once
  - Example:

    `http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.1.3.tar.xz`
  - Extract command:

    `tar Jxf linux-3.1.3.tar.xz`
- **Incremental patches between versions**
  - It assumes you already have a base version and you apply the correct patches in the right order. Quick to download and apply
  - Examples:
    `http://www.kernel.org/pub/linux/kernel/v3.0/patch-3.1.xz`
    (3.0 to 3.1)
    `http://www.kernel.org/pub/linux/kernel/v3.0/patch-3.1.3.xz`
    (3.1 to 3.1.3)
- All previous kernel versions are available in `http://kernel.org/pub/linux/kernel/`

Patch

- A patch is the difference between two source trees
  - Computed with the `diff` tool, or with more elaborate version control systems
- They are very common in the open-source community
- Excerpt from a patch:

```
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*
```

Contents of a patch

- One section per modified file, starting with a header

```
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
```

- One sub-section per modified part of the file, starting with header with the affected line numbers

```
@@ -1,7 +1,7 @@
```

- Three lines of context before the change

```
 VERSION = 2
 PATCHLEVEL = 6
 SUBLEVEL = 11
```

- The change itself

```
-EXTRAVERSION =
+EXTRAVERSION = .1
```

- Three lines of context after the change

```
 NAME=Woozy Numbat

 # *DOCUMENTATION*
```

Using the patch command

The patch command:

- Takes the patch contents on its standard input
- Applies the modifications described by the patch into the current directory

patch usage examples:

- patch -p<n> < diff_file
- cat diff_file | patch -p<n>
- xzcat diff_file.xz | patch -p<n>
- bzcat diff_file.bz2 | patch -p<n>
- zcat diff_file.gz | patch -p<n>
- Notes:
  - n: number of directory levels to skip in the file paths
  - You can reverse apply a patch with the -R option
  - You can test a patch with --dry-run option

Applying a Linux patch

Linux patches...

- Always applied to the `x.y.<z-1>` version
  Can be downloaded in `gzip`, `bzip2` or `xz` (much smaller) compressed files.
- Always produced for `n=1`
  (that's what everybody does... do it too!)
- Need to run the `patch` command inside the kernel source directory
- Linux patch command line example:

```
cd linux-3.0
xzcat ../patch-3.1.xz | patch -p1
xzcat ../patch-3.1.3.xz | patch -p1
cd ..; mv linux-3.0 linux-3.1.3
```

Kernel Source Code

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Linux Code and Device Drivers

Supported kernel version

- The APIs covered in these training slides should be compliant with Linux 3.6.
- We may also mention features in more recent kernels.

Programming language

- Implemented in C like all Unix systems. (C was created to implement the first Unix systems)
- A little Assembly is used too:
  - CPU and machine initialization, exceptions
  - Critical library routines.
- No C++ used, see `http://www.tux.org/lkml/#s15-3`
- All the code compiled with gcc
  - Many gcc specific extensions used in the kernel code, any ANSI C compiler will not compile the kernel
  - A few alternate compilers are supported (Intel and Marvell)
  - See `http://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/C-Extensions.html`

No C library

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

Portability

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

No floating point computation

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

No stable Linux internal API 1/3

- The internal kernel API to implement kernel code can undergo changes between two stable 2.6.x or 3.x releases. A stand-alone driver compiled for a given version may no longer compile or work on a more recent one. See `Documentation/stable_api_nonsense.txt` in kernel sources for reasons why.
- Of course, the external API must not change (system calls, `/proc`, `/sys`), as it could break existing programs. New features can be added, but kernel developers try to keep backward compatibility with earlier versions, at least for 1 or several years.

No stable Linux internal API 2/3

- 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 3/3

- USB example
  - Linux has updated its USB internal API at least 3 times (fixes, security issues, support for high-speed devices) and has now the fastest USB bus speeds (compared to other systems)
  - Windows XP also had to rewrite its USB stack 3 times. But, because of closed-source, binary drivers that can't be updated, they had to keep backward compatibility with all earlier implementation. This is very costly (development, security, stability, performance).
- See "Myths, Lies, and Truths about the Linux Kernel", by Greg K.H., for details about the kernel development process: `http://kroah.com/log/linux/ols_2006_keynote.html`

Kernel memory constraints

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

Linux kernel licensing constraints

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

Proprietary code and the kernel

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

Advantages of GPL drivers 1/2

- 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.

Advantages of GPL drivers 2/2

- 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 (useful to have a kernel image with all drivers needed at boot time)

Advantages of in-tree kernel drivers

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

Userspace device drivers 1/2

- 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 or USB drivers)
  - X drivers: low level kernel drivers + user space X drivers.
  - Userspace drivers based on UIO. See `Documentation/DocBook/uio-howto` in the kernel documentation for details about UIO and the *Using UIO on an Embedded platform* talk at ELC 2008 (`http://j.mp/tBzayM`)

Userspace device drivers 2/2

- Advantages
  - No need for kernel coding skills. Easier to reuse code between devices.
  - Drivers can be written in any language, even Perl!
  - Drivers can be kept proprietary.
  - Driver code can be killed and debugged. Cannot crash the kernel.
  - Can be swapped out (kernel code cannot be).
  - Can use floating-point computation.
  - Less in-kernel complexity.
- Drawbacks
  - Less straightforward to handle interrupts.
  - Increased latency vs. kernel code.

Linux sources

Linux sources structure 1/4

- `arch/<architecture>`
  - Architecture specific code
- `arch/<architecture>/include/asm`
  - Architecture and machine dependent headers
- `arch/<architecture>/mach-<machine>`
  - Machine/board specific code
- `block`
  - Block layer core
- `COPYING`
  - Linux copying conditions (GNU GPL)
- `CREDITS`
  - Linux main contributors
- `crypto/`
  - Cryptographic libraries

Linux sources structure 2/4

- 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/linux
  - Linux kernel core headers
- init/
  - Linux initialization (including main.c)
- ipc/
  - Code used for process communication

Linux sources structure 3/4

- 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)

Linux sources structure 4/4

- `README`
  - Overview and building instructions
- `REPORTING-BUGS`
  - Bug report instructions
- `samples/`
  - Sample code (markers, kprobes, kobjects...)
- `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.

Accessing development sources 1/2

- Useful if you are involved in kernel development or if you found a bug in the source code.
- Kernel development sources are now managed with Git: `http://git-scm.com/`
- You can browse Linus' Git tree (if you just need to check a few files): `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git;a=tree` (`http://j.mp/QaOrzP`)
- You can also directly use Git on your workstation
  - Debian / Ubuntu: install the `git` package

Accessing development sources 2/2

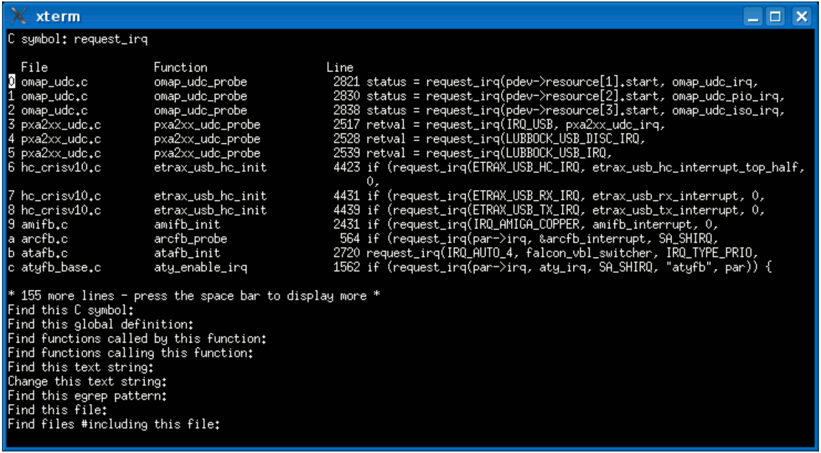
- Choose a Git development tree on `http://git.kernel.org/`
- Get a local copy ("clone") of this tree.
  - `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
- Update your copy whenever needed: `git pull`
- More details in our chapter about Git

Kernel source management tools

Cscope

- `http://cscope.sourceforge.net/`
  - Tool to browse source code (mainly C, but also C++ or Java)
  - Supports huge projects like the Linux kernel. Takes less than 1 min. to index Linux 2.6.17 sources (fast!)
  - Can be used from editors like `vim` and `emacs`.
  - In Linux kernel sources, run it with: `cscope -Rk` (see man cscope for details)
  - `KScope`: graphical front-end (`kscope` package in Ubuntu 12.04 and later)
  - Allows searching for a symbol, a definition, functions, strings, files, etc.

Cscope screenshot



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
  - Very easy to find the declaration, implementation or usage of symbols
  - Supports C and C++
  - Supports huge code projects such as the Linux kernel (431 MB of source code in version 3.0).
  - Takes a little time and patience to setup (configuration, indexing, web server configuration)
  - You don't need to set up LXR by yourself. Use our `http://lxr.free-electrons.com` server!

LXR screenshot

Practical lab - Kernel Source Code

- Get the Linux kernel sources
- Apply patches
- Explore sources manually
- Use automated tools to explore the source code

Kernel configuration

Kernel configuration and build system

- The kernel configuration and build system is based on multiple Makefiles
- One only interacts with the main `Makefile`, present at the **top directory** of the kernel source tree
- Interaction takes place
  - using the `make` tool, which parses the Makefile
  - through various **targets**, defining which action should be done (configuration, compilation, installation, etc.). Run `make help` to see all available targets.
- Example
  - `cd linux-3.6.x/`
  - `make <target>`

Kernel configuration (1)

- The kernel contains thousands of device drivers, filesystem drivers, network protocols and other configurable items
- Thousands of options are available, that are used to selectively compile parts of the kernel source code
- The kernel configuration is the process of defining the set of options with which you want your kernel to be compiled
- The set of options depends
  - On your hardware (for device drivers, etc.)
  - On the capabilities you would like to give to your kernel (network capabilities, filesystems, real-time, etc.)

Kernel configuration (2)

- The configuration is stored in the `.config` file at the root of kernel sources
  - Simple text file, `key=value` style
- As options have dependencies, typically never edited by hand, but through graphical or text interfaces:
  - `make xconfig`, `make gconfig` (graphical)
  - `make menuconfig`, `make nconfig` (text)
  - You can switch from one to another, they all load/save the same `.config` file, and show the same set of options
- To modify a kernel in a GNU/Linux distribution: the configuration files are usually released in `/boot/`, together with kernel images: `/boot/config-3.2.0-31-generic`

Kernel or module?

- The **kernel image** is a **single file**, resulting from the linking of all object files that correspond to features enabled in the configuration
  - This is the file that gets loaded in memory by the bootloader
  - All included features are therefore available as soon as the kernel starts, at a time where no filesystem exists
- Some features (device drivers, filesystems, etc.) can however be compiled as **modules**
  - Those are *plugins* that can be loaded/unloaded dynamically to add/remove features to the kernel
  - Each **module is stored as a separate file in the filesystem**, and therefore access to a filesystem is mandatory to use modules
  - This is not possible in the early boot procedure of the kernel, because no filesystem is available

Kernel option types

- There are different types of options
  - `bool` options, they are either
    - *true* (to include the feature in the kernel) or
    - *false* (to exclude the feature from the kernel)
  - `tristate` options, they are either
    - *true* (to include the feature in the kernel image) or
    - *module* (to include the feature as a kernel module) or
    - *false* (to exclude the feature)
  - `int` options, to specify integer values
  - `string` options, to specify string values

Kernel option dependencies

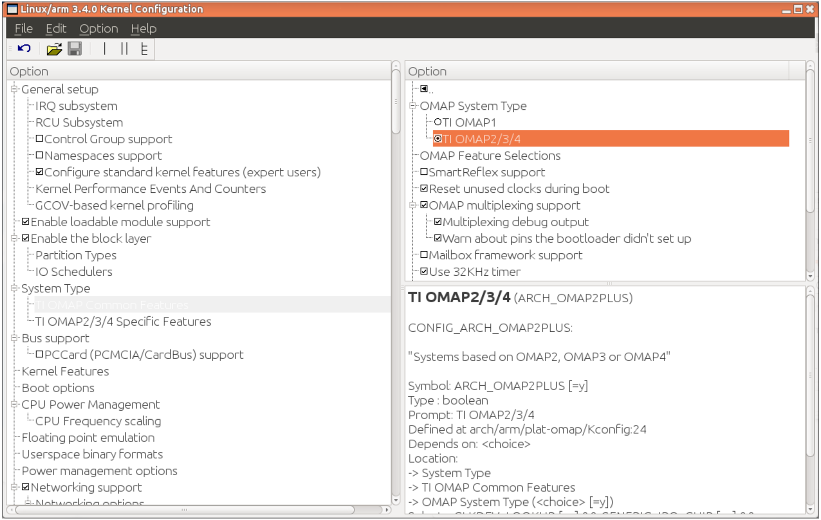
- There are dependencies between kernel options
- For example, enabling a network driver requires the network stack to be enabled
- Two types of dependencies
  - `depends on` dependencies. In this case, option A that depends on option B is not visible until option B is enabled
  - `select` dependencies. In this case, with option A depending on option B, when option A is enabled, option B is automatically enabled
  - `make xconfig` allows to see all options, even those that cannot be selected because of missing dependencies. In this case, they are displayed in gray

make xconfig

`make xconfig`

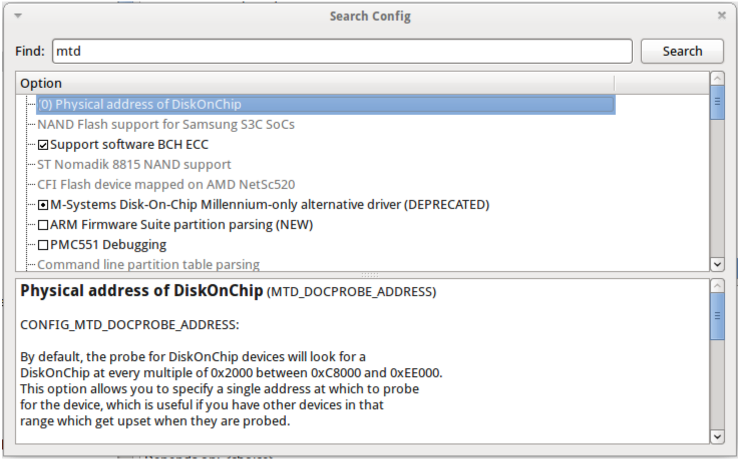
- The most common graphical interface to configure the kernel.
- Make sure you read
  `help -> introduction: useful options!`
- File browser: easier to load configuration files
- Search interface to look for parameters
- Required Debian / Ubuntu packages: `libqt4-dev g++` (`libqt3-mt-dev` for older kernel releases)

make xconfig screenshot



make xconfig search interface

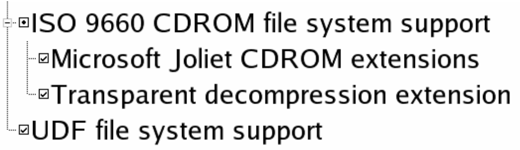
Looks for a keyword in the parameter name. Allows to select or unselect found parameters.



Kernel configuration options

Compiled as a module (separate file)
CONFIG_ISO9660_FS=m

Driver options
CONFIG_JOLIET=y
CONFIG_ZISOFS=y



Compiled statically into the kernel
CONFIG_UDF_FS=y

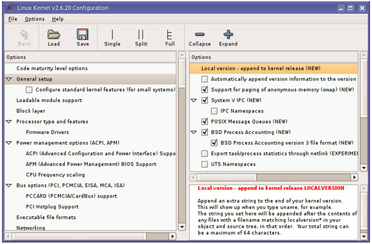
Corresponding .config file excerpt

Options are grouped by sections and are prefixed with CONFIG_.

```
#
# 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
```

`make gconfig`

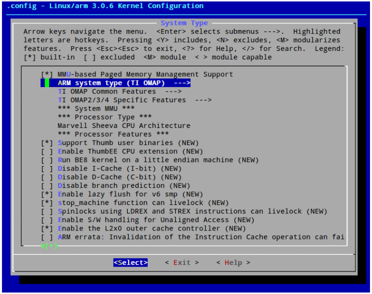
- *GTK* based graphical configuration interface. Functionality similar to that of make `xconfig`.
- Just lacking a search functionality.
- Required Debian packages: `libglade2-dev`



make menuconfig

`make menuconfig`

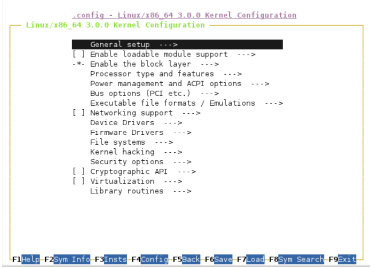
- Useful when no graphics are available. Pretty convenient too!
- Same interface found in other tools: BusyBox, Buildroot...
- Required Debian packages: `libncurses-dev`



make nconfig

`make nconfig`

- A newer, similar text interface
- More user friendly (for example, easier to access help information).
- Required Debian packages: `libncurses-dev`



make oldconfig

`make oldconfig`

- Needed very often!
- Useful to upgrade a `.config` file from an earlier kernel release
- Issues warnings for configuration parameters that no longer exist in the new kernel.
- Asks for values for new parameters

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

`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.

Configuration per architecture

- The set of configuration options is architecture dependent
  - Some configuration options are very architecture-specific
  - Most of the configuration options (global kernel options, network subsystem, filesystems, most of the device drivers) are visible in all architectures.
- By default, the kernel build system assumes that the kernel is being built for the host architecture, i.e. native compilation
- The architecture is not defined inside the configuration, but at a higher level
- We will see later how to override this behaviour, to allow the configuration of kernels for a different architecture

Overview of kernel options (1)

- General setup
  - *Local version - append to kernel release* allows to concatenate an arbitrary string to the kernel version that a user can get using `uname -r`. Very useful for support!
  - *Support for swap*, can usually be disabled on most embedded devices
  - *Configure standard kernel features (expert users)* allows to remove features from the kernel to reduce its size. Powerful, but use with care!

Overview of kernel options (2)

- Loadable module support
  - Allows to enable or completely disable module support. If your system doesn't need kernel modules, best to disable since it saves a significant amount of space and memory
- Enable the block layer
  - If `CONFIG_EXPERT` is enabled, the block layer can be completely removed. Embedded systems using only flash storage can safely disable the block layer
- Processor type and features (x86) or System type (ARM) or CPU selection (MIPS)
  - Allows to select the CPU or machine for which the kernel must be compiled
  - On x86, only optimization-related, on other architectures very important since there's no compatibility

Overview of kernel options (3)

- Kernel features
  - Tickless system, which allows to disable the regular timer tick and use on-demand ticks instead. Improves power savings
  - High resolution timer support. By default, the resolution of timer is the tick resolution. With high resolution timers, the resolution is as precise as the hardware can give
  - Preemptible kernel enables the preemption inside the kernel code (the userspace code is always preemptible). See our real-time presentation for details
- Power management
  - Global power management option needed for all power management related features
  - Suspend to RAM, CPU frequency scaling, CPU idle control, suspend to disk

Overview of kernel options (4)

- Networking support
  - The network stack
  - Networking options
    - Unix sockets, needed for a form of inter-process communication
    - TCP/IP protocol with options for multicast, routing, tunneling, Ipsec, Ipv6, congestion algorithms, etc.
    - Other protocols such as DCCP, SCTP, TIPC, ATM
    - Ethernet bridging, QoS, etc.
  - Support for other types of network
    - CAN bus, Infrared, Bluetooth, Wireless stack, WiMax stack, etc.

Overview of kernel options (5)

- Device drivers
  - MTD is the subsystem for flash (NOR, NAND, OneNand, battery-backed memory, etc.)
  - Parallel port support
  - Block devices, a few misc block drivers such as loopback, NBD, etc.
  - ATA/ATAPI, support for IDE disk, CD-ROM and tapes. A new stack exists
  - SCSI
    - The SCSI core, needed not only for SCSI devices but also for USB mass storage devices, SATA and PATA hard drives, etc.
    - SCSI controller drivers

Overview of kernel options (6)

- Device drivers (cont)
  - SATA and PATA, the new stack for hard disks, relies on SCSI
  - RAID and LVM, to aggregate hard drivers and do replication
  - Network device support, with the network controller drivers. Ethernet, Wireless but also PPP
  - Input device support, for all types of input devices: keyboards, mice, joysticks, touchscreens, tablets, etc.
  - Character devices, contains various device drivers, amongst them
    - serial port controller drivers
    - PTY driver, needed for things like SSH or telnet
  - I2C, SPI, 1-wire, support for the popular embedded buses
  - Hardware monitoring support, infrastructure and drivers for thermal sensors

Overview of kernel options (7)

- Device drivers (cont)
  - Watchdog support
  - Multifunction drivers are drivers that do not fit in any other category because the device offers multiple functionality at the same time
  - Multimedia support, contains the V4L and DVB subsystems, for video capture, webcams, AM/FM cards, DVB adapters
  - Graphics support, infrastructure and drivers for framebuffers
  - Sound card support, the OSS and ALSA sound infrastructures and the corresponding drivers
  - HID devices, support for the devices that conform to the HID specification (Human Input Devices)

Overview of kernel options (8)

- Device drivers (cont)
  - USB support
    - Infrastructure
    - Host controller drivers
    - Device drivers, for devices connected to the embedded system
    - Gadget controller drivers
    - Gadget drivers, to let the embedded system act as a mass-storage device, a serial port or an Ethernet adapter
  - MMC/SD/SDIO support
  - LED support
  - Real Time Clock drivers
  - Voltage and current regulators
  - Staging drivers, crappy drivers being cleaned up

Overview of kernel options (9)

- For some categories of devices the driver is not implemented inside the kernel
  - Printers
  - Scanners
  - Graphics drivers used by X.org
  - Some USB devices
- For these devices, the kernel only provides a mechanism to access the hardware, the driver is implemented in userspace

Overview of kernel options (10)

- File systems
  - The common Linux filesystems for block devices: ext2, ext3, ext4
  - Less common filesystems: XFS, JFS, ReiserFS, GFS2, OCFS2, Btrfs
  - CD-ROM filesystems: ISO9660, UDF
  - DOS/Windows filesystems: FAT and NTFS
  - Pseudo filesystems: proc and sysfs
  - Miscellaneous filesystems, with amongst other flash filesystems such as JFFS2, UBIFS, SquashFS, cramfs
  - Network filesystems, with mainly NFS and SMB/CIFS
- Kernel hacking
  - Debugging features useful for kernel developers

Compiling and installing the kernel for the host system

Kernel compilation

- `make`
  - in the main kernel source directory
  - Remember to run `make -j 4` if you have multiple CPU cores to speed up the compilation process
  - No need to run as root!
- Generates
  - `vmlinux`, the raw uncompressed kernel image, at the ELF format, useful for debugging purposes, but cannot be booted
  - `arch/<arch>/boot/*Image`, the final, usually compressed, kernel image that can be booted
    - `bzImage` for x86, `zImage` for ARM, `vmImage.gz` for Blackfin, etc.
  - All kernel modules, spread over the kernel source tree, as `.ko` files.

