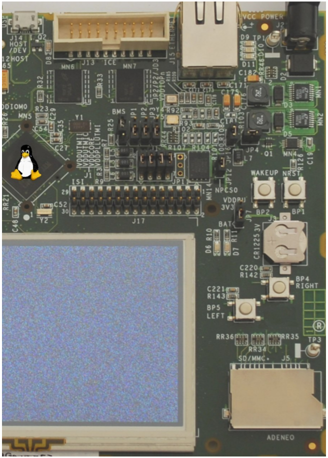
Linux Kernel and Android Development Class

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded

Latest update: December 5, 2012.
Document updates and sources:
`http://adeneo-embedded.com`
Corrections, suggestions, contributions and translations are welcome!



Rights to copy

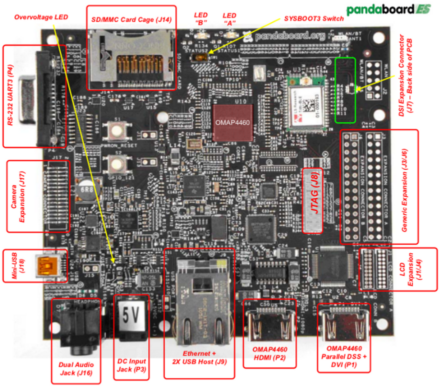
Welcome!

- Audience: embedded software students
- Purpose of this course: development environment setup, drivers and kernel development, image building
- Prerequisites: basic knowledge of Linux, good knowledge of C, basic knowledge of OS
  - Agenda
    Course 1 : Linux kernel - Principles and deployment on embedded platforms (2 hours)
    Course 2 : Android - Principles and architecture (2 hours)
    Course 3 : Linux - Driver development (2 hours)
- Targeted hardware platform : PandaBoard ES

Who is Adeneo Embedded?

- BSP and driver development
- Hardware Design and design reviews
- Systems optimization
- Embedded application development
- Support contract
- Training and Workshop
- Consulting and engineering services

PandaBoard ES

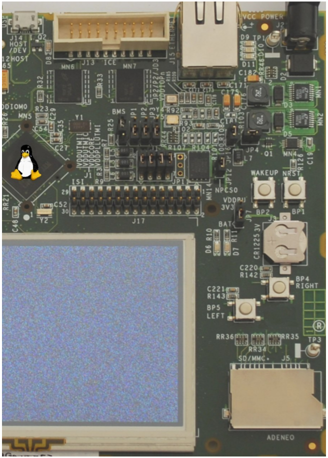


- Core Architecture: ARM
- Core Sub-Architecture: Cortex-A9 dual-core
- OMAP4460
- kit: ES Board revision B1, μSD card and Adaptor
- features: High-Speed USB 2.0 OTG Port, Stereo Audio Out/In, Ethernet, HDMI, DVI, Camera I/F
- clock max: 1.5 GHz

Linux Kernel Introduction

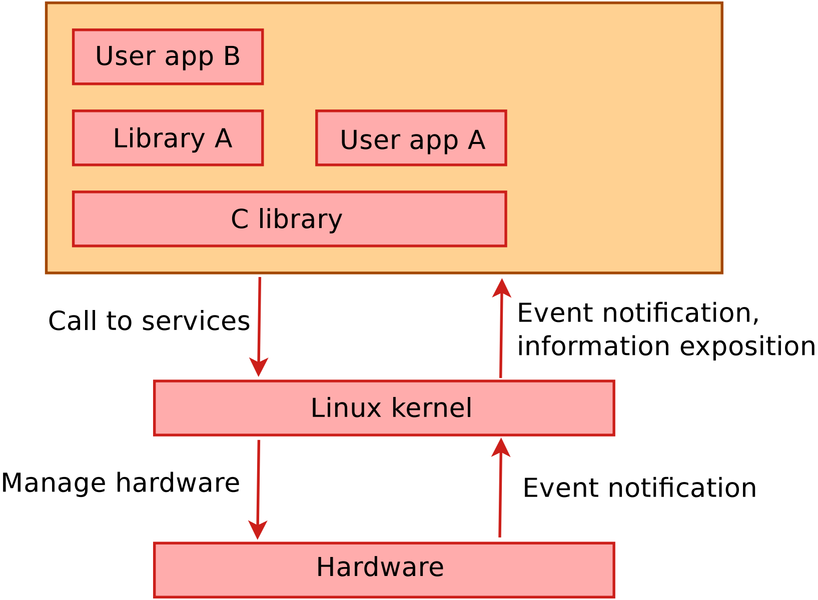
Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded

Corrections, suggestions, contributions and translations are welcome!



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

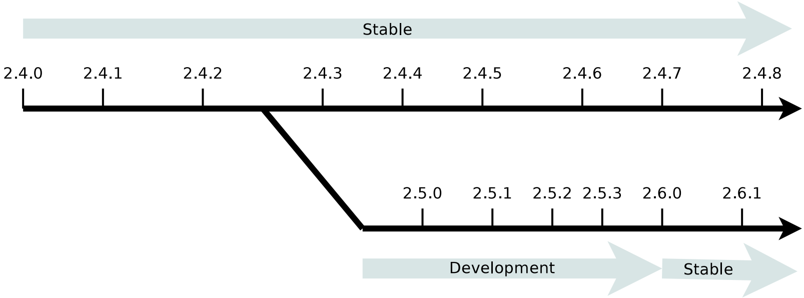
- Linux makes system and kernel information available in user-space through virtual filesystems.
- Virtual filesystems allow applications to see directories and files that do not exist on any real storage: they are created on the fly by the kernel
- The two most important virtual filesystems are
  - `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)

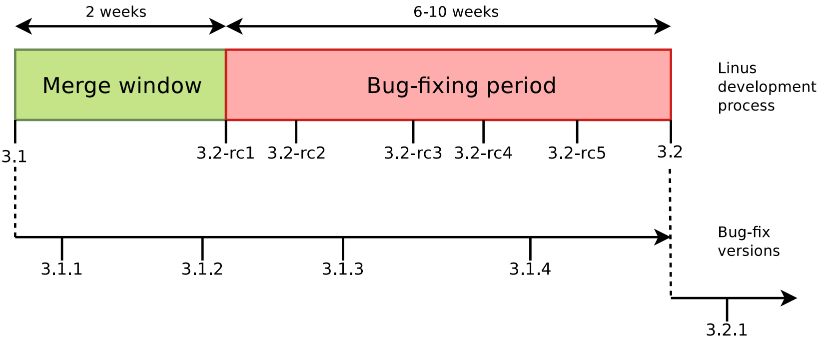
- Since `2.6.0`, kernel developers have been able to introduce lots of new features one by one on a steady pace, without having to make major changes in existing subsystems.
- So far, there was no need to create a new development branch (such as `2.7`), which would massively break compatibility with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

mainline:	**3.5-rc4**
stable:	**3.4.4**
stable:	**3.3.8 (EOL)**
stable:	**3.2.21**
stable:	**3.1.10 (EOL)**
stable:	**3.0.36**
stable:	**2.6.35.13**
stable:	**2.6.34.12**
stable:	**2.6.32.59**
stable:	**2.6.27.62**
linux-next:	**next-20120625**

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?