Kernel installation

- `make install`
  - Does the installation for the host system by default, so needs to be run as root. Generally not used when compiling for an embedded system, and it installs files on the development workstation.
- Installs
  - `/boot/vmlinuz-<version>`
    Compressed kernel image. Same as the one in `arch/<arch>/boot`
  - `/boot/System.map-<version>`
    Stores kernel symbol addresses
  - `/boot/config-<version>`
    Kernel configuration for this version
- Typically re-runs the bootloader configuration utility to take the new kernel into account.

Module installation

- `make modules_install`
  - Does the installation for the host system by default, so needs to be run as root
- Installs all modules in `/lib/modules/<version>/`
  - `kernel/`
    Module `.ko` (Kernel Object) files, in the same directory structure as in the sources.
  - `modules.alias`
    Module aliases for module loading utilities. Example line:
    `alias sound-service-?-0 snd_mixer_oss`
  - `modules.dep`
    Module dependencies
  - `modules.symbols`
    Tells which module a given symbol belongs to.

Kernel cleanup targets

- Clean-up generated files (to force re-compilation):
  `make clean`
- Remove all generated files. Needed when switching from one architecture to another. Caution: it also removes your `.config` file!
  `make mrproper`
- Also remove editor backup and patch reject files (mainly to generate patches):
  `make distclean`

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`, the prefix is `mips-linux-`
  `arm-linux-gnueabi-gcc`, the prefix is `arm-linux-gnueabi-`

Specifying cross-compilation (1)

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

- `ARCH` is the name of the architecture. It is defined by the name of the subdirectory in `arch/` in the kernel sources
  - Example: `arm` if you want to compile a kernel for the `arm` architecture.
- `CROSS_COMPILE` is the prefix of the cross compilation tools
  - Example: `arm-linux-` if your compiler is `arm-linux-gcc`

Specifying cross-compilation (2)

Two solutions to define `ARCH` and `CROSS_COMPILE`:

- Pass `ARCH` and `CROSS_COMPILE` on the `make` command line:
  `make ARCH=arm CROSS_COMPILE=arm-linux- ...`
  Drawback: it is easy to forget to pass these variables when you run any `make` command, causing your build and configuration to be screwed up.
- Define `ARCH` and `CROSS_COMPILE` as environment variables:
  `export ARCH=arm`
  `export CROSS_COMPILE=arm-linux-`
  Drawback: it only works inside the current shell or terminal. You could put these settings in a file that you source every time you start working on the project. If you only work on a single architecture with always the same toolchain, you could even put these settings in your `~/.bashrc` file to make them permanent and visible from any terminal.

Predefined configuration files

- Default configuration files available, per board or per-CPU family
  - They are stored in `arch/<arch>/configs/`, and are just minimal `.config` files
  - This is the most common way of configuring a kernel for embedded platforms
- Run `make help` to find if one is available for your platform
- To load a default configuration file, just run `make acme_defconfig`
  - This will overwrite your existing `.config` file!
- To create your own default configuration file
  - `make savedefconfig`, to create a minimal configuration file
  - `mv defconfig arch/<arch>/configs/myown_defconfig`

Configuring the kernel

- After loading a default configuration file, you can adjust the configuration to your needs with the normal `xconfig`, `gconfig` or `menuconfig` interfaces
- You can also start the configuration from scratch without loading a default configuration file
- As the architecture is different from your host architecture
  - Some options will be different from the native configuration (processor and architecture specific options, specific drivers, etc.)
  - Many options will be identical (filesystems, network protocol, architecture-independent drivers, etc.)
- Make sure you have the support for the right CPU, the right board and the right device drivers.

Building and installing the kernel

- Run `make`
- Copy the final kernel image to the target storage
  - can be `uImage`, `zImage`, `vmlinux`, `bzImage` in `arch/<arch>/boot`
- `make install` is rarely used in embedded development, as the kernel image is a single file, easy to handle
  - It is however possible to customize the make install behaviour in `arch/<arch>/boot/install.sh`
- `make modules_install` is used even in embedded development, as it installs many modules and description files
  - `make INSTALL_MOD_PATH=<dir>/ modules_install`
  - The `INSTALL_MOD_PATH` variable is needed to install the modules in the target root filesystem instead of your host root filesystem.

Kernel command line

- In addition to the compile time configuration, the kernel behaviour can be adjusted with no recompilation using the **kernel command line**
- The kernel command line is a string that defines various arguments to the kernel
  - It is very important for system configuration
  - `root=` for the root filesystem (covered later)
  - `console=` for the destination of kernel messages
  - and many more, documented in `Documentation/kernel-parameters.txt` in the kernel sources
- This kernel command line is either
  - Passed by the bootloader. In U-Boot, the contents of the `bootargs` environment variable is automatically passed to the kernel
  - Built into the kernel, using the `CONFIG_CMDLINE` option.

Practical lab - Module Development Environment

- Set up a cross-compiling environment
- Cross-compile a kernel for an ARM target platform
- Boot this kernel from a directory on your workstation, accessed by the board through NFS

Using kernel modules

Advantages of modules

- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).
- Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later.
- Caution: once loaded, have full control and privileges in the system. No particular protection. That's why only the `root` user can load and unload modules.

Module dependencies

- Some kernel modules can depend on other modules, which need to be loaded first.
- Example: the `usb-storage` module depends on the `scsi_mod`, `libusual` and `usbcore` modules.
- Dependencies are described in `/lib/modules/<kernel-version>/modules.dep`
  This file is generated when you run `make modules_install`.

Kernel log

When a new module is loaded, related information is available in the kernel log.

- The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)
- Kernel log messages are available through the `dmesg` command (**d**iagnostic **mes**sa**g**e)
- Kernel log messages are also displayed in the system console (console messages can be filtered by level using the `loglevel` kernel parameter, or completely disabled with the `quiet` parameter).
- Note that you can write to the kernel log from userspace too:
  `echo "Debug info" > /dev/kmsg`

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. The full path to the module object file must be given.

Understanding module loading issues

- When loading a module fails, `insmod` often doesn't give you enough details!
- Details are often 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 (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. `modprobe` automatically looks in `/lib/modules/<version>/` for the object file corresponding to the given module name.
- `lsmod`
  Displays the list of loaded modules
  Compare its output with the contents of `/proc/modules`!

Module utilities (3)

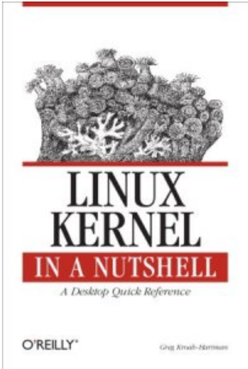
- `sudo rmmod <module_name>`
  Tries to remove the given module.
  Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)
- `sudo modprobe -r <module_name>`
  Tries to remove the given module and all dependent modules (which are no longer needed after removing the module)

Passing parameters to modules

- Find available parameters:
  `modinfo snd-intel8x0m`
- Through `insmod`:
  `sudo insmod ./snd-intel8x0m.ko index=-2`
- Through `modprobe`:
  Set parameters in `/etc/modprobe.conf` or in any file in `/etc/modprobe.d/`:
  `options snd-intel8x0m index=-2`
- Through the kernel command line, when the driver is built statically into the kernel:
  `snd-intel8x0m.index=-2`
  - `snd-intel8x0m` is the *driver name*
  - `index` is the *driver parameter name*
  - `-2` is the *driver parameter value*

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/
- Our rating: 2 stars



Embedded Linux driver development

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Corrections, suggestions, contributions and translations are welcome!

Loadable Kernel Modules

Hello Module 1/2

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

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

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

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

Hello Module 2/2

- `__init`
  - removed after initialization (static kernel or module.)
- `__exit`
  - discarded when module compiled statically into the kernel.
- Example available on `http://free-electrons.com/doc/c/hello.c`

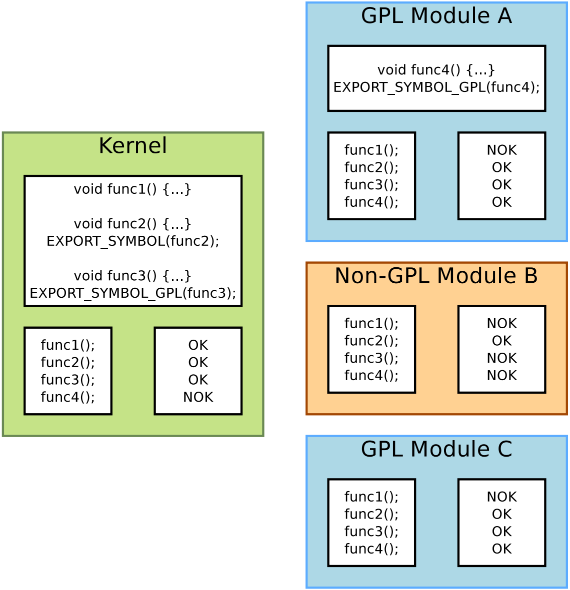
Hello Module Explanations

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

Symbols Exported to Modules 1/2

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

Symbols exported to modules 2/2



Module License

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

Compiling a Module

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

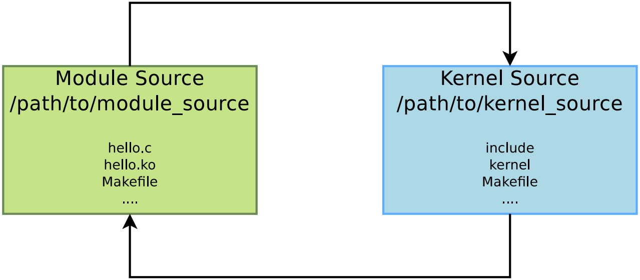
Compiling an out-of-tree Module 1/2

- The below Makefile should be reusable for any single-file out-of-tree Linux module
- The source file is hello.c
- Just run make to build the hello.ko file

```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources

all:
<tab>$(MAKE) -C $(KDIR) M=`pwd` modules
endif
```

- For KDIR, you can either set
  - full kernel source directory (configured and compiled)
  - or just kernel headers directory (minimum needed)

Compiling an out-of-tree Module 2/2



- The module Makefile is interpreted with `KERNELRELEASE` undefined, so it calls the kernel Makefile, passing the module directory in the `M` variable
- the kernel Makefile knows how to compile a module, and thanks to the `M` variable, knows where the Makefile for our module is. The module Makefile is interpreted with `KERNELRELEASE` defined, so the kernel sees the `obj-m` definition.

Modules and Kernel Version

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

New Driver in Kernel Sources 1/2

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

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

New Driver in Kernel Sources 2/2

- Add a line in the `Makefile` file based on the `Kconfig` setting:
  `obj-$(CONFIG_USB_SERIAL_NAVMAN) += navman.o`
- It tells the kernel build system to build `navman.c` when the `USB_SERIAL_NAVMAN` option is enabled. It works both if compiled statically or as a module.
  - Run `make xconfig` and see your new options!
  - Run `make` and your new files are compiled!
  - See `Documentation/kbuild/` for details and more elaborate examples like drivers with several source files, or drivers in their own subdirectory, etc.

How To Create Linux Patches

- The old school way
  - Before making your changes, make sure you have two kernel trees: `cp -a linux-3.5.5/ linux-3.5.5-patch/`
  - Make your changes in `linux-3.5.5-patch/`
  - Run `make distclean` to keep only source files.
  - Create a patch file: `diff -Nur linux-3.5.5/ linux-3.5.5-patch/ > patchfile`
  - Not convenient, does not scale to multiple patches
- The new school ways
  - Use `quilt` (tool to manage a stack of patches)
  - Use `git` (revision control system used by the Linux kernel developers)

Hello Module with Parameters 1/2

```
/* 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);
```

Hello Module with Parameters 2/2

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

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

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

Thanks to Jonathan Corbet for the example!

Example available on

`http://free-electrons.com/doc/c/hello_param.c`

Declaring a module parameter

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

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

/* Example */
int irq=5;
module_param(irq, int, S_IRUGO);
```

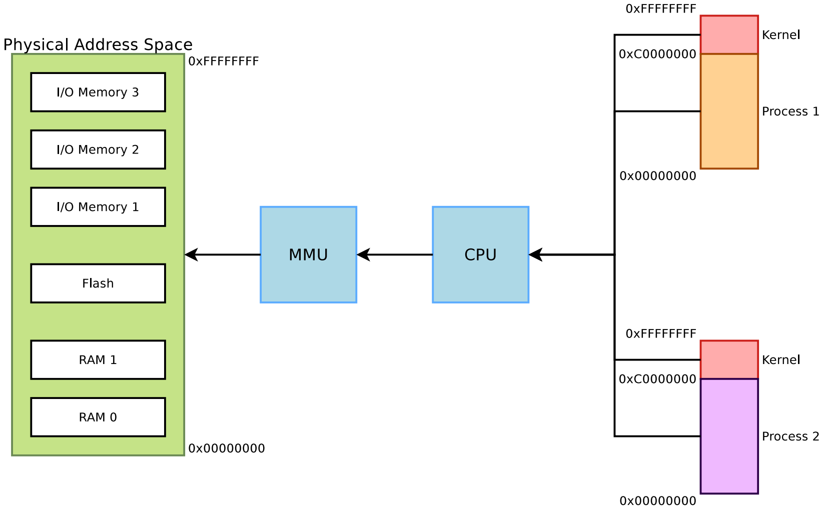
Modules parameter arrays are also possible with `module_param_array()`, but they are less common.

Practical lab - Writing Modules

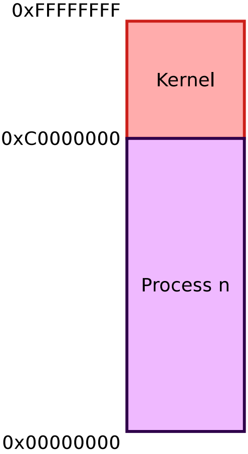
- Create, compile and load your first module
- Add module parameters
- Access kernel internals from your module

Memory Management

Physical and Virtual Memory

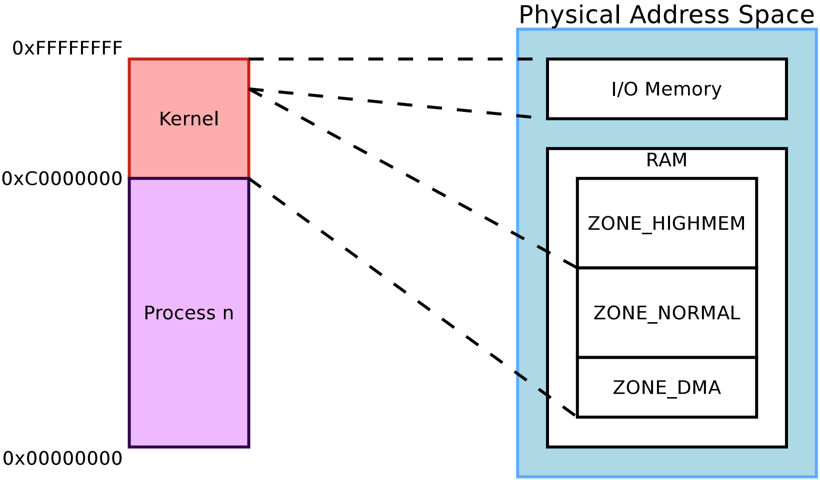


Virtual Memory Organization



- 1GB reserved for kernel-space
  - Contains kernel code and core data structures, identical in all address spaces
  - Most memory can be a direct mapping of physical memory at a fixed offset
- Complete 3GB exclusive mapping available for each user-space process
  - Process code and data (program, stack, ...)
  - Memory-mapped files
  - Not necessarily mapped to physical memory (demand fault paging used for dynamic mapping to physical memory pages)
  - Differs from one address space to another

Physical / virtual memory mapping



Accessing more physical memory

- Only less than 1GB memory addressable directly through kernel virtual address space
- If more physical memory is present on the platform, part of the memory will not be accessible by kernel space, but can be used by user-space
- To allow the kernel to access more physical memory:
  - Change 1GB/3GB memory split (2GB/2GB) (`CONFIG_VMSPLIT_3G`) ⇒ reduces total memory available for each process
  - Change for a 64 bit architecture ;-) See `Documentation/x86/x86_64/mm.txt` for an example.
  - Activate *highmem* support if available for your architecture:
    - Allows kernel to map parts of its non-directly accessible memory
    - Mapping must be requested explicitly
    - Limited addresses ranges reserved for this usage
- See `http://lwn.net/Articles/75174/` for useful explanations

Accessing even more physical memory!

- If your 32 bit platform hosts more than 4GB, they just cannot be mapped
- PAE (Physical Address Expansion) may be supported by your architecture
- Adds some address extension bits used to index memory areas
- Allows accessing up to 64 GB of physical memory through bigger pages (2 MB pages on x86 with PAE)
- Note that each user-space process is still limited to a 3 GB memory space

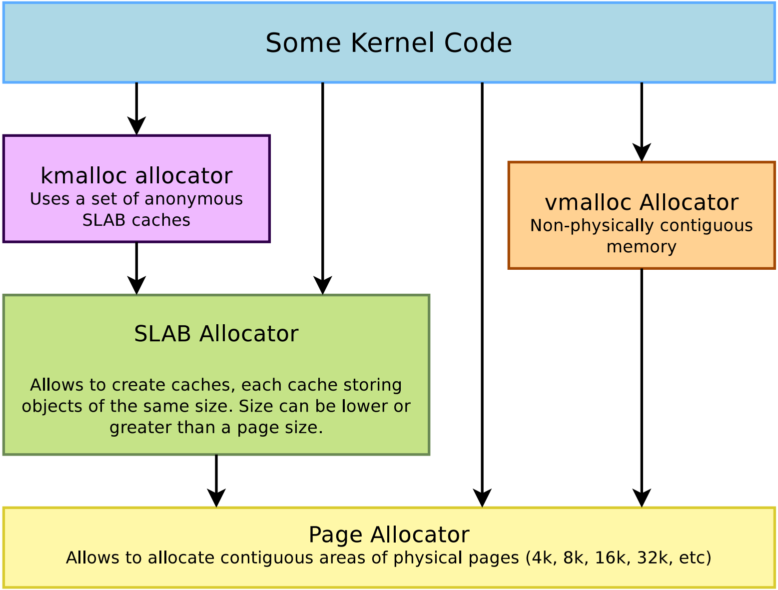
Notes on user-space memory

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

Back to kernel memory

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

Allocators in the Kernel



Page Allocator

- Appropriate for medium-size allocations
- A page is usually 4K, but can be made greater in some architectures (sh, mips: 4, 8, 16 or 64 KB, but not configurable in x86 or arm).
- Buddy allocator strategy, so only allocations of power of two number of pages are possible: 1 page, 2 pages, 4 pages, 8 pages, 16 pages, etc.
- Typical maximum size is 8192 KB, but it might depend on the kernel configuration.
- The allocated area is virtually contiguous (of course), but also physically contiguous. It is allocated in the identity-mapped part of the kernel memory space.
  - This means that large areas may not be available or hard to retrieve due to physical memory fragmentation.

Page Allocator API: Get free pages

- `unsigned long get_zeroed_page(int flags)`
  - Returns the virtual address of a free page, initialized to zero
- `unsigned long __get_free_page(int flags)`
  - Same, but doesn't initialize the contents
- `unsigned long __get_free_pages(int flags, unsigned int order)`
  - Returns the starting virtual address of an area of several contiguous pages in physical RAM, with order being `log2(number_of_pages)`.Can be computed from the size with the `get_order()` function.

Page Allocator API: Free Pages

- `void free_page(unsigned long addr)`
  - Frees one page.
- `void free_pages(unsigned long addr, unsigned int order)`
  - Frees multiple pages. Need to use the same order as in allocation.

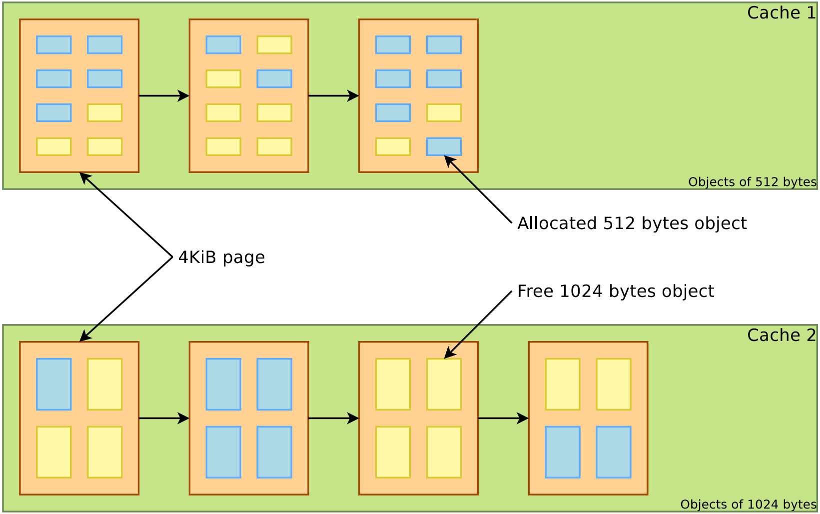
Page Allocator Flags

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

SLAB Allocator 1/2

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

SLAB Allocator 2/2



Different SLAB Allocators

- There are three different, but API compatible, implementations of a SLAB allocator in the Linux kernel. A particular implementation is chosen at configuration time.
  - SLAB: original, well proven allocator in Linux 2.6.
  - SLOB: much simpler. More space efficient but doesn't scale well. Saves a few hundreds of KB in small systems (depends on `CONFIG_EXPERT`)
  - SLUB: the new default allocator since 2.6.23, simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation.

kmalloc Allocator

- The kmalloc allocator is the general purpose memory allocator in the Linux kernel
- For small sizes, it relies on generic SLAB caches, named `kmalloc-XXX` in `/proc/slabinfo`
- For larger sizes, it relies on the page allocator
- The allocated area is guaranteed to be physically contiguous
- The allocated area size is rounded up to the next power of two size (while using the SLAB allocator directly allows to have more flexibility)
- It uses the same flags as the page allocator (`GFP_KERNEL`, `GFP_ATOMIC`, `GFP_DMA`, etc.) with the same semantics.
- Maximum sizes, on `x86` and `arm` (see `http://j.mp/YIGq6W`):
  - Per allocation: 4 MB
  - Total allocations: 128 MB
- Should be used as the primary allocator unless there is a strong reason to use another one.

kmalloc API 1/2

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

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

kmalloc API 2/2

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

vmalloc Allocator

- The `vmalloc` allocator can be used to obtain virtually contiguous memory zones, but not physically contiguous. The requested memory size is rounded up to the next page.
- The allocated area is in the kernel space part of the address space, but outside of the identically-mapped area
- Allocations of fairly large areas is possible (almost as big as total available memory, see `http://j.mp/YIGq6W` again), since physical memory fragmentation is not an issue, but areas cannot be used for DMA, as DMA usually requires physically contiguous buffers.
- API in `include/linux/vmalloc.h`
  - `void *vmalloc(unsigned long size);`
    - Returns a virtual address
  - `void vfree(void *addr);`

Kernel memory debugging

- Debugging features available since 2.6.31
  - `Kmemcheck`
    - Dynamic checker for access to uninitialized memory.
    - Only available on `x86` so far (Linux 3.6 status), but will help to improve architecture independent code anyway.
    - See `Documentation/kmemcheck.txt` for details.
  - `Kmemleak`
    - Dynamic checker for memory leaks
    - This feature is available for all architectures.
    - See `Documentation/kmemleak.txt` for details.
- Both have a significant overhead. Only use them in development!

Useful general-purpose kernel APIs

Memory/string utilities

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

Linked lists

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

Linked Lists Examples (1)

- From include/linux/atmel_tc.h

```
/*
 * Definition of a list element, with a
 * struct list_head member
 */
struct atmel_tc
{
    /* some members */
    struct list_head node;
};
```

Linked Lists Examples (2)

- From drivers/misc/atmel_tclib.c

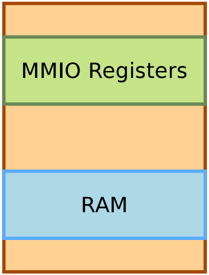
```
/* Define the global list */
static LIST_HEAD(tc_list);

static int __init tc_probe(struct platform_device *pdev) {
    struct atmel_tc *tc;
    tc = kzalloc(sizeof(struct atmel_tc), GFP_KERNEL);
    /* Add an element to the list */
    list_add_tail(&tc->node, &tc_list);
}

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

I/O Memory and Ports

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

- MMIO
  - Same address bus to address memory and I/O devices
  - Access to the I/O devices using regular instructions
  - Most widely used I/O method across the different architectures supported by Linux
- PIO
  - Different address spaces for memory and I/O devices
  - Uses a special class of CPU instructions to access I/O devices
  - Example on x86: IN and OUT instructions

MMIO vs PIO



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

Separate I/O address space,
accessed with specific instructions

Requesting I/O ports

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

/proc/ioports example (x86)

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
03f6-03f6 : ide0
03f8-03ff : serial
0800-087f : 0000:00:1f.0
...
```

Accessing I/O ports

- Functions to read/write bytes (`b`), word (`w`) and longs (`l`) to I/O ports:
  - `unsigned in[bwl](unsigned port)`
  - `void out[bwl](value, unsigned long port)`
- And the strings variants: often more efficient than the corresponding C loop, if the processor supports such operations!
  - `void ins[bwl](unsigned port, void *addr, unsigned long count)`
  - `void outs[bwl](unsigned port, void *addr, unsigned long count)`
- Examples
  - read 8 bits
    - `oldlcr = inb(baseio + UART_LCR)`
  - write 8 bits
    - `outb(MOXA_MUST_ENTER_ENCHANCE, baseio + UART_LCR)`

Requesting I/O memory

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

/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
40400000-407fffff : PCI CardBus #03
40800000-40bfffff : PCI CardBus #03
a0000000-a0000fff : pcmcia_socket0
e8000000-efffffff : PCI Bus #01
...
```

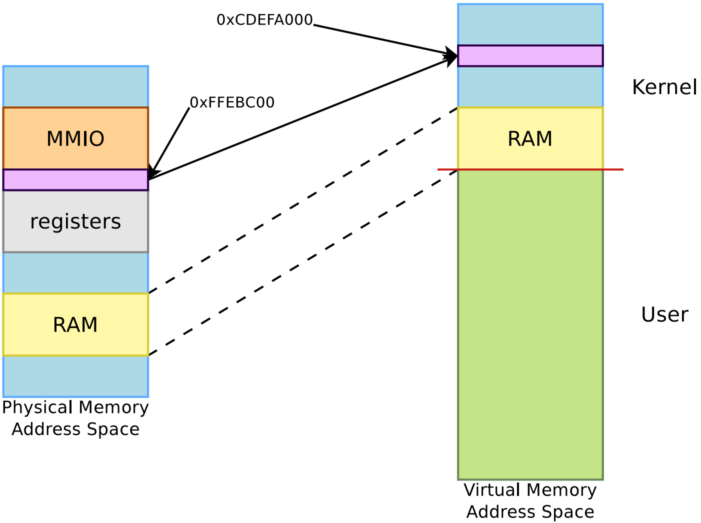
Mapping I/O memory in virtual memory

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

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

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

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

ioremap()




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

Accessing MMIO devices

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

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

- To do raw access, without endianness conversion

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

- Example
  - 32 bits write

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

New API for mixed accesses

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

Avoiding I/O access issues

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

/dev/mem

- Used to provide user-space applications with direct access to physical addresses.
- Usage: open `/dev/mem` and read or write at given offset. What you read or write is the value at the corresponding physical address.
- Used by applications such as the X server to write directly to device memory.
- On `x86`, `arm`, `tile`, `powerpc`, `unicore32`, `s390`: `CONFIG_STRICT_DEVMEM` option to restrict `/dev/mem` non-RAM addresses, for security reasons (Linux 3.6 status).

Practical lab - I/O Memory and Ports

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

Device Files

Devices

- One of the kernel important role is to **allow applications to access hardware devices**
- In the Linux kernel, most devices are presented to userspace applications through two different abstractions
  - **Character** device
  - **Block** device
- Internally, the kernel identifies each device by a triplet of information
  - **Type** (character or block)
  - **Major** (typically the category of device)
  - **Minor** (typically the identifier of the device)

Types of devices

- Block devices
  - A device composed of fixed-sized blocks, that can be read and written to store data
  - Used for hard disks, USB keys, SD cards, etc.
- Character devices
  - Originally, an infinite stream of bytes, with no beginning, no end, no size. The pure example: a serial port.
  - Used for serial ports, terminals, but also sound cards, video acquisition devices, frame buffers
  - Most of the devices that are not block devices are represented as character devices by the Linux kernel

Devices: everything is a file

- A very important Unix design decision was to represent most of the “system objects” as files
- It allows applications to manipulate all “system objects” with the normal file API (`open`, `read`, `write`, `close`, etc.)
- So, devices had to be represented as files to the applications
- This is done through a special artifact called a **device file**
- It is a special type of file, that associates a file name visible to userspace applications to the triplet *(type, major, minor)* that the kernel understands
- All *device files* are by convention stored in the `/dev` directory

Device files examples

Example of device files in a Linux system

```
$ ls -l /dev/ttyS0 /dev/tty1 /dev/sda1 /dev/sda2 /dev/zero
brw-rw---- 1 root disk    8,  1 2011-05-27 08:56 /dev/sda1
brw-rw---- 1 root disk    8,  2 2011-05-27 08:56 /dev/sda2
crw------- 1 root root    4,  1 2011-05-27 08:57 /dev/tty1
crw-rw---- 1 root dialout 4, 64 2011-05-27 08:56 /dev/ttyS0
crw-rw-rw- 1 root root    1,  5 2011-05-27 08:56 /dev/zero
```

Example C code that uses the usual file API to write data to a serial port

```c
int fd;
fd = open("/dev/ttyS0", O_RDWR);
write(fd, "Hello", 5);
close(fd);
```

Creating device files

- On a basic Linux system, the device files have to be created manually using the `mknod` command
  - `mknod /dev/<device> [c|b] major minor`
  - Needs root privileges
  - Coherency between device files and devices handled by the kernel is left to the system developer
- On more elaborate Linux systems, mechanisms can be added to create/remove them automatically when devices appear and disappear
  - `devtmpfs` virtual filesystem, since kernel 2.6.32
  - `udev` daemon, solution used by desktop and server Linux systems
  - `mdev` program, a lighter solution than `udev`

Character drivers

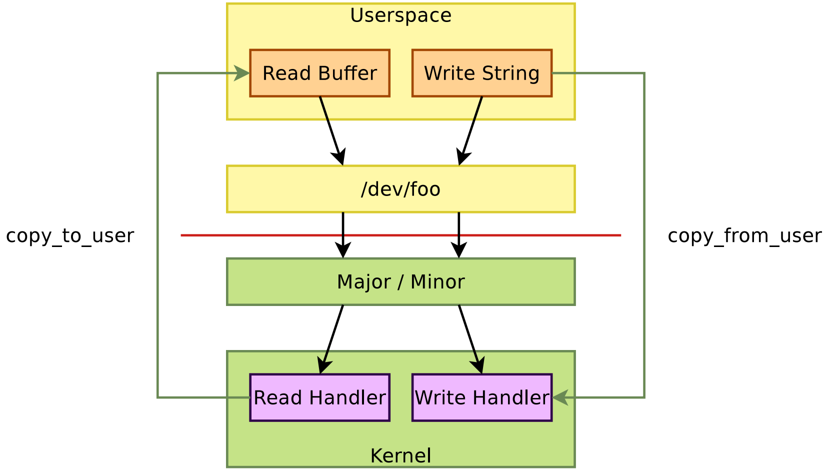
Usefulness of character drivers

- 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.

Creating a Character Driver 1/2

- 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.

Creating a Character Driver 2/2



Implementing a character driver

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

File operations 1/3

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

File operations 2/3

- Here are the most important operations for a character driver. All of them are optional.

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

open() and release()

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

read()

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

write()

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

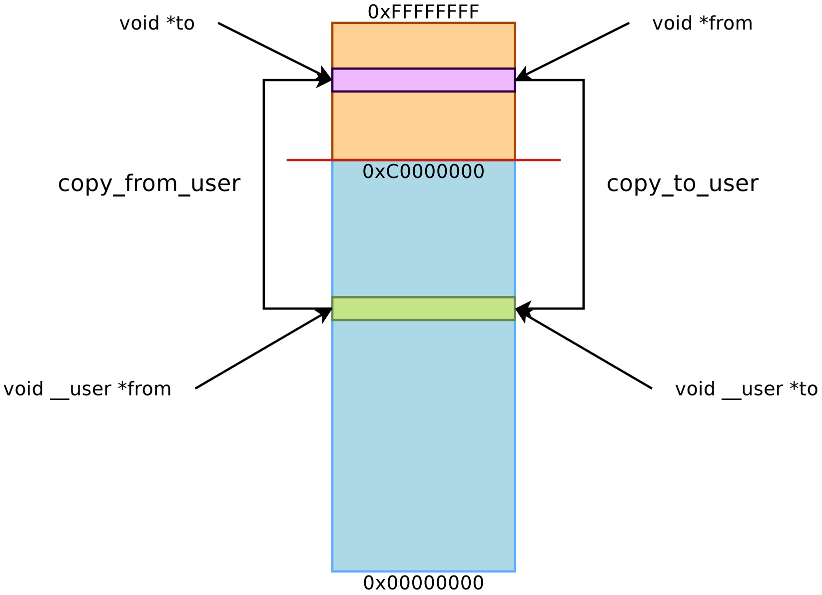
Exchanging data with user-space 1/3

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

Exchanging data with user-space 2/3

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

Exchanging data with user-space 3/3



Zero copy access to user memory

- Having to copy data to our from an intermediate kernel buffer is expensive.
- *Zero copy* options are possible:
  - `mmap()` system call to allow user space to directly access memory mapped I/O space (covered in the `mmap()` section).
  - `get_user_pages()` to get a mapping to user pages without having to copy them. See `http://j.mp/oPW6Fb` (Kernel API doc). This API is more complex to use though.

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_t(int, remaining_size, 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;
        }
}
```

Piece of code available at
`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;
        }
}
```

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

unlocked_ioctl()

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

ioctl() example: kernel side

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

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

    return 0; }
```

Selected excerpt from `drivers/misc/phantom.c`

Ioctl() Example: Application Side

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

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

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

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

    return 0;
}
```

File Operations Definition: Example 3/3

- Defining a `file_operations` structure:

```
#include <linux/fs.h>
static struct file_operations acme_fops =
{
    .owner = THIS_MODULE,
    .read = acme_read,
    .write = acme_write,
};
```

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

dev_t data type

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

Registering device numbers 1/2

```
#include <linux/fs.h>
int register_chrdev_region(
        dev_t from,         /* Starting device number */
        unsigned count,     /* Number of device numbers */
        const char *name); /* Registered name */
```

Returns 0 if the allocation was successful.

Example

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

if (register_chrdev_region(acme_dev, acme_count, "acme")) {
        pr_err("Failed to allocate device number\n");
...
```

Registering device numbers 2/2

- If you don't have fixed device numbers assigned to your driver
  - Better not to choose arbitrary ones. There could be conflicts with other drivers.
  - The kernel API offers an `alloc_chrdev_region` function to have the kernel allocate free ones for you. You can find the allocated major number in `/proc/devices`.

Information on registered devices: /proc/devices

```
Character devices:
  1 mem
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
...

Block devices:
  1 ramdisk
  7 loop
  8 sd
  9 md
 11 sr
179 mmc
254 mdp
```

Character device registration 1/2

- 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 2/2

- 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 */
    unsigned count); /* Number of devices */
```

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

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

Character device unregistration

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

```
cdev_del(&acme_cdev);
unregister_chrdev_region(acme_dev, acme_count);
```

Linux error codes

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

Char driver example summary 1/4

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

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

static struct cdev acme_cdev;

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

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

Char driver example summary 2/4

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

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

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

        cdev_init(&acme_cdev, &acme_fops);

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

        return 0;

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

Character Driver Example Summary 3/4

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

module_init(acme_init);
module_exit(acme_exit);
```

Character Driver Example Summary 4/4

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

Practical lab - Character Drivers

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

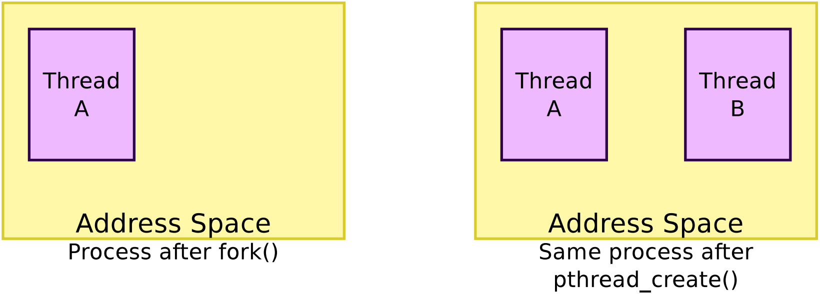
Processes and scheduling

Process, thread?

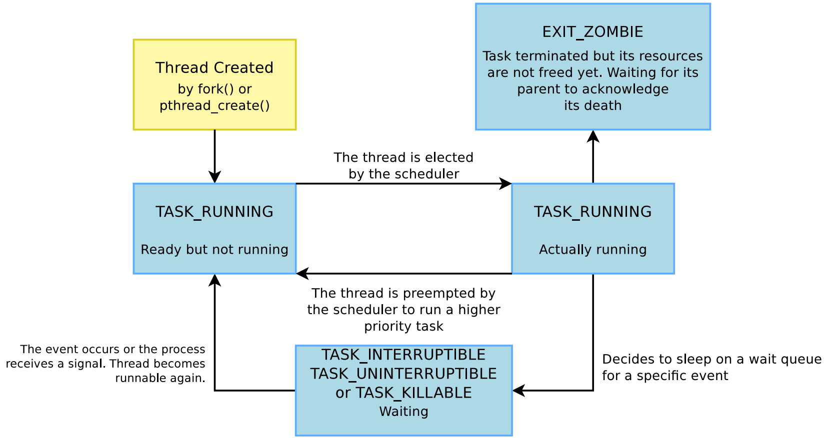
- Confusion about the terms *process*, *thread* and *task*
- In Unix, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - One thread, that starts executing the `main()` function.
  - Upon creation, a process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

Process, thread: kernel point of view

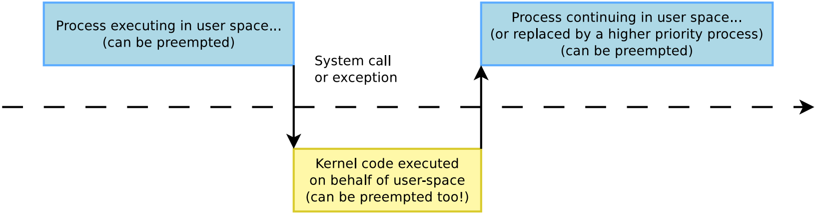
- The kernel represents each thread running in the system by a structure of type `task_struct`
- From a scheduling point of view, it makes no difference between the initial thread of a process and all additional threads created dynamically using `pthread_create()`



A thread life



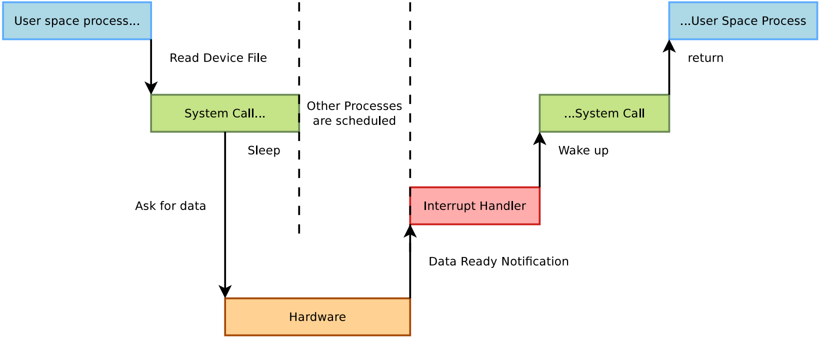
Execution of system calls



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

Sleeping

Sleeping



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

How to sleep 1/3

- Must declare a wait queue
- A wait queue will be used to store the list of threads waiting for an event
  - Static queue declaration
    - useful to declare as a global variable
    - `DECLARE_WAIT_QUEUE_HEAD(module_queue);`
  - Or dynamic queue declaration
    - Useful to embed the wait queue inside another data structure

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

How to sleep 2/3

- Several ways to make a kernel process sleep
  - `void wait_event(queue, condition);`
    - Sleeps until the task is woken up and the given C expression is true. Caution: can't be interrupted (can't kill the user-space process!)
  - `int wait_event_killable(queue, condition);`
    - Can be interrupted, but only by a *fatal* signal (`SIGKILL`). Returns `-ERESTARSYS` if interrupted.
  - `int wait_event_interruptible(queue, condition);`
    - Can be interrupted by any signal. Returns `-ERESTARTSYS` if interrupted.

How to sleep 3/3

- `int wait_event_timeout(queue, condition, timeout);`
  - Also stops sleeping when the task is woken up and the timeout expired. Returns `0` if the timeout elapsed, non-zero if the condition was met.
- `int wait_event_interruptible_timeout(queue, condition, timeout);`
  - Same as above, interruptible. Returns `0` if the timeout elapsed, `-ERESTARTSYS` if interrupted, positive value if the condition was met.

How to Sleep - Example

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

if (ret)
        return ret;
```

Waking up!

- Typically done by interrupt handlers when data sleeping processes are waiting for becomes available.
  - `wake_up(&queue);`
    - Wakes up all processes in the wait queue
  - `wake_up_interruptible(&queue);`
    - Wakes up all processes waiting in an interruptible sleep on the given queue

Exclusive vs. non-exclusive

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

Sleeping and Waking up - Implementation 1/2

- The scheduler doesn't keep evaluating the sleeping condition!

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

Sleeping and Waking up - Implementation 2/2

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

Interrupt Management

Registering an interrupt handler 1/2

- Defined in `include/linux/interrupt.h`
  - `int request_irq(unsigned int irq, irq_handler_t handler, unsigned long irq_flags, const char *devname, void *dev_id);`
    - `irq` is the requested IRQ channel
    - `handler` is a pointer to the IRQ handler
    - `irq_flags` are option masks (see next slide)
    - `devname` is the registered name
    - `dev_id` is a pointer to some data. It cannot be NULL as it is used as an identifier for `free_irq` when using shared IRQs.
  - `void free_irq(unsigned int irq, void *dev_id);`

Registering an interrupt handler 2/2

- Main `irq_flags` bit values (can be combined, none is fine too)
  - `IRQF_SHARED`
    - 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`
    - Use the IRQ arrival time to feed the kernel random number generator.