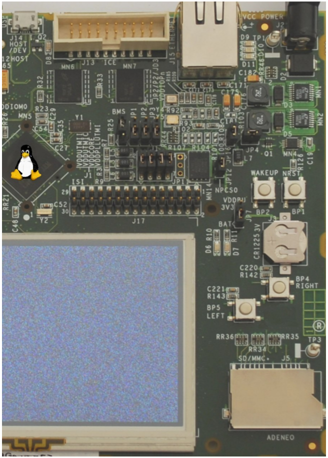
- The official list of changes for each Linux release is just a huge list of individual patches!

```
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

Embedded Linux Kernel Usage

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



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

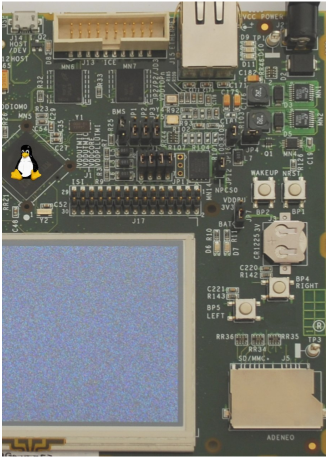
Linux patches...

- 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, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



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, the GNU C Compiler
  - Many gcc specific extensions used in the kernel code, any ANSI C compiler will not compile the kernel
  - A few alternate compilers are supported (Intel and Marvell)
  - See `http://gcc.gnu.org/onlinedocs/gcc-4.6.1/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

```
C symbol: request_irq

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

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

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

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

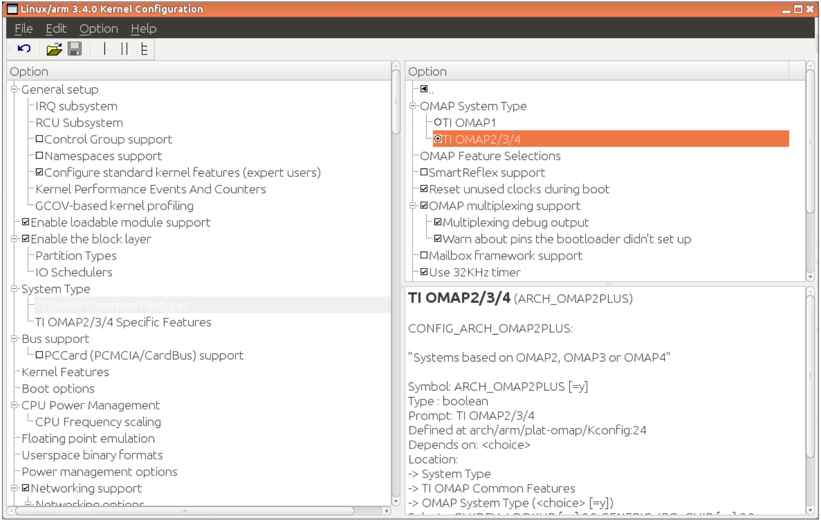
Kernel option dependencies

- 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`

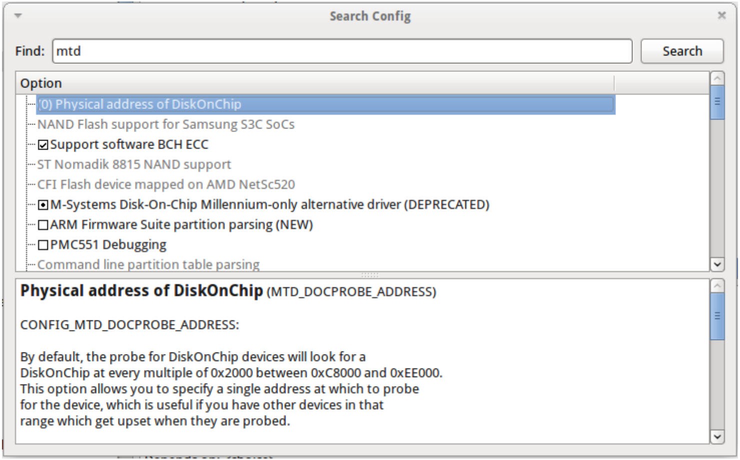
- The most common graphical interface to configure the kernel.
- Make sure you read
  `help -> introduction: useful options!`
- File browser: easier to load configuration files
- Search interface to look for parameters
- Required Debian / Ubuntu packages: `libqt4-dev g++` (`libqt3-mt-dev` for older kernel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

make xconfig screenshot



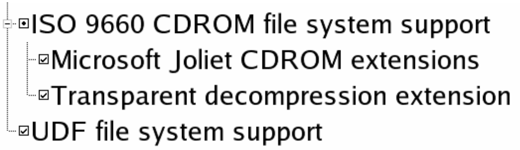
make xconfig search interface

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



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`

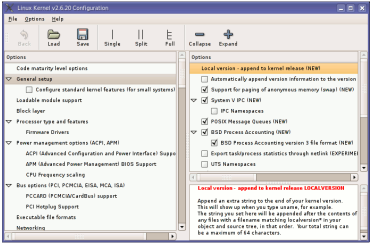
Corresponding .config file excerpt

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`

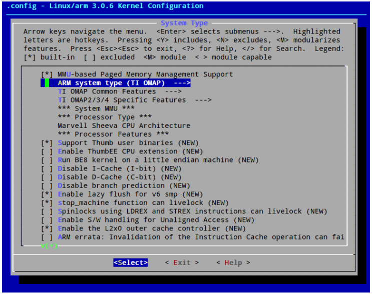
- *GTK* based graphical configuration interface. Functionality similar to that of make `xconfig`.
- Just lacking a search functionality.
- Required Debian packages: `libglade2-dev`



make menuconfig

`make menuconfig`

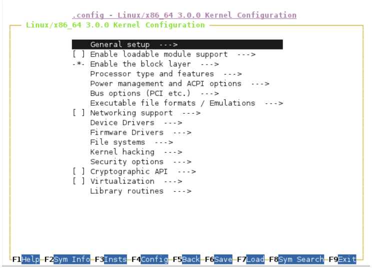
- Useful when no graphics are available. Pretty convenient too!
- Same interface found in other tools: BusyBox, Buildroot...
- Required Debian packages: `libncurses-dev`



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`

- 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.

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)

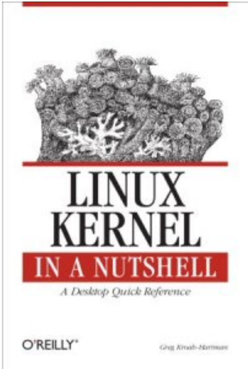
- `sudo rmmod <module_name>`
  Tries to remove the given module.
  Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)
- `sudo modprobe -r <module_name>`
  Tries to remove the given module and all dependent modules (which are no longer needed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

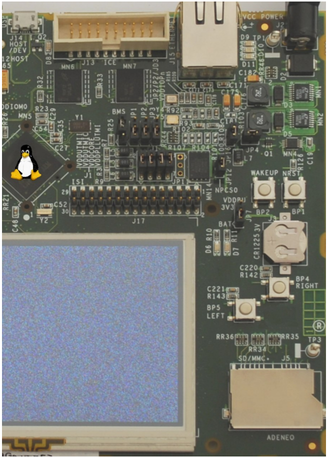
- By Greg Kroah-Hartman, O'Reilly
  `http://www.kroah.com/lkn/`
- A good reference book and guide on configuring, compiling and managing the Linux kernel sources.
- Freely available on-line!
  Great companion to the printed book for easy electronic searches!
  Available as single PDF file on
  `http://free-electrons.com/community/kernel/lkn/`
- Our rating: 2 stars



Bootloaders

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier

Free Electrons, Adeneo Embedded



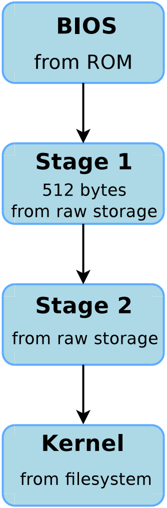
Boot Sequence

Bootloaders

- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel, from flash storage, from the network, or from another type of non-volatile storage.
  - Possibly decompression of the application binary
  - Execution of the application
- Besides these basic functions, most bootloaders provide a shell with various commands implementing different operations.
  - Loading of data from storage or network, memory inspection, hardware diagnostics and testing, etc.

Bootloaders on x86 (1)

- The x86 processors are typically bundled on a board with a non-volatile memory containing a program, the BIOS.
- This program gets executed by the CPU after reset, and is responsible for basic hardware initialization and loading of a small piece of code from non-volatile storage.
  - This piece of code is usually the first 512 bytes of a storage device
- This piece of code is usually a 1st stage bootloader, which will load the full bootloader itself.
- The bootloader can then offer all its features. It typically understands filesystem formats so that the kernel file can be loaded directly from a normal filesystem.

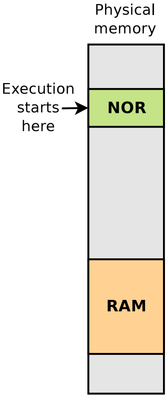


Bootloaders on x86 (2)

- GRUB, Grand Unified Bootloader, the most powerful one.
  `http://www.gnu.org/software/grub/`
  - Can read many filesystem formats to load the kernel image and the configuration, provides a powerful shell with various commands, can load kernel images over the network, etc.
  - See our dedicated presentation for details:
    `http://free-electrons.com/docs/grub/`
- Syslinux, for network and removable media booting (USB key, CD-ROM)
  `http://www.kernel.org/pub/linux/utils/boot/syslinux/`

Booting on embedded CPUs: case 1

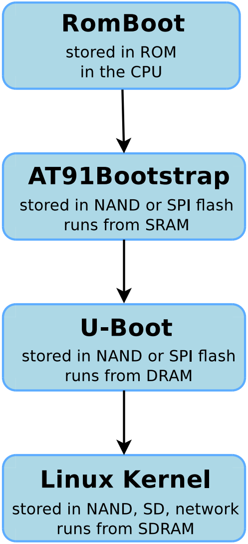
- When powered, the CPU starts executing code at a fixed address
- There is no other booting mechanism provided by the CPU
- The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU starts executing instructions
- The first stage bootloader must be programmed at this address in the NOR
- NOR is mandatory, because it allows random access, which NAND doesn't allow
- **Not very common anymore** (unpractical, and requires NOR flash)



Booting on embedded CPUs: case 2

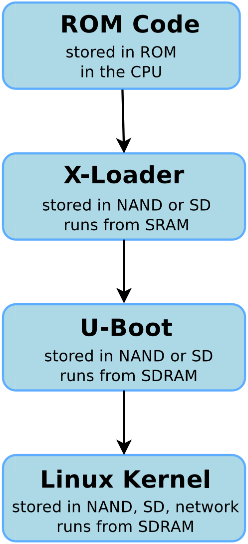
- The CPU has an integrated boot code in ROM
  - BootROM on AT91 CPUs, “ROM code” on OMAP, etc.
  - Exact details are CPU-dependent
- This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet)
  - Storage device can typically be: MMC, NAND, SPI flash, UART, etc.
- The first stage bootloader is
  - Limited in size due to hardware constraints (SRAM size)
  - Provided either by the CPU vendor or through community projects
- This first stage bootloader must initialize DRAM and other hardware devices and load a second stage bootloader into RAM

Booting on ARM Atmel AT91



- **RomBoot**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 4 KB. No user interaction possible in standard boot mode.
- **AT91Bootstrap**: runs from SRAM. Initializes the DRAM, the NAND or SPI controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible.
- **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Booting on ARM OMAP3



- **ROM Code**: tries to find a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a configuration header). Size limited to $<$64 KB. No user interaction possible.
- **X-Loader**: runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible. File called `MLO`.
- **U-Boot**: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called `u-boot.bin`.
- **Linux Kernel**: runs from RAM. Takes over the system completely (bootloaders no longer exists).