Interrupt handler constraints

- No guarantee in which address space the system will be in when the interrupt occurs: can't transfer data to and from user space
- Interrupt handler execution is managed by the CPU, not by the scheduler. Handlers can't run actions that may sleep, because there is nothing to resume their execution. In particular, need to allocate memory with `GFP_ATOMIC`.
- Interrupt handlers are run with all interrupts disabled (since 2.6.36). Therefore, they have to complete their job quickly enough, to avoiding blocking interrupts for too long.

/proc/interrupts on a Panda board

```
        CPU0    CPU1
39:     4       0       GIC     TWL6030-PIH
41:     0       0       GIC     l3-dbg-irq
42:     0       0       GIC     l3-app-irq
43:     0       0       GIC     prcm
44:     20294   0       GIC     DMA
52:     0       0       GIC     gpmc
...
IPI0:   0       0       Timer broadcast interrupts
IPI1:   23095   25663   Rescheduling interrupts
IPI2:   0       0       Function call interrupts
IPI3:   231     173     Single function call interrupts
IPI4:   0       0       CPU stop interrupts
LOC:    196407  136995  Local timer interrupts
Err:    0
```

Interrupt handler prototype

- `irqreturn_t foo_interrupt(int irq, void *dev_id)`
  - `irq`, the IRQ number
  - `dev_id`, the opaque pointer that was passed to `request_irq()`
- Return value
  - `IRQ_HANDLED`: recognized and handled interrupt
  - `IRQ_NONE`: not on a device managed by the module. Useful to share interrupt channels and/or report spurious interrupts to the kernel.

Typical interrupt handler's job

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

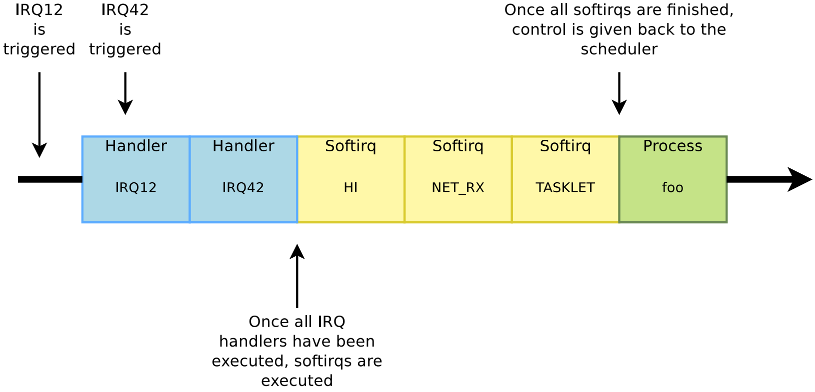
Threaded interrupts

- In 2.6.30, support for threaded interrupts has been added to the Linux kernel
  - The interrupt handler is executed inside a thread.
  - Allows to block during the interrupt handler, which is often needed for I2C/SPI devices as the interrupt handler needs to communicate with them.
  - Allows to set a priority for the interrupt handler execution, which is useful for real-time usage of Linux
- ```
  int request_threaded_irq(unsigned int irq,
  irq_handler_t handler, irq_handler_t thread_fn,
  unsigned long flags, const char *name, void *dev);
  ```
  - `handler`, “hard IRQ” handler
  - `thread_fn`, executed in a thread
  ```

Top half and bottom half processing

- Splitting the execution of interrupt handlers in 2 parts
  - Top half
    - This is the real interrupt handler, which should complete as quickly as possible since all interrupts are disabled. If possible, take the data out of the device and schedule a bottom half to handle it.
  - Bottom half
    - Is the general Linux name for various mechanisms which allow to postpone the handling of interrupt-related work. Implemented in Linux as softirqs, tasklets or workqueues.

Top half and bottom half diagram



Softirqs

- Softirqs are a form of bottom half processing
- The softirqs handlers are executed with all interrupts enabled, and a given softirq handler can run simultaneously on multiple CPUs
- They are executed once all interrupt handlers have completed, before the kernel resumes scheduling processes, so sleeping is not allowed.
- The number of softirqs is fixed in the system, so softirqs are not directly used by drivers, but by complete kernel subsystems (network, etc.)
- The list of softirqs is defined in `include/linux/interrupt.h`: `HI`, `TIMER`, `NET_TX`, `NET_RX`, `BLOCK`, `BLOCK_IOPOLL`, `TASKLET`, `SCHED`, `HRTIMER`, `RCU`
- The `HI` and `TASKLET` softirqs are used to execute tasklets

Tasklets

- Tasklets are executed within the `HI` and `TASKLET` softirqs. They are executed with all interrupts enabled, but a given tasklet is guaranteed to execute on a single CPU at a time.
- A tasklet can be declared statically with the `DECLARE_TASKLET()` macro or dynamically with the `tasklet_init()` function. A tasklet is simply implemented as a function. Tasklets can easily be used by individual device drivers, as opposed to softirqs.
- The interrupt handler can schedule the execution of a tasklet with
  - `tasklet_schedule()` to get it executed in the `TASKLET` softirq
  - `tasklet_hi_schedule()` to get it executed in the `HI` softirq (higher priority)

Tasklet Example: simplified atmel_serial.c 1/2

```
/* The tasklet function */
static void atmel_tasklet_func(unsigned long data) {
        struct uart_port *port = (struct uart_port *)data;
        [...]
}

/* Registering the tasklet */
init function(...) {
        [...]
        tasklet_init(&atmel_port->tasklet,
            atmel_tasklet_func,(unsigned long)port);
        [...]
}
```

Tasklet Example: simplified atmel_serial.c 2/2

```
/* Removing the tasklet */
cleanup function(...) {
        [...]
        tasklet_kill(&atmel_port->tasklet);
        [...]
}

/* Triggering execution of the tasklet */
somewhere function(...) {
        tasklet_schedule(&atmel_port->tasklet);
}
```

Workqueues

- Workqueues are a general mechanism for deferring work. It is not limited in usage to handling interrupts.
- The function registered as workqueue is executed in a thread, which means:
  - All interrupts are enabled
  - Sleeping is allowed
- A workqueue is registered with `INIT_WORK` and typically triggered with `queue_work()`
- The complete API, in `include/linux/workqueue.h` provides many other possibilities (creating its own workqueue threads, etc.)

Interrupt management summary

- Device driver
  - When the device file is first opened, register an interrupt handler for the device's interrupt channel.
- Interrupt handler
  - Called when an interrupt is raised.
  - Acknowledge the interrupt
  - If needed, schedule a tasklet taking care of handling data. Otherwise, wake up processes waiting for the data.
- Tasklet
  - Process the data
  - Wake up processes waiting for the data
- Device driver
  - When the device is no longer opened by any process, unregister the interrupt handler.

Practical lab - Interrupts

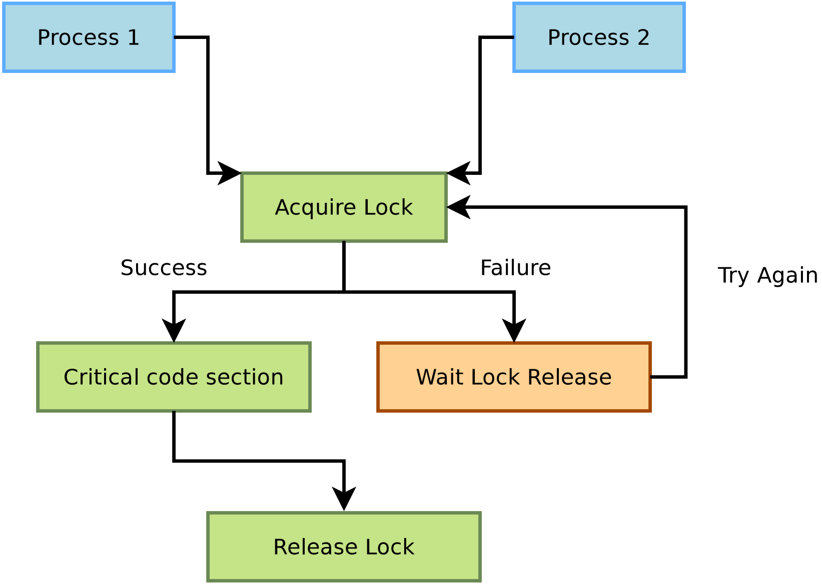
- Adding read capability to the character driver developed earlier.
- Register an interrupt handler.
- Waiting for data to be available in the read file operation.
- Waking up the code when data are available from the device.

Concurrent Access to Resources

Sources of concurrency issues

- In terms of concurrency, the kernel has the same constraint as a multi-threaded program: its state is global and visible in all executions contexts
- Concurrency arises because of
  - *Interrupts*, which interrupts the current thread to execute an interrupt handler. They may be using shared resources.
  - *Kernel preemption*, if enabled, causes the kernel to switch from the execution of one system call to another. They may be using shared resources.
  - *Multiprocessing*, in which case code is really executed in parallel on different processors, and they may be using shared resources as well.
- The solution is to keep as much local state as possible and for the shared resources, use locking.

Concurrency protection with locks



Linux mutexes

- The kernel's main locking primitive
- The process requesting the lock blocks when the lock is already held. Mutexes can therefore only be used in contexts where sleeping is allowed.
- Mutex definition:
  - `#include <linux/mutex.h>`
- Initializing a mutex statically:
  - `DEFINE_MUTEX(name);`
- Or initializing a mutex dynamically:
  - `void mutex_init(struct mutex *lock);`

Locking and Unlocking Mutexes 1/2

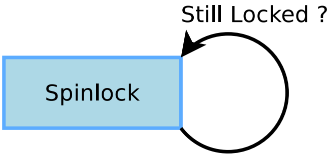
- `void mutex_lock(struct mutex *lock);`
  - Tries to lock the mutex, sleeps otherwise.
  - Caution: can't be interrupted, resulting in processes you cannot kill!
- `int mutex_lock_killable(struct mutex *lock);`
  - Same, but can be interrupted by a fatal (`SIGKILL`) signal. If interrupted, returns a non zero value and doesn't hold the lock. Test the return value!!!
- `int mutex_lock_interruptible(struct mutex *lock);`
  - Same, but can be interrupted by any signal.

Locking and Unlocking Mutexes 2/2

- `int mutex_trylock(struct mutex *lock);`
  - Never waits. Returns a non zero value if the mutex is not available.
- `int mutex_is_locked(struct mutex *lock);`
  - Just tells whether the mutex is locked or not.
- `void mutex_unlock(struct mutex *lock);`
  - Releases the lock. Do it as soon as you leave the critical section.

Spinlocks

- Locks to be used for code that is not allowed to sleep (interrupt handlers), or that doesn't want to sleep (critical sections). Be very careful not to call functions which can sleep!
- Originally intended for multiprocessor systems
- Spinlocks never sleep and keep spinning in a loop until the lock is available.
- Spinlocks cause kernel preemption to be disabled on the CPU executing them.
- The critical section protected by a spinlock is not allowed to sleep.



Initializing Spinlocks

- Statically
  - `DEFINE_SPINLOCK(my_lock);`
- Dynamically
  - `void spin_lock_init(spinlock_t *lock);`

Using Spinlocks 1/2

- Several variants, depending on where the spinlock is called:
  - `void spin_lock(spinlock_t *lock);`
  - `void spin_unlock(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(spinlock_t *lock, unsigned long flags);`
  - `void 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 2/2

- `void spin_lock_bh(spinlock_t *lock);`
- `void spin_unlock_bh(spinlock_t *lock);`
  - Disables software interrupts, but not hardware ones.
  - Useful to protect shared data accessed in process context and in a soft interrupt (*bottom half*).
  - No need to disable hardware interrupts in this case.
- Note that reader / writer spinlocks also exist.

Spinlock example

- Spinlock structure embedded into uart_port

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

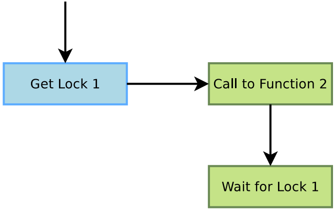
- Spinlock taken/released with protection against interrupts

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

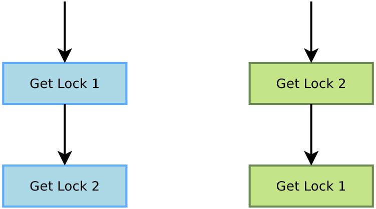
        spin_lock_irqsave(&port->lock, flags);
        /* Do something */
        spin_unlock_irqrestore(&port->lock, flags);
}
```

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 1/2

- Useful when the shared resource is an integer value
- Even an instruction like `n++` is not guaranteed to be atomic on all processors!
- Atomic operations definitions
  - `#include <asm/atomic.h>`
- `atomic_t`
  - Contains a signed integer (at least 24 bits)
- Atomic operations (main ones)
  - Set or read the counter:
    - `void 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);`

Atomic Variables 2/2

- Similar functions testing the result:
  - `int atomic_inc_and_test(...);`
  - `int atomic_dec_and_test(...);`
  - `int atomic_sub_and_test(...);`
- Functions returning the new value:
  - `int atomic_inc_return(...);`
  - `int atomic_dec_return(...);`
  - `int atomic_add_return(...);`
  - `int atomic_sub_return(...);`

Atomic Bit Operations

- Supply very fast, atomic operations
- On most platforms, apply to an unsigned long type.
- Apply to a void type on a few others.
- Set, clear, toggle a given bit:
  - `void set_bit(int nr, unsigned long * addr);`
  - `void clear_bit(int nr, unsigned long * addr);`
  - `void change_bit(int nr, unsigned long * addr);`
- Test bit value:
  - `int test_bit(int nr, unsigned long *addr);`
- Test and modify (return the previous value):
  - `int test_and_set_bit(...);`
  - `int test_and_clear_bit(...);`
  - `int test_and_change_bit(...);`

Practical lab - Locking

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

Debugging and tracing

Debugging Using Messages

- Three APIs are available
  - The old `printk()`, no longer recommended for new debugging messages
  - The `pr_*()` family of functions: `pr_emerg()`, `pr_alert()`, `pr_crit()`, `pr_err()`, `pr_warning()`, `pr_notice()`, `pr_info()`, `pr_cont()` and the special `pr_debug()`
    - They take a classic format string with arguments
    - defined in `include/linux/printk.h`
  - The `dev_*()` family of functions: `dev_emerg()`, `dev_alert()`, `dev_crit()`, `dev_err()`, `dev_warning()`, `dev_notice()`, `dev_info()` and the special `dev_dbg()`
    - They take a pointer to `struct device` as first argument (covered later), and then a format string with arguments
    - defined in `include/linux/device.h`
    - To be used in drivers integrated with the Linux device model

pr_debug() and dev_dbg()

- When the driver is compiled with `DEBUG` defined, all those messages are compiled and printed at the debug level. `DEBUG` can be defined by `#define DEBUG` at the beginning of the driver, or using `ccflags-$(CONFIG_DRIVER) += -DDEBUG` in the `Makefile`
- When the kernel is compiled with `CONFIG_DYNAMIC_DEBUG`, then those messages can dynamically be enabled on a per-file, per-module or per-message basis
  - See `Documentation/dynamic-debug-howto.txt` for details
  - Very powerful feature to only get the debug messages you're interested in.
- When `DEBUG` is not defined and `CONFIG_DYNAMIC_DEBUG` is not enabled, those messages are not compiled in.

Configuring The Priority

- Each message is associated to a priority, ranging from `0` for emergency to `7` for debug.
- All the messages, regardless of their priority, are stored in the kernel log ring buffer
  - Typically accessed using the `dmesg` command
- Some of the messages may appear on the console, depending on their priority and the configuration of
  - The `loglevel` kernel parameter, which defines the priority above which messages are displayed on the console. See `Documentation/kernel-parameters.txt` for details.
  - The value of `/proc/sys/kernel/printk`, which allows to change at runtime the priority above which messages are displayed on the console. See `Documentation/sysctl/kernel.txt` for details.

DebugFS

- A virtual filesystem to export debugging information to user-space.
  - Kernel configuration: `DEBUG_FS`
    - `Kernel hacking -> Debug Filesystem`
  - The debugging interface disappears when Debugfs is configured out.
  - You can mount it as follows:
    - `sudo mount -t debugfs none /sys/kernel/debug`
  - First described on `http://lwn.net/Articles/115405/`
  - API documented in the Linux Kernel Filesystem API:
    - `Documentation/DocBook/filesystems/`

DebugFS API

- Create a sub-directory for your driver:
  - `struct dentry *debugfs_create_dir(const char *name, struct dentry *parent);`
- Expose an integer as a file in DebugFS:
  - `struct dentry *debugfs_create_{u,x}{8,16,32} (const char *name, mode_t mode, struct dentry *parent, u8 *value);`
    - `u` for decimal representation
    - `x` for hexadecimal representation
- Expose a binary blob as a file in DebugFS:
  - `struct dentry *debugfs_create_blob(const char *name, mode_t mode, struct dentry *parent, struct debugfs_blob_wrapper *blob);`
- Also possible to support writable DebugFS files or customize the output using the more generic `debugfs_create_file()` function.

Deprecated Debugging Mechanisms

- Some additional debugging mechanisms, whose usage is now considered deprecated
  - Adding special `ioctl()` commands for debugging purposes. DebugFS is preferred.
  - Adding special entries in the `proc` filesystem. DebugFS is preferred.
  - Adding special entries in the `sysfs` filesystem. DebugFS is preferred.
  - Using `printk()`. The `pr_*()` and `dev_*()` functions are preferred.

Using Magic SysRq

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

kgdb - A Kernel Debugger

- The execution of the kernel is fully controlled by gdb from another machine, connected through a serial line.
- Can do almost everything, including inserting breakpoints in interrupt handlers.
- Feature supported for the most popular CPU architectures

Using kgdb 1/2

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

Using kgdb 2/2

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

Debugging with a JTAG Interface

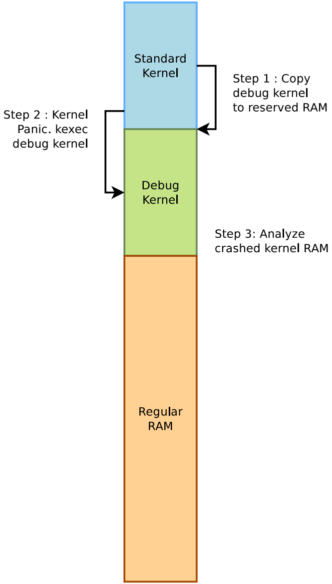
- Two types of JTAG dongles
  - Those offering a `gdb` compatible interface, over a serial port or an Ethernet connexion. `gdb` can directly connect to them.
  - Those not offering a gdb compatible interface are generally supported by OpenOCD (Open On Chip Debugger): `http://openocd.sourceforge.net/`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

More Kernel Debugging Tips

- Enable CONFIG_KALLSYMS_ALL
  - General Setup -
    > Configure standard kernel features
  - To get oops messages with symbol names instead of raw addresses
  - This obsoletes the ksymoops tool
- If your kernel doesn't boot yet or hangs without any message, you can activate the low-level debugging option (Kernel Hacking section, only available on arm and unicore32): CONFIG_DEBUG_LL=y
- Techniques to locate the C instruction which caused an oops
  - http://kerneltrap.org/node/3648

Kernel Crash Analysis with kexec/kdump

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



Tracing with SystemTap

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

SystemTap Script Example (1)

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

global kmalloc

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

# Exit after 10 seconds
probe timer.ms(10000) {
    exit()
}
probe end {
    foreach ([name] in kmalloc) {
        printf("Allocations for %s\n", name)
        printf("Count:     %d allocations\n", @count(kmalloc[name]))
        printf("Sum:       %d Kbytes\n", @sum(kmalloc[name])/1024)
        printf("Average:   %d bytes\n", @avg(kmalloc[name]))
        printf("Min:       %d bytes\n", @min(kmalloc[name]))
        printf("Max:       %d bytes\n", @max(kmalloc[name]))
        print("\nAllocations by size in bytes\n")
        print(@hist_log(kmalloc[name]))
        printf("-------------------------------------------\n\n")
    }
}
```

SystemTap Script Example (2)

```
#! /usr/bin/env stap
# Logs each file read performed by each process

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

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

Kernel Markers

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

LTTng

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

LTTV

- Viewer for LTTng traces
  - Support for huge traces (tested with 15 GB ones)
  - Can combine multiple tracefiles in a single view.
  - Graphical or text interface
- See `http://lttng.org/files/lttv-doc/user_guide/`

Practical lab - Kernel debugging

- Use the dynamic printk feature.
- Add debugfs entries
- Load a broken driver and see it crash
- Analyze the error information dumped by the kernel.
- Disassemble the code and locate the exact C instruction which caused the failure.

mmap

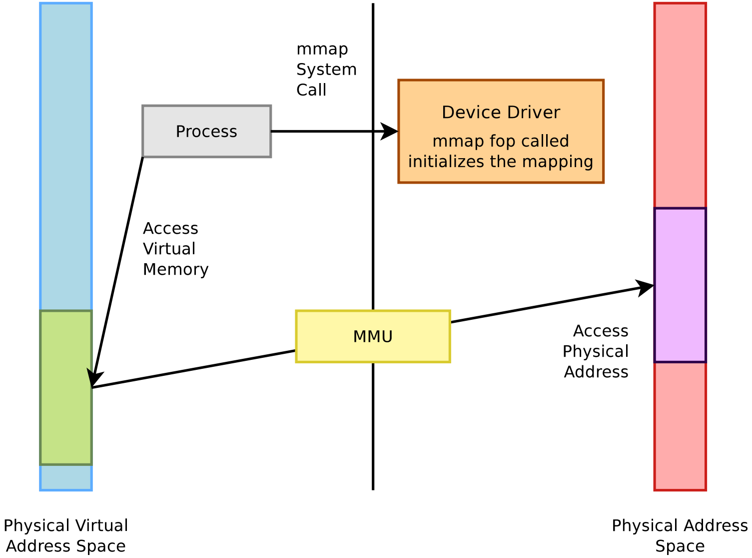
mmap

- Possibility to have parts of the virtual address space of a program mapped to the contents of a file
- 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
- One can access to current mapped files by two means:
  - `/proc/<pid>/maps`
  - `pmap <pid>`

/proc/<pid>/maps

```
      start-end          perm offset major:minor inode  mapped file name
...
7f4516d04000-7f4516d06000 rw-s 1152a2000 00:05 8406   /dev/dri/card0
7f4516d07000-7f4516d0b000 rw-s 120f9e000 00:05 8406   /dev/dri/card0
...
7f4518728000-7f451874f000 r-xp 00000000 08:01 268909  /lib/x86_64-linux-gnu/libexpat.so.1.5.2
7f451874f000-7f451894f000 ---p 00027000 08:01 268909  /lib/x86_64-linux-gnu/libexpat.so.1.5.2
7f451894f000-7f4518951000 r--p 00027000 08:01 268909  /lib/x86_64-linux-gnu/libexpat.so.1.5.2
7f4518951000-7f4518952000 rw-p 00029000 08:01 268909  /lib/x86_64-linux-gnu/libexpat.so.1.5.2
...
7f451da4f000-7f451dc3f000 r-xp 00000000 08:01 1549    /usr/bin/Xorg
7f451de3e000-7f451de41000 r--p 001ef000 08:01 1549    /usr/bin/Xorg
7f451de41000-7f451de4c000 rw-p 001f2000 08:01 1549    /usr/bin/Xorg
...
```

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 an mmap file operation and add it to the driver file operations:

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

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

remap_pfn_range()

- *pfn*: page frame number
- The most significant bits of the page address (without the bits corresponding to the page size).

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

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

Simple mmap Implementation

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

      if (size > ACME_SIZE)
          return -EINVAL;

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

      return 0;
    }
```

devmem2

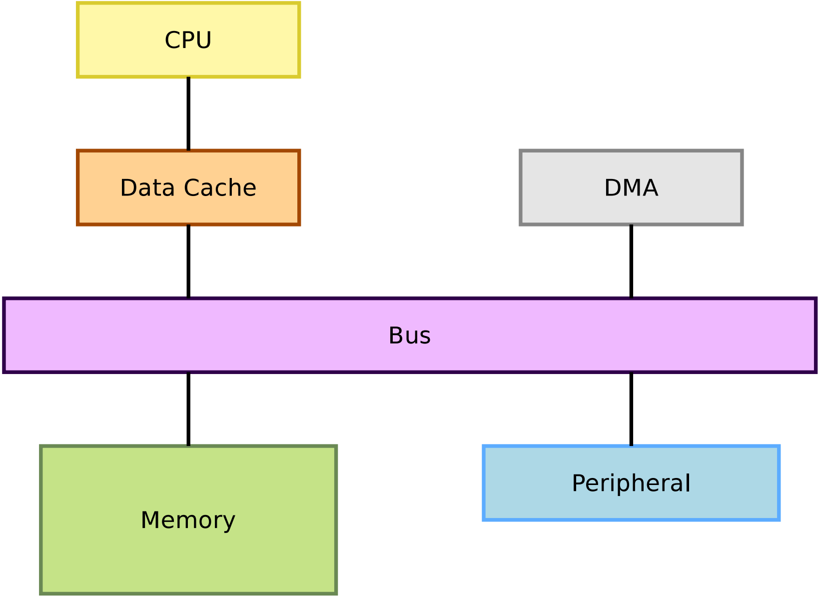
- `http://free-electrons.com/pub/mirror/devmem2.c`, by Jan-Derk Bakker
- Very useful tool to directly peek (read) or poke (write) I/O addresses mapped in physical address space from a shell command line!
  - Very useful for early interaction experiments with a device, without having to code and compile a driver.
  - Uses `mmap` to `/dev/mem`.
  - Examples (`b`: byte, `h`: half, `w`: word)
    - `devmem2 0x000c0004 h` (reading)
    - `devmem2 0x000c0008 w 0xffffffff` (writing)
  - `devmem` is now available in BusyBox, making it even easier to use.

mmap Summary

- The device driver is loaded. It defines an `mmap` file operation.
- A user space process calls the `mmap` system call.
- The `mmap` file operation is called.
- It initializes the mapping using the device physical address.
- The process gets a starting address to read from and write to (depending on permissions).
- The MMU automatically takes care of converting the process virtual addresses into physical ones.
- Direct access to the hardware without any expensive `read` or `write` system calls

DMA

DMA Integration



Constraints with a DMA

- A DMA deals with physical addresses, so:
  - Programming a DMA requires retrieving a physical address at some point (virtual addresses are usually used)
  - The memory accessed by the DMA shall be physically contiguous
- The CPU can access memory through a data cache
  - Using the cache can be more efficient (faster accesses to the cache than the bus)
  - But the DMA does not access to the CPU cache, so one need to take care of cache coherency (cache content vs memory content)
  - Either flush or invalidate the cache lines corresponding to the buffer accessed by DMA and processor at strategic times

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 physical 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 */
);
```

  - You can also use `mem=` to reserve specific RAM areas for specific devices (DSP, video device...).
  - Panda board example:
    - `mem=456M@0x80000000`, `mem=512M@0xA0000000`

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 making 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.

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 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);
```

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 */
);

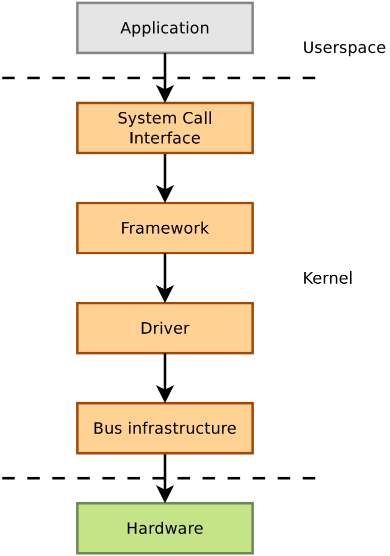
void dma_unmap_single(struct device *dev, dma_addr_t handdle,
    size_t size, enum dma_data_direction dir);
```

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.

Kernel Architecture for Device Drivers

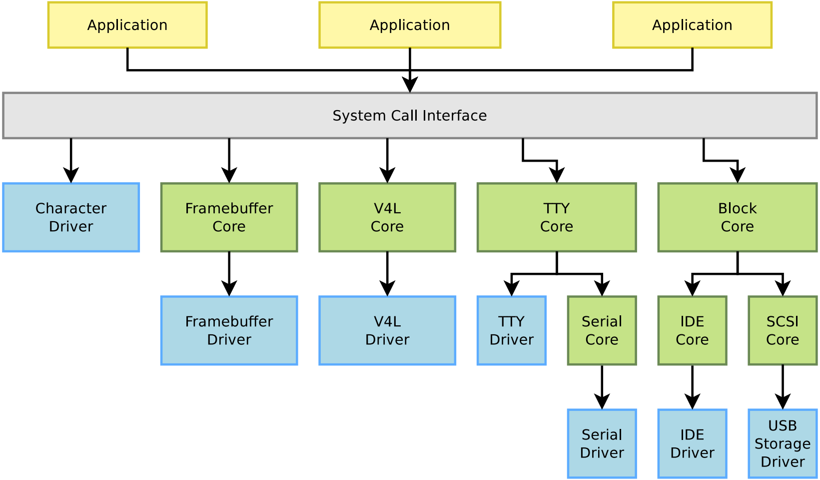
Kernel and Device Drivers



Kernel and Device Drivers

- Many device drivers are not implemented directly as character drivers
- They are implemented under a *framework*, specific to a given device type (framebuffer, V4L, serial, etc.)
  - The framework allows to factorize the common parts of drivers for the same type of devices
  - From userspace, they are still seen as character devices by the applications
  - The framework allows to provide a coherent userspace interface (`ioctl`, etc.) for every type of device, regardless of the driver
- The device drivers rely on the *bus infrastructure* to enumerate the devices and communicate with them.

Kernel Frameworks



Example: Framebuffer Framework

- Kernel option `CONFIG_FB`
  - `menuconfig FB`
    - `tristate "Support for frame buffer devices"`
- Implemented in `drivers/video/`
  - `fb.c`, `fbmem.c`, `fbmon.c`, `fbcmap.c`, `fbsysfs.c`, `modedb.c`, `fbcvt.c`
- Implements a single character driver and defines the user/kernel API
  - First part of `include/linux/fb.h`
- Defines the set of operations a framebuffer driver must implement and helper functions for the drivers
  - `struct fb_ops`
  - Second part of `include/linux/fb.h` (in `ifdef __KERNEL__`)

Framebuffer Driver Skeleton

- Skeleton driver in `drivers/video/skeletonfb.c`
- Implements the set of framebuffer specific operations defined by the `struct fb_ops` structure

- `xxxfb_open()`
- `xxxfb_read()`
- `xxxfb_write()`
- `xxxfb_release()`
- `xxxfb_checkvar()`
- `xxxfb_setpar()`
- `xxxfb_setcolreg()`
- `xxxfb_blank()`
- `xxxfb_pan_display()`
- `xxxfb_fillrect()`
- `xxxfb_copyarea()`
- `xxxfb_imageblit()`
- `xxxfb_cursor()`
- `xxxfb_rotate()`
- `xxxfb_sync()`
- `xxxfb_ioctl()`
- `xxxfb_mmap()`

Framebuffer Driver Skeleton

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

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

Framebuffer Driver Skeleton

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

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

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

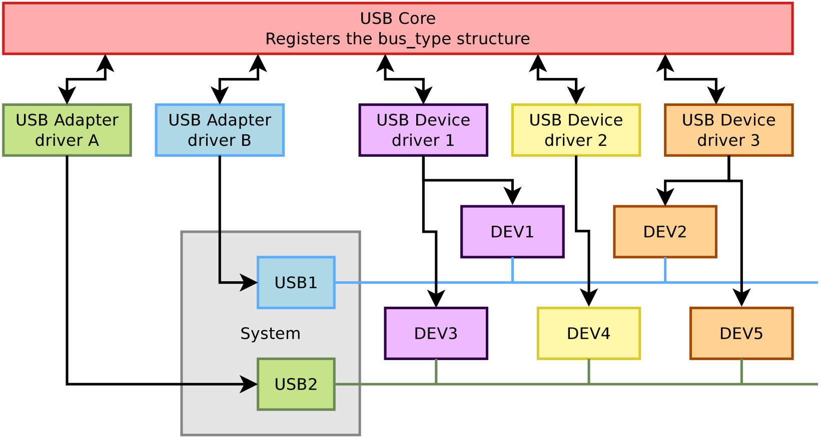
Unified Device Model

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

Bus Drivers

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

Example: USB Bus 1/2



Example: USB Bus 2/2

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

Example of Device Driver

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

Device Identifiers

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

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

Instantiation of usb_driver

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

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

Driver (Un)Registration

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

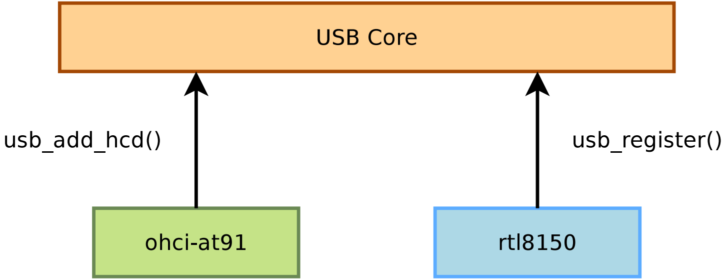
```
static int __init usb_rtl8150_init(void)
{
    return usb_register(&rtl8150_driver);
}

static void __exit usb_rtl8150_exit(void)
{
    usb_deregister(&rtl8150_driver);
}

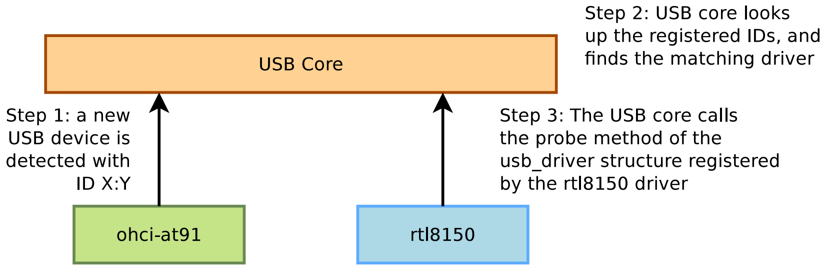
module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

At Initialization

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



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


Step 2: USB core looks up the registered IDs, and finds the matching driver
USB Core
Step 1: a new USB device is detected with ID X:Y
Step 3: The USB core calls the probe method of the usb_driver structure registered by the rtl8150 driver
ohci-at91
rtl8150

Probe Method

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

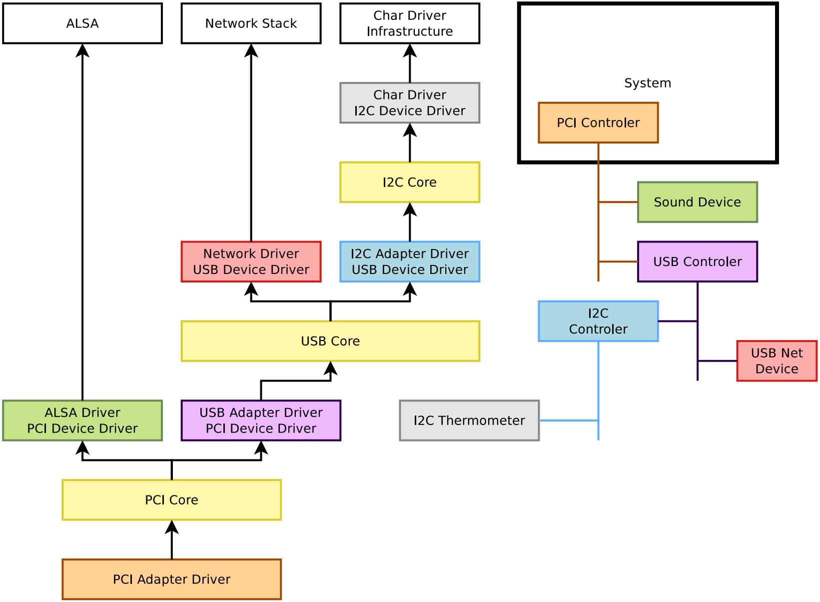
Probe Method Example

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

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

    return 0;
}
```

The Model is Recursive



sysfs

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

Platform Devices

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

Implementation of the Platform Driver

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

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

- And registers its driver to the platform driver infrastructure

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

static void __exit imx_serial_cleanup(void) {
    platform_driver_unregister(&serial_imx_driver);
}
```

Platform Device Instantiation 1/2

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

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

Platform device instantiation 2/2

- The device is part of a list

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

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

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

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

The Resource Mechanism

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

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

Using Resources

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

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

platform_data Mechanism

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

platform_data example 1/2

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

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

- The MX1ADS board code instantiates such a structure

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

platform_data Example 2/2

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

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

- The driver can access the platform data:

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

Driver-specific Data Structure

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

Driver-specific Data Structure Examples

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

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

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

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

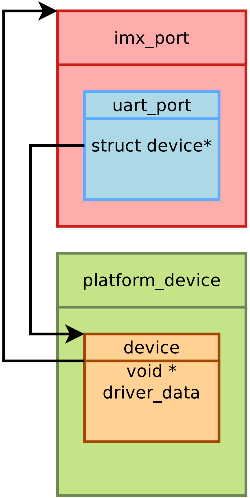
Link Between Structures 1/3

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

Link Between Structures 2/3

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

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



Link Between Structures 3/3

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

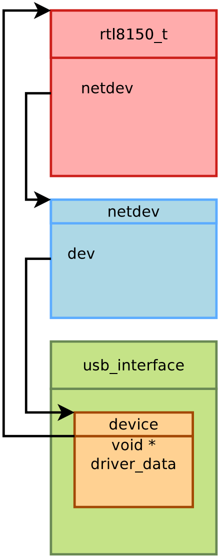
    netdev = alloc_etherdev(sizeof(rtl8150_t));
    dev = netdev_priv(netdev);

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

    [...]
}

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

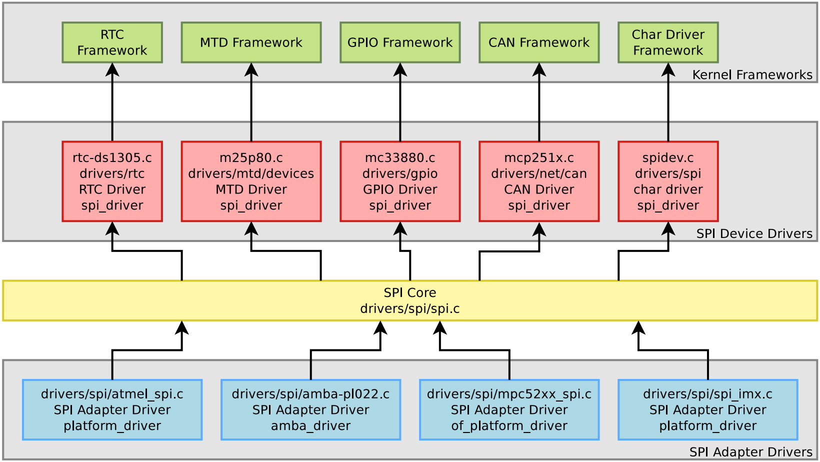
    [...]
}
```



Example of Another Non-Dynamic Bus: SPI

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

SPI Components



SPI AT91 SoC Code: at91sam9260_devices 1/2

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

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

SPI AT91 SoC Code: at91sam9260_devices 2/2

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

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

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

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

    [...]
}
```

AT91RM9200DK Board Code for SPI

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

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

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

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

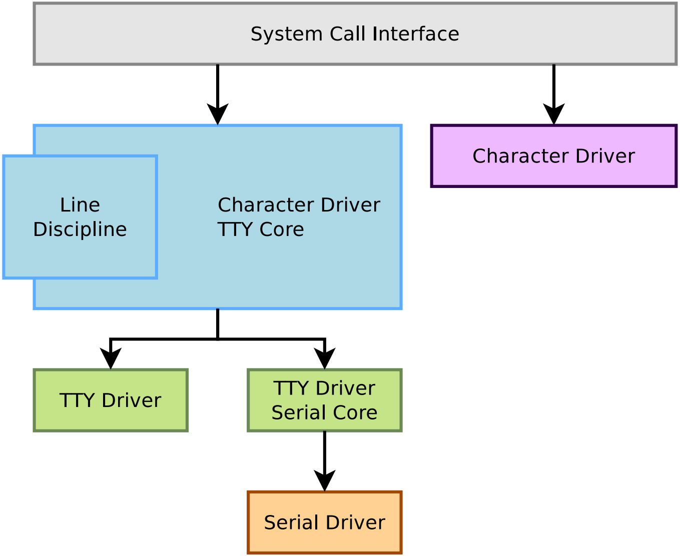
References

- Kernel documentation
  - `Documentation/driver-model/`
  - `Documentation/filesystems/sysfs.txt`
- The kernel source code
  - Full of examples of other drivers!

Serial Drivers

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Architecture (1)



Architecture (2)

- To be properly integrated in a Linux system, serial ports must be visible as TTY devices from userspace applications
- Therefore, the serial driver must be part of the kernel TTY subsystem
- Until 2.6, serial drivers were implemented directly behind the TTY core
  - A lot of complexity was involved
- Since 2.6, a specialized TTY driver, `serial_core`, eases the development of serial drivers
  - See `include/linux/serial_core.h` for the main definitions of the `serial_core` infrastructure
- The line discipline that cooks the data exchanged with the `tty` driver. For normal serial ports, `N_TTY` is used.

Data Structures

- A data structure representing a driver: `uart_driver`
  - Single instance for each driver
  - `uart_register_driver()` and `uart_unregister_driver()`
- A data structure representing a port: `uart_port`
  - One instance for each port (several per driver are possible)
  - `uart_add_one_port()` and `uart_remove_one_port()`
- A data structure containing the pointers to the operations: `uart_ops`
  - Linked from `uart_port` through the `ops` field

uart_driver

- Usually
  - Defined statically in the driver
  - Registered in `module_init()`
  - Unregistered in `module_cleanup()`
- Contains
  - `owner`, usually set to `THIS_MODULE`
  - `driver_name`
  - `dev_name`, the device name prefix, usually `ttyS`
  - `major` and `minor`
    - Use `TTY_MAJOR` and `64` to get the normal numbers. But they might conflict with the 8250-reserved numbers
  - `nr`, the maximum number of ports
  - `cons`, pointer to the console device (covered later)

uart_driver Code Example (1)

```
static struct uart_driver atmel_uart = {
    .owner = THIS_MODULE,
    .driver_name = "atmel_serial",
    .dev_name = ATMEL_DEVICENAME,
    .major = SERIAL_ATMEL_MAJOR,
    .minor = MINOR_START,
    .nr = ATMEL_MAX_UART,
    .cons = ATMEL_CONSOLE_DEVICE,
};

static struct platform_driver atmel_serial_driver = {
    .probe = atmel_serial_probe,
    .remove = __devexit_p(atmel_serial_remove),
    .suspend = atmel_serial_suspend,
    .resume = atmel_serial_resume,
    .driver = {
        .name = "atmel_usart",
        .owner = THIS_MODULE,
    },
};
```

Example code from `drivers/serial/atmel_serial.c`

uart_driver Code Example (2)

```
static int __init atmel_serial_init(void)
{
    /* Warning: Error management removed */
    uart_register_driver(&atmel_uart);
    platform_driver_register(&atmel_serial_driver);
    return 0;
}

static void __exit atmel_serial_exit(void)
{
    platform_driver_unregister(&atmel_serial_driver);
    uart_unregister_driver(&atmel_uart);
}

module_init(atmel_serial_init);
module_exit(atmel_serial_exit);
```

uart_port

- Can be allocated statically or dynamically
- Usually registered at `probe()` time and unregistered at `remove()` time
- Most important fields
  - `iotype`, type of I/O access, usually `UPIO_MEM` for memory-mapped devices
  - `mapbase`, physical address of the registers
  - `irq`, the IRQ channel number
  - `membase`, the virtual address of the registers
  - `uartclk`, the clock rate
  - `ops`, pointer to the operations
  - `dev`, pointer to the device (`platform_device` or other)

uart_port Code Example (1)