Generic bootloaders for embedded CPUs

- We will focus on the generic part, the main bootloader, offering the most important features.
- There are several open-source generic bootloaders. Here are the most popular ones:
  - **U-Boot**, the universal bootloader by Denx
    The most used on ARM, also used on PPC, MIPS, x86, m68k, NIOS, etc. The de-facto standard nowadays. We will study it in detail.
    `http://www.denx.de/wiki/U-Boot`
  - **Barebox**, a new architecture-neutral bootloader, written as a successor of U-Boot. Better design, better code, active development, but doesn't yet have as much hardware support as U-Boot.
    `http://www.barebox.org`
- There are also a lot of other open-source or proprietary bootloaders, often architecture-specific
  - RedBoot, Yaboot, PMON, etc.

The U-boot bootloader

U-Boot

U-Boot is a typical free software project

- Freely available at `http://www.denx.de/wiki/U-Boot`
- Documentation available at `http://www.denx.de/wiki/U-Boot/Documentation`
- The latest development source code is available in a Git repository: `http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=summary`
- Development and discussions happen around an open mailing-list `http://lists.denx.de/pipermail/u-boot/`
- Since the end of 2008, it follows a fixed-interval release schedule. Every two months, a new version is released. Versions are named `YYYY.MM`.

U-Boot configuration

- Get the source code from the website, and uncompress it
- The `include/configs/` directory contains one configuration file for each supported board
  - It defines the CPU type, the peripherals and their configuration, the memory mapping, the U-Boot features that should be compiled in, etc.
  - It is a simple `.h` file that sets C pre-processor constants. See the `README` file for the documentation of these constants.
- Assuming that your board is already supported by U-Boot, there should be one file corresponding to your board, for example `include/configs/igep0020.h`
- This file can also be adjusted to add or remove features from U-Boot

U-Boot configuration file excerpt

```
/* CPU configuration */
#define CONFIG_ARMV7 1
#define CONFIG_OMAP 1
#define CONFIG_OMAP34XX 1
#define CONFIG_OMAP3430 1
#define CONFIG_OMAP3_IGEP0020 1
[...]
/* Memory configuration */
#define CONFIG_NR_DRAM_BANKS 2
#define PHYS_SDRAM_1 OMAP34XX_SDRC_CS0
#define PHYS_SDRAM_1_SIZE (32 <<  20)
#define PHYS_SDRAM_2 OMAP34XX_SDRC_CS1
[...]
/* USB configuration */
#define CONFIG_MUSB_UDC 1
#define CONFIG_USB_OMAP3 1
#define CONFIG_TWL4030_USB 1
[...]
```

```
/* Available commands and features */
#define CONFIG_CMD_CACHE
#define CONFIG_CMD_EXT2
#define CONFIG_CMD_FAT
#define CONFIG_CMD_I2C
#define CONFIG_CMD_MMC
#define CONFIG_CMD_NAND
#define CONFIG_CMD_NET
#define CONFIG_CMD_DHCP
#define CONFIG_CMD_PING
#define CONFIG_CMD_NFS
#define CONFIG_CMD_MTDPARTS
[...]
```

Configuring and compiling U-Boot

- U-Boot must be configured before being compiled
  - `make BOARDNAME_config`
  - Where `BOARDNAME` is usually the name of the configuration file in `include/configs/`, without `.h`
- Make sure that the cross-compiler is available in `PATH`
- Compile U-Boot, by specifying the cross-compiler prefix. Example, if your cross-compiler executable is `arm-linux-gcc`:
  `make CROSS_COMPILE=arm-linux-`
- The result is a `u-boot.bin` file, which is the U-Boot image

Installing U-Boot

- U-Boot must usually be installed in flash memory to be executed by the hardware. Depending on the hardware, the installation of U-Boot is done in a different way:
  - The CPU provides some kind of specific boot monitor with which you can communicate through serial port or USB using a specific protocol
  - The CPU boots first on removable media (MMC) before booting from fixed media (NAND). In this case, boot from MMC to reflash a new version
  - U-Boot is already installed, and can be used to flash a new version of U-Boot. However, be careful: if the new version of U-Boot doesn't work, the board is unusable
  - The board provides a JTAG interface, which allows to write to the flash memory remotely, without any system running on the board. It also allows to rescue a board if the bootloader doesn't work.

U-boot prompt

- Connect the target to the host through a serial console
- Power-up the board. On the serial console, you will see something like:

```
U-Boot 2011.12 (May 04 2012 - 10:31:05)
OMAP36XX/37XX-GP ES1.2, CPU-OPP2, L3-165MHz, Max CPU Clock 1 Ghz
IGEP v2 board + LPDDR/ONENAND
I2C:   ready
DRAM:  512 MiB
NAND:  512 MiB
MMC:   OMAP SD/MMC: 0
[...]
Net:   smc911x-0
Hit any key to stop autoboot:  0
U-Boot #
```

- The U-Boot shell offers a set of commands. We will study the most important ones, see the documentation for a complete reference or the `help` command.

Information commands

Flash information (NOR and SPI flash)

```
U-Boot> flinfo
DataFlash:AT45DB021
Nb pages: 1024
Page Size: 264
Size= 270336 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C0001FFF (RO) Bootstrap
Area 1: C0002000 to C0003FFF Environment
Area 2: C0004000 to C0041FFF (RO) U-Boot
```

NAND flash information

```
U-Boot> nand info
Device 0: NAND 256MiB 3,3V 8-bit, sector size 128 KiB
```

Version details

```
U-Boot> version
U-Boot 2009.08 (Nov 15 2009 - 14:48:35)
```

Those details will vary from one board to the other (according to the U-Boot configuration and hardware devices)

Important commands (1)

- The exact set of commands depends on the U-Boot configuration
- `help` and `help command`
- `boot`, runs the default boot command, stored in `bootcmd`
- `bootm <address>` , starts a kernel image loaded at the given address in RAM
- `ext2load`, loads a file from an ext2 filesystem to RAM
  - And also `ext2ls` to list files, `ext2info` for information
- `fatload`, loads a file from a FAT filesystem to RAM
  - And also `fatls` and `fatinfo`
- `tftp`, loads a file from the network to RAM
- `ping`, to test the network

Important commands (2)

- `loadb`, `loads`, `loady`, load a file from the serial line to RAM
- `usb`, to initialize and control the USB subsystem, mainly used for USB storage devices such as USB keys
- `mmc`, to initialize and control the MMC subsystem, used for SD and microSD cards
- `nand`, to erase, read and write contents to NAND flash
- `erase`, `protect`, `cp`, to erase, modify protection and write to NOR flash
- `md`, displays memory contents. Can be useful to check the contents loaded in memory, or to look at hardware registers.
- `mm`, modifies memory contents. Can be useful to modify directly hardware registers, for testing purposes.

Environment variables commands

- U-Boot can be configured through environment variables, which affect the behavior of the different commands.
- Environment variables are loaded from flash to RAM at U-Boot startup, can be modified and saved back to flash for persistence
- There is a dedicated location in flash to store U-Boot environment, defined in the board configuration file
- Commands to manipulate environment variables:
  - `printenv`, shows all variables
  - `printenv <variable-name>`, shows the value of one variable
  - `setenv <variable-name> <variable- value>`, changes the value of a variable, only in RAM
  - `saveenv`, saves the current state of the environment to flash

Environment variables commands (2)

```
u-boot # printenv
baudrate=19200
ethaddr=00:40:95:36:35:33
netmask=255.255.255.0
ipaddr=10.0.0.11
serverip=10.0.0.1
stdin=serial
stdout=serial
stderr=serial
u-boot # printenv serverip
serverip=10.0.0.2
u-boot # setenv serverip 10.0.0.100
u-boot # saveenv
```

Important U-Boot env variables

- `bootcmd`, contains the command that U-Boot will automatically execute at boot time after a configurable delay, if the process is not interrupted
- `bootargs`, contains the arguments passed to the Linux kernel, covered later
- `serverip`, the IP address of the server that U-Boot will contact for network related commands
- `ipaddr`, the IP address that U-Boot will use
- `netmask`, the network mask to contact the server
- `ethaddr`, the MAC address, can only be set once
- `bootdelay`, the delay in seconds before which U-Boot runs `bootcmd`
- `autostart`, if yes, U-Boot starts automatically an image that has been loaded into memory

Scripts in environment variables

- Environment variables can contain small scripts, to execute several commands and test the results of commands.
  - Useful to automate booting or upgrade processes
  - Several commands can be chained using the `;` operator
  - Tests can be done using
    `if command ; then ... ; else ... ; fi`
  - Scripts are executed using `run <variable-name>`
  - You can reference other variables using `${variable-name}`
- Example
  - `setenv mmc-boot 'mmc init 0; if fatload mmc 0 80000000 boot.ini; then source; else if fatload mmc 0 80000000 uImage; then run mmc-bootargs; bootm; fi; fi'`

Transferring files to the target

- U-Boot is mostly used to load and boot a kernel image, but it also allows to change the kernel image and the root filesystem stored in flash.
- Files must be exchanged between the target and the development workstation. This is possible:
  - Through the network if the target has an Ethernet connection, and U-Boot contains a driver for the Ethernet chip. This is the fastest and most efficient solution.
  - Through a USB key, if U-Boot support the USB controller of your platform
  - Through a SD or microSD card, if U-Boot supports the MMC controller of your platform
  - Through the serial port

TFTP

- Network transfer from the development workstation and U-Boot on the target takes place through TFTP
  - *Trivial File Transfer Protocol*
  - Somewhat similar to FTP, but without authentication and over UDP
- A TFTP server is needed on the development workstation
  - `sudo apt-get install tftpd-hpa`
  - All files in `/var/lib/tftpboot` are then visible through TFTP
  - A TFTP client is available in the `tftp-hpa` package, for testing
- A TFTP client is integrated into U-Boot
  - Configure the `ipaddr` and `serverip` environment variables
  - Use `tftp <address> <filename>` to load a file

U-boot mkimage

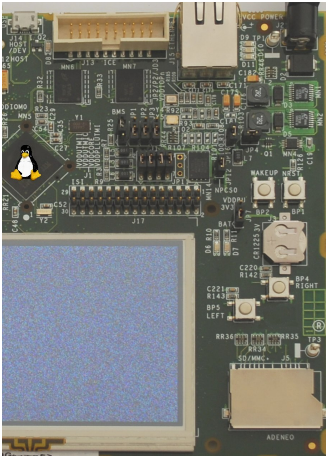
- The kernel image that U-Boot loads and boots must be prepared, so that a U-Boot specific header is added in front of the image
  - This header gives details such as the image size, the expected load address, the compression type, etc.
- This is done with a tool that comes in U-Boot, `mkimage`
- Debian / Ubuntu: just install the `u-boot-tools` package.
- Or, compile it by yourself: simply configure U-Boot for any board of any architecture and compile it. Then install `mkimage`:

  ```
  cp tools/mkimage /usr/local/bin/
  ```
- The special target `uImage` of the kernel Makefile can then be used to generate a kernel image suitable for U-Boot.

Linux Root Filesystem

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded

Corrections, suggestions, contributions and translations are welcome!



Principle and solutions

Filesystems

- Filesystems are used to organize data in directories and files on storage devices or on the network. The directories and files are organized as a hierarchy
- In Unix systems, applications and users see a **single global hierarchy** of files and directories, which can be composed of several filesystems.
- Filesystems are **mounted** in a specific location in this hierarchy of directories
  - When a filesystem is mounted in a directory (called *mount point*), the contents of this directory reflects the contents of the storage device
  - When the filesystem is unmounted, the *mount point* is empty again.
- This allows applications to access files and directories easily, regardless of their exact storage location

Filesystems (2)

- Create a mount point, which is just a directory

```
$ mkdir /mnt/usbkey
```

- It is empty

```
$ ls /mnt/usbkey
$
```

- Mount a storage device in this mount point

```
$ mount -t vfat /dev/sda1 /mnt/usbkey
$
```

- You can access the contents of the USB key

```
$ ls /mnt/usbkey
docs prog.c picture.png movie.avi
$
```

mount / umount

- `mount` allows to mount filesystems
  - `mount -t type device mountpoint`
  - `type` is the type of filesystem
  - `device` is the storage device, or network location to mount
  - `mountpoint` is the directory where files of the storage device or network location will be accessible
  - `mount` with no arguments shows the currently mounted filesystems
- `umount` allows to unmount filesystems
  - This is needed before rebooting, or before unplugging a USB key, because the Linux kernel caches writes in memory to increase performances. umount makes sure that those writes are committed to the storage.