```
static int __devinit atmel_serial_probe(struct platform_device *pdev)
{
    struct atmel_uart_port *port;

    port = &atmel_ports[pdev->id];
    port->backup_imr = 0;

    atmel_init_port(port, pdev);

    uart_add_one_port(&atmel_uart, &port->uart);

    platform_set_drvdata(pdev, port);

    return 0;
}

static int __devexit atmel_serial_remove(struct platform_device *pdev)
{
    struct uart_port *port = platform_get_drvdata(pdev);

    platform_set_drvdata(pdev, NULL);
    uart_remove_one_port(&atmel_uart, port);

    return 0;
}
```

uart_port Code Example (2)

```
static void __devinit atmel_init_port(
    struct atmel_uart_port *atmel_port,
    struct platform_device *pdev)
{
    struct uart_port *port = &atmelt_port->uart;
    struct atmel_uart_data *data = pdev->dev.platform_data;

    port->iotype = UPIO_MEM;
    port->flags = UPF_BOOT_AUTOCONF;
    port->ops = &atmel_pops;
    port->fifosize = 1;
    port->line = pdev->id;
    port->dev = &pdev->dev;

    port->mapbase = pdev->resource[0].start;
    port->irq = pdev->resource[1].start;

    tasklet_init(&atmel_port->tasklet, atmel_tasklet_func,
        (unsigned long)port);
```

uart_port Code Example (3)

```
    if (data->regs)
        /* Already mapped by setup code */
        port->membase = data->regs;
    else {
        port->flags |= UPF_IOREMAP;
        port->membase = NULL;
    }

    /* for console, the clock could already be configured */
    if (!atmel_port->clk) {
        atmel_port->clk = clk_get(&pdev->dev, "usart");
        clk_enable(atmel_port->clk);
        port->uartclk = clk_get_rate(atmel_port->clk);
        clk_disable(atmel_port->clk);
        /* only enable clock when USART is in use */
    }
}
```

uart_ops

- Important operations
  - tx_empty(), tells whether the transmission FIFO is empty or not
  - set_mctrl() and get_mctrl(), allow to set and get the modem control parameters (RTS, DTR, LOOP, etc.)
  - start_tx() and stop_tx(), to start and stop the transmission
  - stop_rx(), to stop the reception
  - startup() and shutdown(), called when the port is opened/closed
  - request_port() and release_port(), request/release I/O or memory regions
  - set_termios(), change port parameters
- See the detailed description in Documentation/serial/driver

Implementing Transmission

- The `start_tx()` method should start transmitting characters over the serial port
- The characters to transmit are stored in a circular buffer, implemented by a `struct uart_circ` structure. It contains
  - `buf[]`, the buffer of characters
  - `tail`, the index of the next character to transmit. After transmit, tail must be updated using

    ```
    tail = tail &(UART_XMIT_SIZE - 1)
    ```
- Utility functions on `uart_circ`
  - `uart_circ_empty()`, tells whether the circular buffer is empty
  - `uart_circ_chars_pending()`, returns the number of characters left to transmit
- From an `uart_port` pointer, this structure can be reached using `port->state->xmit`

Polled-Mode Transmission

```
foo_uart_putc(struct uart_port *port, unsigned char c) {
    while(__raw_readl(port->membase + UART_REG1) & UART_TX_FULL)
        cpu_relax();
    __raw_writel(c, port->membase + UART_REG2);
}

foo_uart_start_tx(struct uart_port *port) {
    struct circ_buf *xmit = &port->state->xmit;

    while (!uart_circ_empty(xmit)) {
        foo_uart_putc(port, xmit->buf[xmit->tail]);
        xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);
        port->icount.tx++;
    }
}
```

Transmission with Interrupts (1)

```
foo_uart_interrupt(int irq, void *dev_id) {
    [...]
    if (interrupt_cause & END_OF_TRANSMISSION)
        foo_uart_handle_transmit(port);
    [...]
}

foo_uart_start_tx(struct uart_port *port) {
    enable_interrupt_on_txrdy();
}
```

Transmission with Interrupts (2)

```
foo_uart_handle_transmit(port) {
    struct circ_buf *xmit = &port->state->xmit;
    if (uart_circ_empty(xmit) || uart_tx_stopped(port)) {
        disable_interrupt_on_txrdy();
        return;
    }

    while (!uart_circ_empty(xmit)) {
        if (!(__raw_readl(port->membase + UART_REG1) &
            UART_TX_FULL))
            break;
        __raw_writel(xmit->buf[xmit->tail],
            port->membase + UART_REG2);
        xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);
        port->icount.tx++;
    }

    if (uart_circ_chars_pending(xmit) < WAKEUP_CHARS)
        uart_write_wakeup(port);
}
```

Reception

- On reception, usually in an interrupt handler, the driver must
  - Increment `port->icount.rx`
  - Call `uart_handle_break()` if a `BRK` has been received, and if it returns TRUE, skip to the next character
  - If an error occurred, increment `port->icount.parity`, `port->icount.frame`, `port->icount.overrun` depending on the error type
  - Call `uart_handle_sysrq_char()` with the received character, and if it returns TRUE, skip to the next character
  - Call `uart_insert_char()` with the received character and a status
    - Status is `TTY_NORMAL` is everything is OK, or `TTY_BREAK`, `TTY_PARITY`, `TTY_FRAME` in case of error
  - Call `tty_flip_buffer_push()` to push data to the TTY layer

Understanding Sysrq

- Part of the reception work is dedicated to handling `Sysrq`
  - Sysrq are special commands that can be sent to the kernel to make it reboot, unmount filesystems, dump the task state, nice real-time tasks, etc.
  - These commands are implemented at the lowest possible level so that even if the system is locked, you can recover it.
  - Through serial port: send a `BRK` character, send the character of the `Sysrq` command
  - See `Documentation/sysrq.txt`
- In the driver
  - `uart_handle_break()` saves the current time + 5 seconds in a variable
  - `uart_handle_sysrq_char()` will test if the current time is below the saved time, and if so, will trigger the execution of the Sysrq command

Reception Code Sample (1)

```
foo_receive_chars(struct uart_port *port) {
    int limit = 256;

    while (limit-- > 0) {
        status = __raw_readl(port->membase + REG_STATUS);
        ch = __raw_readl(port->membase + REG_DATA);
        flag = TTY_NORMAL;

        if (status & BREAK) {
            port->icount.break++;
            if (uart_handle_break(port))
                continue;
        }
        else if (status & PARITY)
            port->icount.parity++;
        else if (status & FRAME)
            port->icount.frame++;
        else if (status & OVERRUN)
            port->icount.overrun++;

        [...]
```

Reception Code Sample (2)

```
        [...]
        status &= port->read_status_mask;

        if (status & BREAK)
            flag = TTY_BREAK;
        else if (status & PARITY)
            flag = TTY_PARITY;
        else if (status & FRAME)
            flag = TTY_FRAME;

        if (uart_handle_sysrq_char(port, ch))
            continue;

        uart_insert_char(port, status, OVERRUN, ch, flag);
    }

    spin_unlock(& port->lock);
    tty_flip_buffer_push(port->state->port.tty);
    spin_lock(& port->lock);
}
```

Modem Control Lines

- Set using the `set_mctrl()` operation
  - The `mctrl` argument can be a mask of `TIOCM_RTS` (request to send), `TIOCM_DTR` (Data Terminal Ready), `TIOCM_OUT1`, `TIOCM_OUT2`, `TIOCM_LOOP` (enable loop mode)
  - If a bit is set in `mctrl`, the signal must be driven active, if the bit is cleared, the signal must be driven inactive
- Status using the `get_mctrl()` operation
  - Must return read hardware status and return a combination of `TIOCM_CD` (Carrier Detect), `TIOCM_CTS` (Clear to Send), `TIOCM_DSR` (Data Set Ready) and `TIOCM_RI` (Ring Indicator)

set_mctrl() Example

```
foo_set_mctrl(struct uart_port *uart, u_int mctrl) {
    unsigned int control = 0, mode = 0;

    if (mctrl & TIOCM_RTS)
        control |= ATMEL_US_RTSEN;
    else
        control |= ATMEL_US_RTSDIS;

    if (mctrl & TIOCM_DTS)
        control |= ATMEL_US_DTREN;
    else
        control |= ATMEL_US_DTRDIS;

    __raw_writel(port->membase + REG_CTRL, control);

    if (mctrl & TIOCM_LOOP)
        mode |= ATMEL_US_CHMODE_LOC_LOOP;
    else
        mode |= ATMEL_US_CHMODE_NORMAL;

    __raw_writel(port->membase + REG_MODE, mode);
}
```

get_mctrl() example

```
foo_get_mctrl(struct uart_port *uart, u_int mctrl) {
    unsigned int status, ret = 0;

    status = __raw_readl(port->membase + REG_STATUS);

    /*
     * The control signals are active low.
     */
     if (!(status & ATMEL_US_DCD))
         ret |= TIOCM_CD;
     if (!(status & ATMEL_US_CTS))
         ret |= TIOCM_CTS;
     if (!(status & ATMEL_US_DSR))
         ret |= TIOCM_DSR;
     if (!(status & ATMEL_US_RI))
         ret |= TIOCM_RI;

     return ret;
}
```

termios

- *The termios functions describe a general terminal interface that is provided to control asynchronous communication ports*
- A mechanism to control from userspace serial port parameters such as
  - Speed
  - Parity
  - Byte size
  - Stop bit
  - Hardware handshake
  - Etc.
- See `termios(3)` for details

set_termios()

- The `set_termios()` operation must
  - apply configuration changes according to the arguments
  - update `port->read_config_mask` and `port->ignore_config_mask` to indicate the events we are interested in receiving
- `static void set_termios(struct uart_port *port, struct ktermios *termios, struct ktermios *old)`
  - `port`, the port, `termios`, the new values and `old`, the old values
- Relevant `ktermios` structure fields are
  - `c_cflag` with word size, stop bits, parity, reception enable, CTS status change reporting, enable modem status change reporting
  - `c_iflag` with frame and parity errors reporting, break event reporting

set_termios() example (1)

```
static void atmel_set_termios(struct uart_port *port,
    struct ktermios *termios, struct ktermios *old)
{
    unsigned long flags;
    unsigned int mode, imr, quot, baud;

    mode = __raw_readl(port->membase + REG_MODE);
    baud = uart_get_baud_rate(port, termios, old, 0, port->uartclk / 16);
    /* Read current configuration */
    quot = uart_get_divisor(port, baud);

    /* Compute the mode modification for the byte size parameter */
    switch (termios->c_cflag & CSIZE) {
    case CS5:
        mode |= ATMEL_US_CHRL_5;
        break;
    case CS6:
        mode |= ATMEL_US_CHRL_6;
        break;
    [...]
    default:
        mode |= ATMEL_US_CHRL_8;
        break;
    }
```

set_termios() example (2)

```
/* Compute the mode modification for the stop bit */
if (termios->c_cflag & CSTOPB)
    mode |= ATMEL_US_NBSTOP_2;

/* Compute the mode modification for parity */
if (termios->c_cflag & PARENB) {
    /* Mark or Space parity */
    if (termios->c_cflag & CMSPAR) {
        if (termios->c_cflag & PARODD)
            mode |= ATMEL_US_PAR_MARK;
        else
            mode |= ATMEL_US_PAR_SPACE;
    } else if (termios->c_cflag & PARODD)
        mode |= ATMEL_US_PAR_ODD;
    else
        mode |= ATMEL_US_PAR_EVEN;
} else
    mode |= ATMEL_US_PAR_NONE;

/* Compute the mode modification for CTS reporting */
if (termios->c_cflag & CRTSCTS)
    mode |= ATMEL_US_USMODE_HWHS;
else
    mode |= ATMEL_US_USMODE_NORMAL;
```

set_termios() Example (3)

```
/* Compute the read_status_mask and ignore_status_mask
 * according to the events we're interested in. These
 * values are used in the interrupt handler. */
port->read_status_mask = ATMEL_US_OVRE;
if (termios->c_iflag & INPCK)
    port->read_status_mask |= (ATMEL_US_FRAME | ATMEL_US_PARE);
if (termios->c_iflag & (BRKINT | PARMRK))
    port->read_status_mask |= ATMEL_US_RXBRK;

port->ignore_status_mask = 0;
if (termios->c_iflag & IGNPAR)
    port->ignore_status_mask |= (ATMEL_US_FRAME | ATMEL_US_PARE);
if (termios->c_iflag & IGNBRK) {
    port->ignore_status_mask |= ATMEL_US_RXBRK;
    if (termios->c_iflag & IGNPAR)
        port->ignore_status_mask |= ATMEL_US_OVRE;
}
/* The serial_core maintains a timeout that corresponds to the
 * duration it takes to send the full transmit FIFO. This timeout has
 * to be updated. */
uart_update_timeout(port, termios->c_cflag, baud);
```

set_termios() Example (4)

```
    /* Finally, apply the mode and baud rate modifications. Interrupts,
     * transmission and reception are disabled when the modifications
     * are made. */

    /* Save and disable interrupts */
    imr = UART_GET_IMR(port);
    UART_PUT_IDR(port, -1);
    /* disable receiver and transmitter */
    UART_PUT_CR(port, ATMEL_US_TXDIS | ATMEL_US_RXDIS);
    /* set the parity, stop bits and data size */
    UART_PUT_MR(port, mode);
    /* set the baud rate */
    UART_PUT_BRGR(port, quot);
    UART_PUT_CR(port, ATMEL_US_RSTSTA | ATMEL_US_RSTRX);
    UART_PUT_CR(port, ATMEL_US_TXEN | ATMEL_US_RXEN);
    /* restore interrupts */
    UART_PUT_IER(port, imr);
    /* CTS flow-control and modem-status interrupts */
    if (UART_ENABLE_MS(port, termios->c_cflag))
        port->ops->enable_ms(port);
}
```

Console

- To allows early boot messages to be printed, the kernel provides a separate but related facility: `console`
  - This console can be enabled using the `console=` kernel argument
- The driver developer must
  - Implement a `console_write()` operation, called to print characters on the console
  - Implement a `console_setup()` operation, called to parse the `console=` argument
  - Declare a `struct console` structure
  - Register the console using a `console_initcall()` function

Console: Registration

```
static struct console serial_txx9_console = {
    .name = TXX9_TTY_NAME,
    .write = serial_txx9_console_write,
    /* Helper function from the serial_core layer */
    .device = uart_console_device,
    .setup = serial_txx9_console_setup,
    /* Ask for the kernel messages buffered during
     * boot to be printed to the console when activated */
    .flags = CON_PRINTBUFFER,
    .index = -1,
    .data = &serial_txx9_reg,
};

static int __init serial_txx9_console_init(void)
{
    register_console(&serial_txx9_console);
    return 0;
}

/* This will make sure the function is called early during the boot process.
 * start_kernel() calls console_init() that calls our function */
console_initcall(serial_txx9_console_init);
```

Console: Setup

```
static int __init serial_txx9_console_setup(struct console *co,
    char *options)
{
    struct uart_port *port;
    struct uart_txx9_port *up;
    int baud = 9600;
    int bits = 8;
    int parity = 'n';
    int flow = 'n';

    if (co->index >= UART_NR)
        co->index = 0;
    up = &serial_txx9_ports[co->index];
    port = &up->port;
    if (!port->ops)
        return -ENODEV;

    /* Function shared with the normal serial driver */
    serial_txx9_initialize(&up->port);

    if (options)
        /* Helper function from serial_core that parses the console= string */
        uart_parse_options(options, &baud, &parity, &bits, &flow);

    /* Helper function from serial_core that calls the ->set_termios() */
    /* operation with the proper arguments to configure the port */
    return uart_set_options(port, co, baud, parity, bits, flow);
}
```

Console: Write

```
static void serial_txx9_console_putchar(struct uart_port *port, int ch)
{
struct uart_txx9_port *up = (struct uart_txx9_port *)port;
/* Busy-wait for transmitter ready and output a single character. */
wait_for_xmitr(up);
sio_out(up, TXX9_SITFIFO, ch);
}

static void serial_txx9_console_write(struct console *co,
    const char *s, unsigned int count)
{
    struct uart_txx9_port *up = &serial_txx9_ports[co->index];
    unsigned int ier, flcr;

    /* Disable interrupts */
    ier = sio_in(up, TXX9_SIDICR);
    sio_out(up, TXX9_SIDICR, 0);

    /* Disable flow control */
    flcr = sio_in(up, TXX9_SIFLCR);
    if (!(up->port.flags & UPF_CONS_FLOW) && (flcr & TXX9_SIFLCR_TES))
        sio_out(up, TXX9_SIFLCR, flcr & ~TXX9_SIFLCR_TES);

    /* Helper function from serial_core that repeatedly calls the given putchar() */
    /* callback */
    uart_console_write(&up->port, s, count, serial_txx9_console_putchar);

    /* Re-enable interrupts */
    wait_for_xmitr(up);
    sio_out(up, TXX9_SIFLCR, flcr);
    sio_out(up, TXX9_SIDICR, ier);
}
```

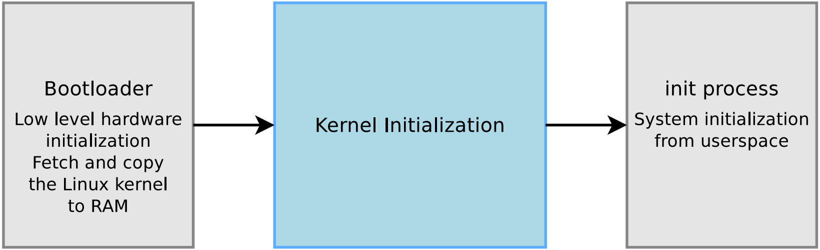
Practical lab - Serial drivers

- Improve the character driver of the previous labs to make it a real serial driver

Kernel Initialization

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

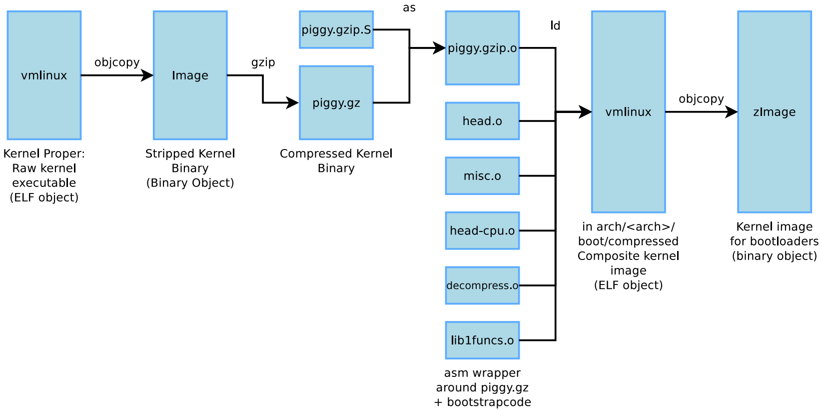
From Bootloader to Userspace



Kernel Bootstrap (1)

How the kernel bootstraps itself appears in kernel building.
Example on ARM (pxa cpu) in Linux 2.6.36:

```
...
  LD      vmlinux
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  AS      arch/arm/boot/compressed/head.o
  GZIP    arch/arm/boot/compressed/piggy.gzip
  AS      arch/arm/boot/compressed/piggy.gzip.o
  CC      arch/arm/boot/compressed/misc.o
  CC      arch/arm/boot/compressed/decompress.o
  AS      arch/arm/boot/compressed/head-xscale.o
  SHIPPED arch/arm/boot/compressed/lib1funcs.S
  AS      arch/arm/boot/compressed/lib1funcs.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
...
```


vmlinux
objcopy
Image
gzip
piggy.gzip.S
piggy.gz
as
piggy.gzip.o
head.o
misc.o
head-cpu.o
decompress.o
lib1funcs.o
ld
vmlinux
objcopy
zImage
Kernel Proper: Raw kernel executable (ELF object)
Stripped Kernel Binary (Binary Object)
Compressed Kernel Binary
asm wrapper around piggy.gz + bootstrapcode
in arch/<arch>/ boot/compressed Composite kernel image (ELF object)
Kernel image for bootloaders (binary object)

Bootstrap Code for Compressed Kernels

- Located in `arch/<arch>/boot/compressed`
  - `head.o`
    - Architecture specific initialization code.
    - This is what is executed by the bootloader
  - `head-cpu.o` (here `head-xscale.o`)
    - CPU specific initialization code
  - `decompress.o`, `misc.o`
    - Decompression code
  - `piggy.<compressionformat>.o`
    - The kernel itself
- Responsible for uncompressing the kernel itself and jumping to its entry point.

Architecture-specific Initialization Code

- The uncompression code jumps into the main kernel entry point, typically located in `arch/<arch>/kernel/head.S`, whose job is to:
  - Check the architecture, processor and machine type.
  - Configure the MMU, create page table entries and enable virtual memory.
  - Calls the `start_kernel` function in `init/main.c`.
  - Same code for all architectures.
  - Anybody interested in kernel startup should study this file!

start_kernel Main Actions

- Calls `setup_arch(&command_line)`
  - Function defined in `arch/<arch>/kernel/setup.c`
  - Copying the command line from where the bootloader left it.
  - On arm, this function calls `setup_processor` (in which CPU information is displayed) and `setup_machine`(locating the machine in the list of supported machines).
- Initializes the console as early as possible (to get error messages)
- Initializes many subsystems (see the code)
- Eventually calls `rest_init`.

rest_init: Starting the Init Process

```
static noinline void __init_refok rest_init(void)
        __releases(kernel_lock)
{
        int pid;

        rcu_scheduler_starting();
        /*
         * We need to spawn init first so that it obtains pid 1, however
         * the init task will end up wanting to create kthreads, which, if
         * we schedule it before we create kthreadd, will OOPS.
         */
        kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
        numa_default_policy();
        pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
        rcu_read_lock();
        kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
        rcu_read_unlock();
        complete(&kthreadd_done);

        /*
         * The boot idle thread must execute schedule()
         * at least once to get things moving:
         */
        init_idle_bootup_task(current);
        preempt_enable_no_resched();
        schedule();
        preempt_disable();

        /* Call into cpu_idle with preempt disabled */
        cpu_idle();
}
```

- kernel_init does two main things:
  - Call do_basic_setup
  - Once kernel services are ready, start device initialization (Linux 2.6.36 code excerpt):

```
static void __init do_basic_setup(void)
{
    cpuset_init_smp();
    usermodehelper_init();
    init_tmpfs();
    driver_init();
    init_irq_proc();
    do_ctors();
    do_initcalls();
}
```

  - Call init_post

do_initcalls

Calls pluggable hooks registered with the macros below.
Advantage: the generic code doesn't have to know about them.