Root filesystem

- A particular filesystem is mounted at the root of the hierarchy, identified by `/`
- This filesystem is called the **root filesystem**
- As `mount` and `umount` are programs, they are files inside a filesystem.
  - They are not accessible before mounting at least one filesystem.
- As the root filesystem is the first mounted filesystem, it cannot be mounted with the normal `mount` command
- It is mounted directly by the kernel, according to the `root=` kernel option
- When no root filesystem is available, the kernel panics

```
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)
```

Location of the root filesystem

- It can be mounted from different locations
  - From the partition of a hard disk
  - From the partition of a USB key
  - From the partition of an SD card
  - From the partition of a NAND flash chip or similar type of storage device
  - From the network, using the NFS protocol
  - From memory, using a pre-loaded filesystem (by the bootloader)
  - etc.
- It is up to the system designer to choose the configuration for the system, and configure the kernel behaviour with `root=`

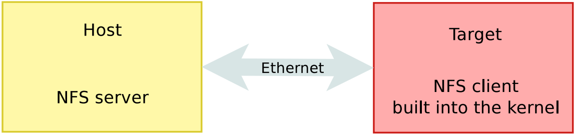
Mounting rootfs from storage devices

- Partitions of a hard disk or USB key
  - `root=/dev/sdXY`, where `X` is a letter indicating the device, and `Y` a number indicating the partition
  - `/dev/sdb2` is the second partition of the second disk drive (either USB key or ATA hard drive)
- Partitions of an SD card
  - `root=/dev/mmcblkXpY`, where `X` is a number indicating the device and `Y` a number indicating the partition
  - `/dev/mmcblk0p2` is the second partition of the first device
- Partitions of flash storage
  - `root=/dev/mtdblockX`, where `X` is the partition number
  - `/dev/mtdblock3` is the fourth partition of a NAND flash chip (if only one NAND flash chip is present)

Mounting rootfs over the network (1)

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

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



Mounting rootfs over the network (2)

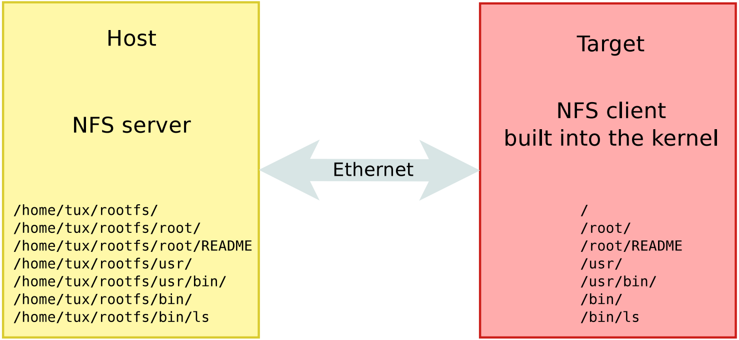
On the development workstation side, a NFS server is needed

- Install an NFS server (example: Debian, Ubuntu)
  `sudo apt-get install nfs-kernel-server`
- Add the exported directory to your `/etc/exports` file:
  `/home/tux/rootfs 192.168.1.111(rw,no_root_squash, no_subtree_check)`
  - `192.168.1.111` is the client IP address
  - `rw,no_root_squash,no_subtree_check` are the NFS server options for this directory export.
- Start or restart your NFS server (example: Debian, Ubuntu)
  `sudo /etc/init.d/nfs-kernel-server restart`

Mounting rootfs over the network (3)

- On the target system
- The kernel must be compiled with
  - `CONFIG_NFS_FS=y` (NFS support)
  - `CONFIG_IP_PNP=y` (configure IP at boot time)
  - `CONFIG_ROOT_NFS=y` (support for NFS as rootfs)
- The kernel must be booted with the following parameters:
  - `root=/dev/nfs` (we want rootfs over NFS)
  - `ip=192.168.1.111` (target IP address)
  - `nfsroot=192.168.1.110:/home/tux/rootfs/` (NFS server details)

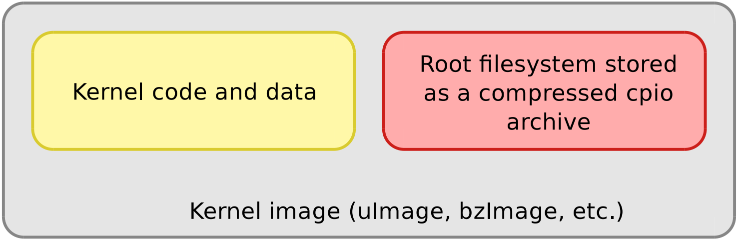
Mounting rootfs over the network (4)



rootfs in memory: initramfs (1)

- It is also possible to have the root filesystem integrated into the kernel image
- It is therefore loaded into memory together with the kernel
- This mechanism is called **initramfs**
  - It integrates a compressed archive of the filesystem into the kernel image
- It is useful for two cases
  - Fast booting of very small root filesystems. As the filesystem is completely loaded at boot time, application startup is very fast.
  - As an intermediate step before switching to a real root filesystem, located on devices for which drivers not part of the kernel image are needed (storage drivers, filesystem drivers, network drivers). This is always used on the kernel of desktop/server distributions to keep the kernel image size reasonable.

rootfs in memory: initramfs (2)



rootfs in memory: initramfs (3)

- The contents of an initramfs are defined at the kernel configuration level, with the `CONFIG_INITRAMFS_SOURCE` option
  - Can be the path to a directory containing the root filesystem contents
  - Can be the path to a cpio archive
  - Can be a text file describing the contents of the initramfs (see documentation for details)
- The kernel build process will automatically take the contents of the `CONFIG_INITRAMFS_SOURCE` option and integrate the root filesystem into the kernel image
- `Documentation/filesystems/ramfs-rootfs-initramfs.txt`
  `Documentation/early-userspace/README`

Contents

Root filesystem organization

- The organization of a Linux root filesystem in terms of directories is well-defined by the **Filesystem Hierarchy Standard**
- `http://www.linuxfoundation.org/collaborate/workgroups/lsb/fhs`
- Most Linux systems conform to this specification
  - Applications expect this organization
  - It makes it easier for developers and users as the filesystem organization is similar in all systems