```
/*
 * A "pure" initcall has no dependencies on anything else, and purely
 * initializes variables that couldn't be statically initialized.
 *
 * This only exists for built-in code, not for modules.
 */
#define pure_initcall(fn)               __define_initcall("0",fn,1)

#define core_initcall(fn)               __define_initcall("1",fn,1)
#define core_initcall_sync(fn)          __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)           __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)      __define_initcall("2s",fn,2s)
#define arch_initcall(fn)               __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)          __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)             __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)        __define_initcall("4s",fn,4s)
#define fs_initcall(fn)                 __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)            __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)             __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)             __define_initcall("6",fn,6)
#define device_initcall_sync(fn)        __define_initcall("6s",fn,6s)
#define late_initcall(fn)               __define_initcall("7",fn,7)
#define late_initcall_sync(fn)          __define_initcall("7s",fn,7s)
```

Defined in `include/linux/init.h`

initcall example

From arch/arm/mach-pxa/lpd270.c (Linux 2.6.36)

```
static int __init lpd270_irq_device_init(void)
{
    int ret = -ENODEV;
    if (machine_is_logicpd_pxa270()) {
        ret = sysdev_class_register(&lpd270_irq_sysclass);
        if (ret == 0)
            ret = sysdev_register(&lpd270_irq_device);
    }
    return ret;
}

device_initcall(lpd270_irq_device_init);
```

init_post

- The last step of Linux booting
  - First tries to open a console
  - Then tries to run the init process, effectively turning the current kernel thread into the userspace init process.

init_post Code: init/main.c

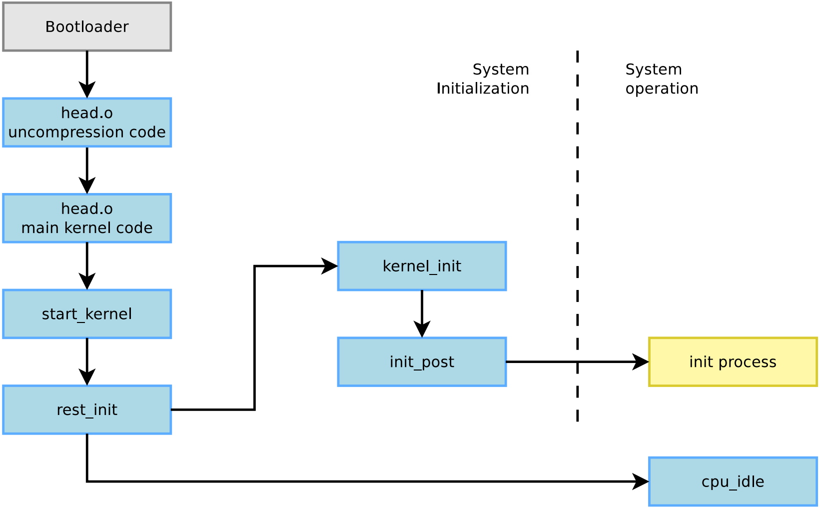
```
static noinline int init_post(void) __releases(kernel_lock) {
    /* need to finish all async __init code before freeing the memory */
    async_synchronize_full();
    free_initmem();
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();

    current->signal->flags |= SIGNAL_UNKILLABLE;
    if (ramdisk_execute_command) {
        run_init_process(ramdisk_execute_command);
        printk(KERN_WARNING "Failed to execute %s\n", ramdisk_execute_command);
    }

    /* We try each of these until one succeeds.
     * The Bourne shell can be used instead of init if we are
     * trying to recover a really broken machine. */
    if (execute_command) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s.  Attempting defaults...\n", execute_command);
    }
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found. Try passing init= option to kernel. See Linux Documentation/init.txt");
}
```

Kernel Initialization Graph



Kernel Initialization - Summary

- The bootloader executes bootstrap code.
- Bootstrap code initializes the processor and board, and uncompresses the kernel code to RAM, and calls the kernel's `start_kernel` function.
- Copies the command line from the bootloader.
- Identifies the processor and machine.
- Initializes the console.
- Initializes kernel services (memory allocation, scheduling, file cache...)
- Creates a new kernel thread (future init process) and continues in the idle loop.
- Initializes devices and execute initcalls.

Porting the Linux Kernel to an ARM Board

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Porting the Linux kernel

- The Linux kernel supports a lot of different CPU architectures
- Each of them is maintained by a different group of contributors
  - See the `MAINTAINERS` file for details
- The organization of the source code and the methods to port the Linux kernel to a new board are therefore very architecture-dependent
- For example, PowerPC and ARM are very different
  - PowerPC relies on device trees to describe hardware details
  - ARM relies on source code only, but the migration to device tree is in progress
- This presentation is focused on the ARM architecture only

Architecture, CPU and Machine

- In the source tree, each architecture has its own directory
  - `arch/arm` for the ARM architecture
- This directory contains generic ARM code
  - `boot`, `common`, `configs`, `kernel`, `lib`, `mm`, `nwfpe`, `vfp`, `oprofile`, `tools`
- And many directories for different SoC families
  - `mach-*` directories: `mach-pxa` for PXA CPUs, `mach-imx` for Freescale iMX CPUs, etc.
  - Each of these directories contain
    - Support for the SoC family (GPIO, clocks, pinmux, power management, interrupt controller, etc.)
    - Support for several boards using this SoC
- Some CPU types share some code, in directories named `plat-*`

Source Code for Calao USB A9263

- Taking the case of the Calao USB A9263 board, which uses a AT91SAM9263 CPU.
- `arch/arm/mach-at91`
  - AT91 generic code
    - `clock.c`
    - `leds.c`
    - `irq.c`
    - `pm.c`
  - CPU-specific code for the AT91SAM9263
    - `at91sam9263.c`
    - `at91sam926x_time.c`
    - `at91sam9263_devices.c`
  - Board specific code
    - `board-usb-a9263.c`
- For the rest of this presentation, we will focus on board support only

Configuration

- A configuration option must be defined for the board, in `arch/arm/mach-at91/Kconfig`

```
config MACH_USB_A9263
        bool "CALAO USB-A9263"
        depends on ARCH_AT91SAM9263
        help
          Select this if you are using a Calao Systems USB-A9263.
          <http://www.calao-systems.com>
```

- This option must depend on the CPU type option corresponding to the CPU used in the board
  - Here the option is `ARCH_AT91SAM9263`, defined in the same file
- A default configuration file for the board can optionally be stored in `arch/arm/configs/`. For our board, it's `at91sam9263_defconfig`

Compilation

- The source files corresponding to the board support must be associated with the configuration option of the board

```
obj-$(CONFIG_MACH_USB_A9263) += board-usb-a9263.o
```

- This is done in arch/arm/mach-at91/Makefile

```
obj-y              := irq.o gpio.o
obj-$(CONFIG_AT91_PMC_UNIT)   += clock.o
obj-y                         += leds.o
obj-$(CONFIG_PM)              += pm.o
obj-$(CONFIG_AT91_SLOW_CLOCK) += pm_slowclock.o
```

- The Makefile also tells which files are compiled for every AT91 CPU
- And which files for our particular CPU, the AT91SAM9263

```
obj-$(CONFIG_ARCH_AT91SAM9263) += at91sam9263.o
at91sam926x_time.o at91sam9263_devices.o sam9_smc.o
```

Machine Structure

- Each board is defined by a machine structure
  - The word *machine* is quite confusing since every `mach-*` directory contains several machine definitions, one for each board using a given CPU type
- For the Calao board, at the end of `arch/arm/mach-at91/board-usb-a926x.c`

```
MACHINE_START(USB_A9263, "CALAO USB_A9263")
    /* Maintainer: calao-systems */
    .phys_io = AT91_BASE_SYS,
    .io_pg_offst = (AT91_VA_BASE_SYS >> 18) & 0xfffc,
    .boot_params = AT91_SDRAM_BASE + 0x100,
    .timer = &at91sam926x_timer,
    .map_io = ek_map_io,
    .init_irq = ek_init_irq,
    .init_machine = ek_board_init,
MACHINE_END
```

Machine Structure Macros

- MACHINE_START and MACHINE_END
  - Macros defined in arch/arm/include/asm/mach/arch.h
  - They are helpers to define a struct machine_desc structure stored in a specific ELF section
  - Several machine_desc structures can be defined in a kernel, which means that the kernel can support several boards.
  - The right structure is chosen at boot time

Machine Type Number

- In the ARM architecture, each board type is identified by a machine type number
- The latest machine type numbers list can be found at `http://www.arm.linux.org.uk/developer/machines/download.php`
- A copy of it exists in the kernel tree in `arch/arm/tools/mach-types`
  - For the Calao board
    - `usb_a9263 MACH_USB_A9263 USB_A9263 1710`
- At compile time, this file is processed to generate a header file, `include/asm-arm/mach-types.h`
  - For the Calao board
    - `#define MACH_TYPE_USB_A9263 1710`
  - And a few other macros in the same file

Machine Type Number

- The machine type number is set in the MACHINE_START() definition
  - MACHINE_START(USB_A9263, "CALAO USB_A9263")
- At run time, the machine type number of the board on which the kernel is running is passed by the bootloader in register r1
- Very early in the boot process (arch/arm/kernel/head.S), the kernel calls __lookup_machine_type in arch/arm/kernel/head-common.S
- __lookup_machine_type looks at all the machine_desc structures of the special ELF section
  - If it doesn't find the requested number, prints a message and stops
  - If found, it knows the machine descriptions and continues the boot process

Early Debugging and Boot Parameters

- Early debugging
  - `phys_io` is the physical address of the I/O space
  - `io_pg_offset` is the offset in the page table to remap the I/O space
  - These are used when `CONFIG_DEBUG_LL` is enabled to provide very early debugging messages on the serial port
- Boot parameters
  - `boot_params` is the location where the bootloader has left the boot parameters (the kernel command line)
  - The bootloader can override this address in register `r2`
  - See also `Documentation/arm/Booting` for the details of the environment expected by the kernel when booted

System Timer

- The timer field points to a `struct sys_timer` structure, that describes the system timer
  - Used to generate the periodic tick at HZ frequency to call the scheduler periodically
- On the Calao board, the system timer is defined by the `at91sam926x_timer` structure in `at91sam926x_time.c`
- It contains the interrupt handler called at HZ frequency
- It is integrated with the `clockevents` and the `clocksource` infrastructures
  - See `include/linux/clocksource.h` and `include/linux/clockchips.h` for details

map_io()

- The `map_io()` function points to `ek_map_io()`, which
  - Initializes the CPU using `at91sam9263_initialize()`
    - Map I/O space
    - Register and initialize the clocks
  - Configures the debug serial port and set the console to be on this serial port
  - Called at the very beginning of the C code execution
    - `init/main.c`: `start_kernel()`
    - `arch/arm/kernel/setup.c`: `setup_arch()`
    - `arch/arm/mm/mmu.c`: `paging_init()`
    - `arch/arm/mm/mmu.c`: `devicemaps_init()`
    - `mdesc->map_io()`

init_irq()

- init_irq() to initialize the IRQ hardware specific details
- Implemented by ek_init_irq(), which calls at91sam9263_init_interrupts() in at91sam9263.c, which mainly calls at91_aic_init() in irq.c
  - Initialize the interrupt controller, assign the priorities
  - Register the IRQ chip (irq_chip structure) to the kernel generic IRQ infrastructure, so that the kernel knows how to ack, mask, unmask the IRQs
- Called a little bit later than map_io()
  - init/main.c: start_kernel()
  - arch/arm/kernel/irq.c: init_IRQ()
  - init_arch_irq() (equal to mdesc->init_irq)

init_machine()

- init_machine() completes the initialization of the board by registering all platform devices
- Called by customize_machines() in arch/arm/kernel/setup.c
- This function is an arch_initcall (list of functions whose address is stored in a specific ELF section, by levels)
- At the end of kernel initialization, just before running the first userspace program init:
  - init/main.c: kernel_init()
  - init/main.c: do_basic_setup()
  - init/main.c: do_initcalls()
  - Calls all initcalls, level by level

init_machine() for Calao

- For the Calao board, implemented in `ek_board_init()`
  - Registers serial ports, USB host, USB device, SPI, Ethernet, NAND flash, 2IC, buttons and LEDs
  - Uses `at91_add_device_*()` helpers, defined in `at91sam9263_devices.c`
  - These helpers call `platform_device_register()` to register the different `platform_device` structures defined in the same file
  - For some devices, the board specific code does the registration itself (buttons) or passes board-specific data to the registration helper (USB host and device, NAND, Ethernet, etc.)

Drivers

- The `at91sam9263_devices.c` file doesn't implement the drivers for the platform devices
- The drivers are implemented at different places of the kernel tree
- For the Calao board
  - USB host, driver `at91_ohci`, `drivers/usb/host/ohci-at91.c`
  - USB device, driver `at91_udc`, `drivers/usb/gadget/at91_udc.c`
  - Ethernet, driver `macb`, `drivers/net/macb.c`
  - NAND, driver `atmel_nand`, `drivers/mtd/nand/atmel_nand.c`
  - I2C on GPIO, driver `i2c-gpio`, `drivers/i2c/busses/i2c-gpio.c`
  - SPI, driver `atmel_spi`, `drivers/spi/atmel_spi.c`
  - Buttons, driver `gpio-keys`, `drivers/input/keyboard/gpio_keys.c`
- All these drivers are selected by the default configuration file

New Directions in the ARM Architecture

- The ARM architecture is migrating to the device tree
  - *The Device Tree is a data structure for describing hardware*
  - Instead of describing the hardware in C, a special data structure, external to the kernel is used
  - Allows to more easily port the kernel to newer platforms and to make a single kernel image support multiple platforms
- The ARM architecture is being consolidated
  - The *clock* API is being converted to a proper framework, with drivers in `drivers/clk`
  - The GPIO support is being converted as proper GPIO drivers in `drivers/gpio`
  - The pin muxing support is being converted as drivers in `drivers/pinctrl`

Board Device Tree Example: tegra-harmony.dts

```
/dts-v1/;
/memreserve/ 0x1c000000 0x04000000;
/include/ "tegra20.dtsi"
/ {
    model = "NVIDIA Tegra2 Harmony evaluation board";
    compatible = "nvidia,harmony", "nvidia,tegra20";
    chosen {
        bootargs = "vmalloc=192M video=tegrafb console=ttyS0,115200n8";
    };

    memory@0 {
        reg = < 0x00000000 0x40000000 >;
    };

    i2c@7000c000 {
        clock-frequency = <400000>;

        codec: wm8903@1a {
            compatible = "wlf,wm8903";
            reg = <0x1a>;
            interrupts = < 347 >;

            gpio-controller;
            #gpio-cells = <2>;

            /* 0x8000 = Not configured */
            gpio-cfg = < 0x8000 0x8000 0 0x8000 0x8000 >;
        };
    };
    [...]
};
```

Device Tree Usage

- The *device tree source* (`.dts`) is compiled into a *device tree blob* (`.dtb`) using a *device tree compiler* (`.dtc`)
  - The `dtb` is an efficient binary data structure
  - The `dtb` is either appended to the kernel image, or better, passed by the bootloader to the kernel
- At runtime, the kernel parses the device tree to find out
  - which devices are present
  - what drivers are needed
  - which parameters should be used to initialize the devices
- On ARM, device tree support is only beginning

Porting to a New Board: Advise

- Porting Linux to a new board is easy, when Linux already supports the evaluation kit / development board for your CPU.
- Most work has already been done and it is just a matter of customizing devices instantiated on your boards and their settings.
- Therefore, look for how the development board is supported, or at least for a similar board with the same CPU.
- For example, review the (few) differences between the Calao `qil-a9260` board and Atmel's `sam9260` Evaluation Kit:
  - `meld board-sam9260ek.c board-qil-a9260.c`
- Similarly, you will find very few differences in U-boot between code for a board and for the corresponding evaluation board.

Power Management

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

PM Building Blocks

- Several power management *building blocks*
  - Suspend and resume
  - CPUidle
  - Runtime power management
  - Frequency and voltage scaling
  - Applications
- Independent *building blocks* that can be improved gradually during development

Clock Framework (1)

- Generic framework to manage clocks used by devices in the system
- Allows to reference count clock users and to shutdown the unused clocks to save power
- Simple API described in `http://free-electrons.com/kerneldoc/latest/DocBook/kernel-api/clk.html`
  - `clk_get()` to get a reference to a clock
  - `clk_enable()` to start the clock
  - `clk_disable()` to stop the clock
  - `clk_put()` to free the clock source
  - `clk_get_rate()` to get the current rate

Clock Framework (2)

- The clock framework API and the `clk` structure are usually implemented by each architecture (code duplication!)
  - See `arch/arm/mach-at91/clock.c` for an example
  - This is also where all clocks are defined.
  - Clocks are identified by a name string specific to a given platform
- Drivers can then use the clock API. Example from `drivers/net/macb.c`:
  - `clk_get()` called from the `probe()` function, to get the definition of a clock for the current board, get its frequency, and run `clk_enable()`.
  - `clk_put()` called from the `remove()` function to release the reference to the clock, after calling `clk_disable()`

Clock Disable Implementation

From arch/arm/mach-at91/clock.c: (2.6.36)

```
static void __clk_disable(struct clk *clk)
{
    BUG_ON(clk->users == 0);
    if (--clk->users == 0 && clk->mode)
        /* Call the hardware function switching off this clock */
        clk->mode(clk, 0);
    if (clk->parent)
        __clk_disable(clk->parent);
}

[...]

static void pmc_sys_mode(struct clk *clk, int is_on)
{
    if (is_on)
        at91_sys_write(AT91_PMC_SCER, clk->pmc_mask);
    else
        at91_sys_write(AT91_PMC_SCDR, clk->pmc_mask);
}
```

Suspend and Resume

- Infrastructure in the kernel to support suspend and resume
- Platform hooks
  - `prepare()`, `enter()`, `finish()`, `valid()` in a `platform_suspend_ops` structure
  - Registered using the `suspend_set_ops()` function
  - See `arch/arm/mach-at91/pm.c`
- Device drivers
  - `suspend()` and `resume()` hooks in the `*_driver` structures (`platform_driver`, `usb_driver`, etc.)
  - See `drivers/net/macb.c`

Board-specific Power Management

- Typically takes care of battery and charging management.
- Also defines `presuspend` and `postsuspend` handlers.
- Example: `arch/arm/mach-pxa/spitz_pm.c`

arch/arm/mach-cpu/sleep.S

- Assembly code implementing CPU specific suspend and resume code.
- Note: only found on arm, just 3 other occurrences in other architectures, with other paths.
- First scenario: only a suspend function. The code goes in sleep state (after enabling DRAM self-refresh), and continues with resume code.
- Second scenario: suspend and resume functions. Resume functions called by the bootloader.
- Examples to look at:
  - `arch/arm/mach-omap2/sleep24xx.S` (1st case)
  - `arch/arm/mach-pxa/sleep.S` (2nd case)

Triggering Suspend

- Whatever the power management implementation, CPU specific `suspend_ops` functions are called by the `enter_state` function.
- `enter_state` also takes care of executing the suspend and resume functions for your devices.
- The execution of this function can be triggered from userspace. To suspend to RAM:
  - `echo mem > /sys/power/state`
- Can also use the s2ram program from http://suspend.sourceforge.net/
- Read `kernel/power/suspend.c`

Runtime Power Management

- According to the kernel configuration interface: *Enable functionality allowing I/O devices to be put into energy-saving (low power) states at run time (or autosuspended) after a specified period of inactivity and woken up in response to a hardware-generated wake-up event or a driver's request.*
- New hooks must be added to the drivers: `runtime_suspend()`, `runtime_resume()`, `runtime_idle()`
- API and details on `Documentation/power/runtime_pm.txt`
- See also Kevin Hilman's presentation at ELC Europe 2010: http://elinux.org/images/c/cd/ELC-2010-khilman-Runtime-PM.odp

Saving Power in the Idle Loop

- The idle loop is what you run when there's nothing left to run in the system.
- Implemented in all architectures in `arch/<arch>/kernel/process.c`
- Example to read: look for `cpu_idle` in `arch/arm/kernel/process.c`
- Each ARM cpu defines its own `arch_idle` function.
- The CPU can run power saving HLT instructions, enter NAP mode, and even disable the timers (tickless systems).
- See also `http://en.wikipedia.org/wiki/Idle_loop`

Managing Idle

- Adding support for multiple idle levels
  - Modern CPUs have several sleep states offering different power savings with associated wake up latencies
  - Since 2.6.21, the dynamic tick feature allows to remove the periodic tick to save power, and to know when the next event is scheduled, for smarter sleeps.
  - CPUidle infrastructure to change sleep states
    - Platform-specific driver defining sleep states and transition operations
    - Platform-independent governors (ladder and menu)
    - Available for x86/ACPI, not supported yet by all ARM cpus. (look for `cpuidle*` files under `arch/arm/`)
    - See `Documentation/cpuidle/` in kernel sources.

PowerTOP

- http://www.lesswatts.org/projects/powertop/
  - With dynamic ticks, allows to fix parts of kernel code and applications that wake up the system too often.
  - PowerTOP allows to track the worst offenders
  - Now available on ARM cpus implementing CPUidle
  - Also gives you useful hints for reducing power.

Frequency and Voltage Scaling (1)

- Frequency and voltage scaling possible through the `cpufreq` kernel infrastructure.
  - Generic infrastructure: `drivers/cpufreq/cpufreq.c` and `include/linux/cpufreq.h`
  - Generic governors, responsible for deciding frequency and voltage transitions
    - `performance`: maximum frequency
    - `powersave`: minimum frequency
    - `ondemand`: measures CPU consumption to adjust frequency
    - `conservative`: often better than `ondemand`. Only increases frequency gradually when the CPU gets loaded.
    - `userspace`: leaves the decision to a userspace daemon.
  - This infrastructure can be controlled from `/sys/devices/system/cpu/cpu<n>/cpufreq/`

Frequency and Voltage Scaling (2)

- CPU support code in architecture dependent files. Example to read: `arch/arm/plat-omap/cpu-omap.c`
- Must implement the operations of the `cpufreq_driver` structure and register them using `cpufreq_register_driver()`
  - `init()` for initialization
  - `exit()` for cleanup
  - `verify()` to verify the user-chosen policy
  - `setpolicy()` or `target()` to actually perform the frequency change
- See `Documentation/cpu-freq/` for useful explanations

PM QoS

- PM QoS is a framework developed by Intel introduced in 2.6.25
- It allows kernel code and applications to set their requirements in terms of
  - CPU DMA latency
  - Network latency
  - Network throughput
- According to these requirements, PM QoS allows kernel drivers to adjust their power management
- See `Documentation/power/pm_qos_interface.txt` and Mark Gross' presentation at ELC 2008
- Still in very early deployment (only 4 drivers in 2.6.36).

Regulator Framework

- Modern embedded hardware have hardware responsible for voltage and current regulation
- The regulator framework allows to take advantage of this hardware to save power when parts of the system are unused
  - A consumer interface for device drivers (i.e users)
  - Regulator driver interface for regulator drivers
  - Machine interface for board configuration
  - sysfs interface for userspace
- Merged in Linux 2.6.27.
- See `Documentation/power/regulator/` in kernel sources.
- See Liam Girdwood's presentation at ELC 2008 http://free-electrons.com/blog/elc-2008-report#girdwood

BSP Work for a New Board

- In case you just need to create a BSP for your board, and your CPU already has full PM support, you should just need to:
  - Create clock definitions and bind your devices to them.
  - Implement PM handlers (suspend, resume) in the drivers for your board specific devices.
  - Implement runtime PM handlers in your drivers.
  - Implement board specific power management if needed (mainly battery management)
  - Implement regulator framework hooks for your board if needed.
  - All other parts of the PM infrastructure should be already there: suspend / resume, cpuidle, cpu frequency and voltage scaling.

Useful Resources

- Documentation/power/ in the Linux kernel sources.
  - Will give you many useful details.
- http://lesswatts.org
  - Intel effort trying to create a Linux power saving community.
  - Mainly targets Intel processors.
  - Lots of useful resources.
- http://wiki.linaro.org/WorkingGroups/PowerManagement/
  - Ongoing developments on the ARM platform.
- Tips and ideas for prolonging battery life
  - http://j.mp/fVdxKh

- Suspend and resume your Linux system
- Change the CPU frequency of your system

Kernel Advice and Resources

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Advice

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.

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`
- Check for previous bugs reports. 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 Become a Kernel Developer?

- Recommended resources
  - See `Documentation/SubmittingPatches` for guidelines and `http://kernelnewbies.org/UpstreamMerge` for very helpful advice to have your changes merged upstream (by Rik van Riel).
  - Watch the *Write and Submit your first Linux kernel Patch* talk by Greg. K.H: `http://www.youtube.com/watch?v=LLBrBBImJt4`
  - How to Participate in the Linux Community (by Jonathan Corbet) A Guide To The Kernel Development Process `http://j.mp/tX2Ld6`

How to Submit Patches or Drivers

- Use git to prepare make your changes
- Don't merge patches addressing different issues
- Make sure that your changes compile well, and if possible, run well.
- Run Linux patch checks: `scripts/checkpatch.pl`
- Send the patches to yourself first, as an inline attachment. This is required to let people reply to parts of your patches. Make sure your patches still applies. See `Documentation/email-clients.txt` for help configuring e-mail clients. Best to use `git send-email`, which never corrupts patches.
- Run `scripts/get_maintainer.pl` on your patches, to know who you should send them to.

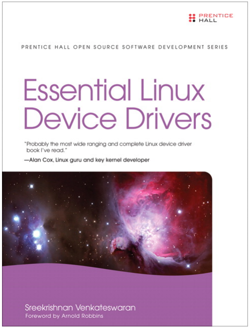
References

Kernel Development News

- 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 ($7 / month)
  - Articles available for non subscribers after 1 week.

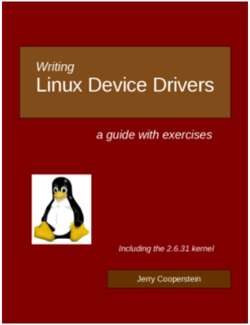
Useful Reading (1)

- Essential Linux Device Drivers, April 2008
  - `http://free-electrons.com/redirect/eldd-book.html`
  - By Sreekrishnan Venkateswaran, an embedded IBM engineer with more than 10 years of experience
  - Covers a wide range of topics not covered by LDD: serial drivers, input drivers, I2C, PCMCIA and Compact Flash, PCI, USB, video drivers, audio drivers, block drivers, network drivers, Bluetooth, IrDA, MTD, drivers in userspace, kernel debugging, etc.
  - *Probably the most wide ranging and complete Linux device driver book I've read* – Alan Cox



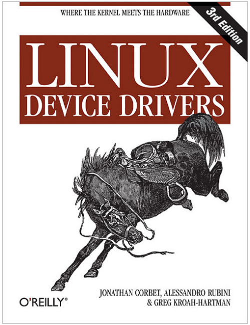
Useful Reading (2)

- Writing Linux Device drivers, September 2009
  - http://www.coopj.com/
  - Self published by Jerry Cooperstein
  - Available like any other book (Amazon and others)
  - Though not as thorough as the previous book on specific drivers, still a good complement on multiple aspects of kernel and device driver development.
  - Based on Linux 2.6.31
  - Multiple exercises. Updated solutions for 2.6.36.



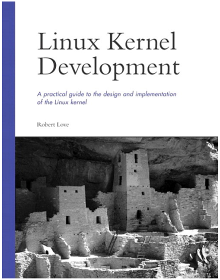
Useful Reading (3)

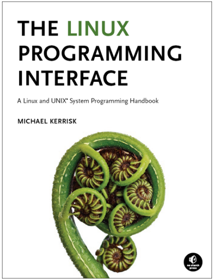
- Linux Device Drivers, 3rd edition, Feb 2005
  - `http://www.oreilly.com/catalog/linuxdrive3/`
  - By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, O'Reilly
  - Freely available on-line! Great companion to the printed book for easy electronic searches!
  - `http://lwn.net/Kernel/LDD3/` (1 PDF file per chapter)
  - `http://free-electrons.com/community/kernel/ldd3/` (single PDF file)
  - Getting outdated but still useful for Linux device driver writers!



Useful Reading (4)

- Linux Kernel Development, 3rd Edition, Jun 2010
  - Robert Love, Novell Press
  - http://free-electrons.com/redir/lkd3-book.html
  - A very synthetic and pleasant way to learn about kernel subsystems (beyond the needs of device driver writers)
- The Linux Programming Interface, Oct 2010
  - Michael Kerrisk, No Starch Press
  - http://man7.org/tlpi/
  - A gold mine about the kernel interface and how to use it





Useful Online Resources

- Kernel documentation (`Documentation/` in kernel sources)
  - Available on line: `http://free-electrons.com/kerneldoc/` (with HTML documentation extracted from source code)
- 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`

International Conferences

- Embedded Linux Conference:
  http://embeddedlinuxconference.com/
  - Organized by the CE Linux Forum:
  - in California (San Francisco, April)
  - in Europe (October-November)
  - Very interesting kernel and userspace topics for embedded systems developers.
  - Presentation slides freely available
- Linux Plumbers: http://linuxplumbersconf.org
  - Conference on the low-level plumbing of Linux: kernel, audio, power management, device management, multimedia, etc.
- linux.conf.au: http://linux.org.au/conf/
  - In 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/

ARM resources

- ARM Linux project: `http://www.arm.linux.org.uk/`
  - Developer documentation: `http://www.arm.linux.org.uk/developer/`
  - linux-arm-kernel mailing list: `http://lists.infradead.org/mailman/listinfo/linux-arm-kernel`
  - FAQ: `http://www.arm.linux.org.uk/armlinux/mlfaq.php`
- Linaro: `http://linaro.org`
  - Many optimizations and resources for recent ARM CPUs (toolchains, kernels, debugging utilities...).
- ARM Limited: `http://www.linux-arm.com/`
  - Wiki with links to useful developer resources

Introduction to Git

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

What is Git?

- A version control system, like CVS, SVN, Perforce or ClearCase
- Originally developed for the Linux kernel development, now used by a large number of projects, including U-Boot, GNOME, Buildroot, uClibc and many more
- Contrary to CVS or SVN, Git is a distributed version control system
  - No central repository
  - Everybody has a local repository
  - Local branches are possible, and very important
  - Easy exchange of code between developers
  - Well-suited to the collaborative development model used in open-source projects

Install and Setup

- Git is available as a package in your distribution
  - `sudo apt-get install git`
- Everything is available through the git command
  - `git` has many commands, called using `git <command>`, where `<command>` can be `clone`, `checkout`, `branch`, etc.
  - Help can be found for a given command using `git help <command>`
- Setup your name and e-mail address
  - They will be referenced in each of your commits
  - `git config --global user.name 'My Name'`
  - `git config --global user.email me@mydomain.net`

Clone a Repository

- To start working on a project, you use Git's clone operation.
- With CVS or SVN, you would have used the checkout operation, to get a working copy of the project (latest version)
- With Git, you get a full copy of the repository, including the history, which allows to perform most of the operations offline.
- Cloning Linus Torvalds' Linux kernel repository
  `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
- `git://` is a special Git protocol. Most repositories can also be accessed using `http://`, but this is slower.
- After cloning, in `linux/`, you have the repository and a working copy of the master branch.

Explore the History

- `git log` will list all the commits. The latest commit is the first.

```
commit 4371ee353c3fc41aad9458b8e8e627eb508bc9a3
Author: Florian Fainelli <florian@openwrt.org>
Date: Mon Jun 1 02:43:17 2009 -0700

MAINTAINERS: take maintainership of the cpmac Ethernet driver

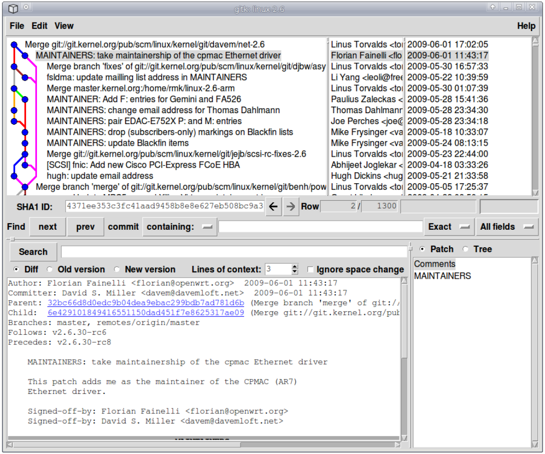
This patch adds me as the maintainer of the CPMAC (AR7)
Ethernet driver.

Signed-off-by: Florian Fainelli <florian@openwrt.org>
Signed-off-by: David S. Miller <davem@davemloft.net>
```

- `git log -p` will list the commits with the corresponding diff
- The history in Git is not linear like in CVS or SVN, but it is a graph of commits
  - Makes it a little bit more complicated to understand at the beginning
  - But this is what allows the powerful features of Git (distributed, branching, merging)

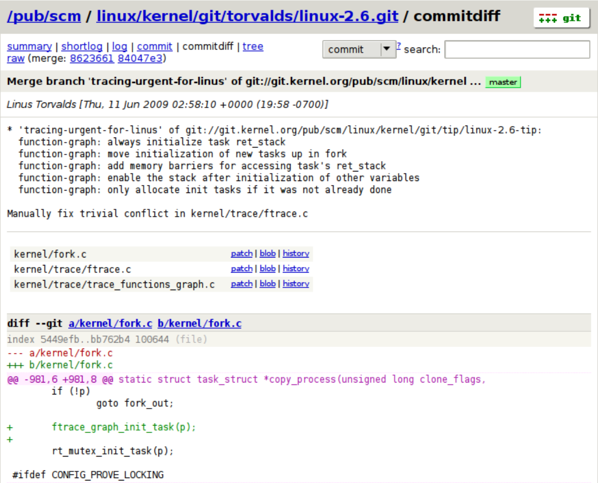
Visualize the History: gitk

- `gitk` is a graphical tool that represents the history of the current Git repository
- Can be installed from the `gitk` package



Visualize the History: gitweb

- Another great tool is the Web interface to Git. For the kernel, it is available at `http://git.kernel.org/`



Update your Repository

- The repository that has been cloned at the beginning will change over time
- Updating your local repository to reflect the changes of the remote repository will be necessary from time to time
- `git pull`
- Internally, does two things
  - Fetch the new changes from the remote repository (`git fetch`)
  - Merge them in the current branch (`git merge`)

Tags

- The list of existing tags can be found using
  - `git tag -l`
- To check out a working copy of the repository at a given tag
  - `git checkout <tagname>`
- To get the list of changes between a given tag and the latest available version
  - `git log v2.6.30..master`
- List of changes with diff on a given file between two tags
  - `git log -p v2.6.29..v2.6.30 MAINTAINERS`
- With gitk
  - `gitk v2.6.30..master`

Branches

- To start working on something, the best is to make a branch
  - It is local-only, nobody except you sees the branch
  - It is fast
  - It allows to split your work on different topics, try something and throw it away
  - It is cheap, so even if you think you're doing something small and quick, do a branch
- Unlike other version control systems, Git encourages the use of branches. Don't hesitate to use them.

Branches

- Create a branch
  - `git branch <branchname>`
- Move to this branch
  - `git checkout <branchname>`
- Both at once (create and switch to branch)
  - `git checkout -b <branchname>`
- List of local branches
  - `git branch`
- List of all branches, including remote branches
  - `git branch -a`

Making Changes

- Edit a file with your favorite text editor
- Get the status of your working copy
  - `git status`
- Git has a feature called the index, which allows you to stage your commits before committing them. It allows to commit only part of your modifications, by file or even by chunk.
- On each modified file
  - `git add <filename>`
- Then commit. No need to be on-line or connected to commit
  - Linux requires the -s option to sign your changes
  - `git commit -s`
- If all modified files should be part of the commit
  - `git commit -as`

Sharing Changes: E-mail

- The simplest way of sharing a few changes is to send patches by e-mail
- The first step is to generate the patches
  - `git format-patch -n master..<yourbranch>`
  - Will generate one patch for each of the commits done on `<yourbranch>`
  - The patch files will be `0001-....`, `0002-....`, etc.
- The second step is to send these patches by e-mail
  - `git send-email --compose --to email@domain.com 00*.patch`
  - Required Ubuntu package: `git-email`
  - In a later slide, we will see how to use git config to set the SMTP server, port, user and password.

Sharing Changes: Your Own Repository

- If you do a lot of changes and want to ease collaboration with others, the best is to have your own public repository
- Use a git hosting service on the Internet:
  - Gitorious (`https://gitorious.org/`)
    - Open Source server. Easiest. For public repositories.
  - GitHub (`https://github.com/`)
    - For public repositories. Have to pay for private repositories.
- Publish on your own web server
  - Easy to implement.
  - Just needs git software on the server and ssh access.
  - Drawback: only supports http cloning (less efficient)
- Set up your own git server
  - Most flexible solution.
  - Today's best solutions are `gitolite` (`https://github.com/sitaramc/gitolite`) for the server and `cgit` for the web interface (`http://hjemli.net/git/cgit/`).

Sharing changes: HTTP Hosting

- Create a bare version of your repository
  - `cd /tmp`
  - `git clone --bare ~/project project.git`
  - `touch project.git/git-daemon-export-ok`
- Transfer the contents of `project.git` to a publicly-visible place (reachable read-only by HTTP for everybody, and read-write by you through SSH)
- Tell people to clone `http://yourhost.com/path/to/project.git`
- Push your changes using
  - `git push ssh://yourhost.com/path/toproject.git srcbranch:destbranch`

Tracking Remote Trees

- In addition to the official Linus Torvalds tree, you might want to use other development or experimental trees
  - The OMAP tree at `git://git.kernel.org/pub/scm/linux/kernel/git/tmlind/linux-omap.git`
  - The stable realtime tree at `git://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git`
- The `git remote` command allows to manage remote trees
  - `git remote add rt git://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git`
- Get the contents of the tree
  - `git fetch rt`
- Switch to one of the branches
  - `git checkout rt/master`

Contribute to the Linux Kernel (1)

- Clone Linus Torvalds' tree:
  - `git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`
- Keep your tree up to date
  - `git pull`
- Look at the master branch and check whether your issue / change hasn't been solved / implemented yet. Also check the maintainer's git tree and mailing list (see the `MAINTAINERS` file).You may miss submissions that are not in mainline yet.
- If the maintainer has its own git tree, create a remote branch tracking this tree. This is much better than creating another clone (doesn't duplicate common stuff):
  - `git remote add linux-omap git://git.kernel.org/pub/scm/linux/kernel/git/tmlind/linux-omap.git`
  - `git fetch linux-omap`

Contribute to the Linux Kernel (2)

- Either create a new branch starting from the current commit in the master branch:
  - `git checkout -b feature`
- Or, if more appropriate, create a new branch starting from the maintainer's master branch:
  - `git checkout -b feature linux-omap/master` (remote tree / remote branch)
- In your new branch, implement your changes.
- Test your changes (must at least compile them).
- Run `git add` to add any new files to the index.
- Check that each file you modified is ready for submission:
  - `scripts/check_patch.pl --strict --file <file>`
- If needed, fix indenting rule violations:
  - `indent -linux <file>`

Configure git send-email

- Make sure you already have configured your name and e-mail address (should be done before the first commit).
  - `git config --global user.name 'My Name'`
  - `git config --global user.email me@mydomain.net`
- Configure your SMTP settings. Example for a Google Mail account:
  - `git config --global sendemail.smtpserver smtp.googlemail.com`
  - `git config --global sendemail.smtpserverport 587`
  - `git config --global sendemail.smtpencryption tls`
  - `git config --global sendemail.smtpuser jdoe@gmail.com`
  - `git config --global sendemail.smtppass xxx`

Contribute to the Linux Kernel (3)

- Group your changes by sets of logical changes, corresponding to the set of patches that you wish to submit.
- Commit and sign these groups of changes (signing required by Linux developers).
  - `git commit -s`
  - Make sure your first description line is a useful summary and starts with the name of the modified subsystem. This first description line will appear in your e-mails
- The easiest way is to look at previous commit summaries on the main file you modify
  - `git log --pretty=oneline <path-to-file>`
- Examples subject lines (`[PATCH]` omitted):

```
Documentation: prctl/seccomp_filter
PCI: release busn when removing bus
ARM: add support for xz kernel decompression
```

Contribute to the Linux Kernel (4)

- Remove previously generated patches
  - `rm 00*.patch`
- Have git generate patches corresponding to your branch
  - If your branch is based on mainline
    - `git format-patch master..<your branch>`
  - If your branch is based on a remote branch
    - `git format-patch <remote>/<branch>..<your branch>`
- You can run a last check on all your patches (easy)
  - `scripts/check_patch.pl --strict 00*.patch`
- Now, send your patches to yourself
  - `git send-email --compose --to me@mydomain.com 00*.patch`
- If you have just one patch, or a trivial patch, you can remove the empty line after `In-Reply-To:`. This way, you won't add a summary e-mail introducing your changes (recommended otherwise).

Contribute to the Linux Kernel (5)

- Check that you received your e-mail properly, and that it looks good.
- Now, find the maintainers for your patches

```
scripts/get_maintainer.pl ~/patches/00*.patch
Russell King <linux@arm.linux.org.uk> (maintainer:ARM PORT)
Nicolas Pitre <nicolas.pitre@linaro.org>
(commit_signer:1/1=100%)
linux-arm-kernel@lists.infradead.org (open list:ARM PORT)
linux-kernel@vger.kernel.org (open list)
```

- Now, send your patches to each of these people and lists
  - `git send-email --compose --to linux@arm.linux.org.uk --to nicolas.pitre@linaro.org --to linux-arm-kernel@lists.infradead.org --to linux-kernel@vger.kernel.org 00*.patch`
- Wait for replies about your changes, take the comments into account, and resubmit if needed, until your changes are eventually accepted.

Contribute to the Linux Kernel (6)

- If you use `git format-patch` to produce your patches, you will need to update your branch and may need to group your changes in a different way (one patch per commit).
- Here's what we recommend
  - Update your master branch
    - `git checkout master; git pull`
  - Back to your branch, implement the changes taking community feedback into account. Commit these changes.
  - Still in your branch: reorganize your commits and commit messages
    - `git rebase --interactive origin/master`
    - `git rebase` allows to rebase (replay) your changes starting from the latest commits in master. In interactive mode, it also allows you to merge, edit and even reorder commits, in an interactive way.
  - Third, generate the new patches with `git format-patch`.

About Git

- We have just seen the very basic features of Git.
- A lot more interesting features are available (rebasing, bisection, merging and more)
- References
  - Git Manual
    - `http://schacon.github.com/git/user-manual.html`
  - Git Book
    - `http://book.git-scm.com/`
  - Git official website
    - `http://git-scm.com/`
  - Video: James Bottomley's tutorial on using Git
    - `http://free-electrons.com/pub/video/2008/ols/ols2008-james-bottomley-git.ogg`

Practical lab - Git

- Get familiar with git by contributing to a real project: the Linux kernel
- Send your patches to the maintainers and mailing lists.

Last slides

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni
Free Electrons

Corrections, suggestions, contributions and translations are welcome!

- Clean up files that are easy to retrieve, remove downloads.
- Generate an archive of your lab directory.

Evaluation form

Please take a few minutes to rate this training session, by answering our on-line survey:

`http://free-electrons.com/doc/training/linux-kernel/survey.html`

Thank you!

And may the Source be with you