Important directories (1)

- **/bin** Basic programs
- **/boot** Kernel image (only when the kernel is loaded from a filesystem, not common on non-x86 architectures)
- **/dev** Device files (covered later)
- **/etc** System-wide configuration
- **/home** Directory for the users home directories
- **/lib** Basic libraries
- **/media** Mount points for removable media
- **/mnt** Mount points for static media
- **/proc** Mount point for the proc virtual filesystem

Important directories (2)

- **/root** Home directory of the `root` user
- **/sbin** Basic system programs
- **/sys** Mount point of the sysfs virtual filesystem
- **/tmp** Temporary files
- **/usr**
  - **/usr/bin** Non-basic programs
  - **/usr/lib** Non-basic libraries
  - **/usr/sbin** Non-basic system programs
- **/var** Variable data files. This includes spool directories and files, administrative and logging data, and transient and temporary files

Separation of programs and libraries

- Basic programs are installed in `/bin` and `/sbin` and basic libraries in `/lib`
- All other programs are installed in `/usr/bin` and `/usr/sbin` and all other libraries in `/usr/lib`
- In the past, on Unix systems, `/usr` was very often mounted over the network, through NFS
- In order to allow the system to boot when the network was down, some binaries and libraries are stored in `/bin`, `/sbin` and `/lib`
- `/bin` and `/sbin` contain programs like `ls`, `ifconfig`, `cp`, `bash`, etc.
- `/lib` contains the C library and sometimes a few other basic libraries
- All other programs and libraries are in `/usr`

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`

Virtual Filesystems

proc virtual filesystem

- The `proc` virtual filesystem exists since the beginning of Linux
- It allows
  - The kernel to expose statistics about running processes in the system
  - The user to adjust at runtime various system parameters about process management, memory management, etc.
- The `proc` filesystem is used by many standard userspace applications, and they expect it to be mounted in `/proc`
- Applications such as `ps` or `top` would not work without the `proc` filesystem
- Command to mount `/proc`:
  `mount -t proc nodev /proc`
- `Documentation/filesystems/proc.txt` in the kernel sources
- `man proc`

proc contents

- One directory for each running process in the system
  - `/proc/<pid>`
  - `cat /proc/3840/cmdline`
  - It contains details about the files opened by the process, the CPU and memory usage, etc.
- `/proc/interrupts`, `/proc/devices`, `/proc/iomem`, `/proc/ioports` contain general device-related information
- `/proc/cmdline` contains the kernel command line
- `/proc/sys` contains many files that can be written to to adjust kernel parameters
  - They are called *sysctl*. See `Documentation//latest/sysctl/` in kernel sources.
  - Example
    `echo 3 > /proc/sys/vm/drop_caches`

sysfs filesystem

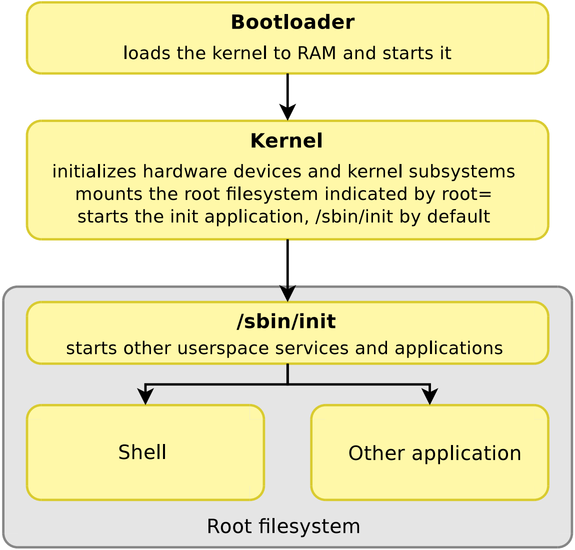
- The `sysfs` filesystem is a feature integrated in the 2.6 Linux kernel
- It allows to represent in userspace the vision that the kernel has of the buses, devices and drivers in the system
- It is useful for various userspace applications that need to list and query the available hardware, for example `udev` or `mdev`.
- All applications using sysfs expect it to be mounted in the `/sys` directory
- Command to mount `/sys`:
  `mount -t sysfs nodev /sys`
- ```
  $ ls /sys/
  block bus class dev devices firmware
  fs kernel modulepower
  ```
  ```

Minimal filesystem

Basic applications

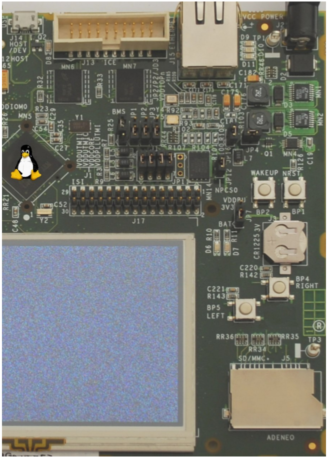
- In order to work, a Linux system needs at least a few applications
- An `init` application, which is the first userspace application started by the kernel after mounting the root filesystem
  - The kernel tries to run `/sbin/init`, `/bin/init`, `/etc/init` and `/bin/sh`.
  - If none of them are found, the kernel panics and the boot process is stopped.
  - The init application is responsible for starting all other userspace applications and services
- Usually a shell, to allow a user to interact with the system
- Basic Unix applications, to copy files, move files, list files (commands like `mv`, `cp`, `mkdir`, `cat`, etc.)
- Those basic components have to be integrated into the root filesystem to make it usable

Overall booting process



Busybox

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



Why Busybox?

- A Linux system needs a basic set of programs to work
  - An init program
  - A shell
  - Various basic utilities for file manipulation and system configuration
- In normal Linux systems, those programs are provided by different projects
  - `coreutils`, `bash`, `grep`, `sed`, `tar`, `wget`, `modutils`, etc. are all different projects
  - A lot of different components to integrate
  - Components not designed with embedded systems constraints in mind: they are not very configurable and have a wide range of features
- Busybox is an alternative solution, extremely common on embedded systems

General purpose toolbox: BusyBox

- Rewrite of many useful Unix command line utilities
  - Integrated into a single project, which makes it easy to work with
  - Designed with embedded systems in mind: highly configurable, no unnecessary features
- All the utilities are compiled into a single executable, `/bin/busybox`
  - Symbolic links to `/bin/busybox` are created for each application integrated into Busybox
- For a fairly featureful configuration, less than 500 KB (statically compiled with uClibc) or less than 1 MB (statically compiled with glibc).
- `http://www.busybox.net/`

BusyBox commands!

Commands available in BusyBox 1.13

[, [[, addgroup, adduser, adjtimex, ar, arp, arping, ash, awk, basename, bbconfig, bbsh, brctl, bunzip2, busybox, bzcat, bzip2, cal, cat, catv, chat, chattr, chcon, chgrp, chmod, chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm, cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devfsd, df, dhcprelay, diff, dirname, dmesg, dnsd, dos2unix, dpkg, dpkg_deb, du, dumpkmap, dumpleases, e2fsck, echo, ed, egrep, eject, env, envdir, envuidgid, ether_wake, expand, expr, fakeidentd, false, fbset, fbsplash, fdflush, fdformat, fdisk, fetchmail, fgrep, find, findfs, fold, free, freeramdisk, fsck, fsck_minix, ftpget, ftpput, fuser, getenforce, getopt, getsebool, getty, grep, gunzip, gzip, halt, hd, hdparm, head, hexdump, hostid, hostname, httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifup, inetd, init, inotifyd, insmod, install, ip, ipaddr, ipcalc, ipcrm, ipcs, iplink, iproute, iprule, iptunnel, kbd_mode, kill, killall, killall5, klogd, lash, last, length, less, linux32, linux64, linuxrc, ln, load_policy, loadfont, loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lzmacat, makedevs, man, matchpathcon, md5sum, mdev, mesg, microcom, mkdir, mke2fs, mkfifo, mkfs_minix, mknod, mkswap, mktemp, modprobe, more, mount, mountpoint, msh, mt, mv, nameif, nc, netstat, nice, nmeter, nohup, nslookup, od, openvt, parse, passwd, patch, pgrep, pidof, ping, ping6, pipe_progress, pivot_root, pkill, poweroff, printenv, printf, ps, pscan, pwd, raidautorun, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, renice, reset, resize, restorecon, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run_parts, runcon, runlevel, runsv, runsvdir, rx, script, sed, selinuxenabled, sendmail, seq, sestatus, setarch, setconsole, setenforce, setfiles, setfont, setkeycodes, setlogcons, setsebool, setsid, setuidgid, sh, sha1sum, showkey, slattach, sleep, softlimit, sort, split, start_stop_daemon, stat, strings, stty, su, sulogin, sum, sv, svlogd, swapoff, swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, taskset, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, top, touch, tr, traceroute, true, tty, ttysize, tune2fs, udhcpc, udhcpd, udpsvd, umount, uname, uncompress, unexpand, uniq, unix2dos, unlzma, unzip, uptime, usleep, uudecode, uuencode, vconfig, vi, vlock, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat, zcip

Applet highlight: Busybox init

- Busybox provides an implementation of an `init` program
- Simpler than the init implementation found on desktop/server systems: no runlevels are implemented
- A single configuration file: `/etc/inittab`
  - Each line has the form `<id>::<action>:<process>`
- Allows to run services at startup, and to make sure that certain services are always running on the system
- See `examples/inittab` in Busybox for details on the configuration

Applet highlight - BusyBox vi

- If you are using BusyBox, adding `vi` support only adds 20K. (built with shared libraries, using uClibc).
- You can select which exact features to compile in.
- Users hardly realize that they are using a lightweight `vi` version!
- Tip: you can learn `vi` on the desktop, by running the `vimtutor` command.

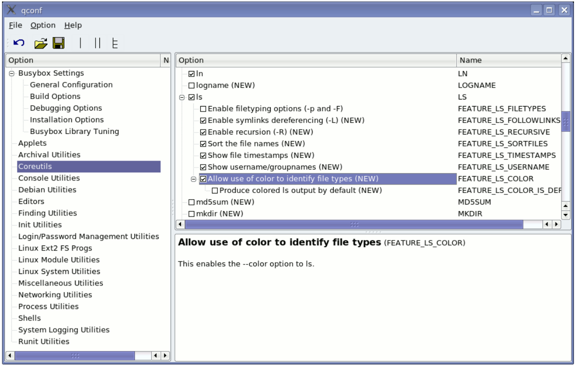
Configuring BusyBox

- Get the latest stable sources from `http://busybox.net`
- Configure BusyBox (creates a `.config` file):
  - `make defconfig`
    Good to begin with BusyBox.
    Configures BusyBox with all options for regular users.
  - `make allnoconfig`
    Unselects all options. Good to configure only what you need.
- `make xconfig` (graphical, needs the `libqt3-mt-dev` package)
  or `make menuconfig` (text)
  Same configuration interfaces as the ones used by the Linux kernel (though older versions are used).

BusyBox make xconfig

You can choose:

- the commands to compile,
- and even the command options and features that you need!



Compiling BusyBox

- Set the cross-compiler prefix in the configuration interface:
  `BusyBox Settings -> Build Options -> Cross Compiler prefix`
  Example: `arm-linux-`
- Set the installation directory in the configuration interface:
  `BusyBox Settings -> Installation Options -> BusyBox installation prefix`
- Add the cross-compiler path to the PATH environment variable:
  `export PATH=/usr/xtools/arm-unknown-linux-uclibcgnueabi/bin:$PATH`
- Compile BusyBox:
  `make`
- Install it (this creates a Unix directory structure symbolic links to the `busybox` executable):
  `make install`

Init

System V init

- Examines the /etc/inittab file for an :initdefault: entry, which tells init whether there is a default runlevel.
- The runlevels in System V describe certain states of a machine, characterized by the processes run. These are the runlevels 0 to 6 and S or s, which are aliased to the same runlevel. Of these eight, 3 are so-called "reserved" runlevels:
  - 0: Halt
  - 1: Single user mode
  - 6: Reboot

System V scripts

- Each service is started and stopped using scripts located in `/etc/init.d`. Those scripts are taking arguments (start, stop, restart).
- For each runlevel, there is a directory called `/etc/rc<level>.d`. That directory contains links to the scripts in `/etc/init.d`.
  - The script are launched by alphabetical order
  - If the link starts by an S, "start" is passed as an argument to the script
  - If the link starts by a K, "stop" is passed as an argument to the script
- example:

```
/etc/rc2.d/S99rc.local -> ../init.d/rc.local
```

Upstart

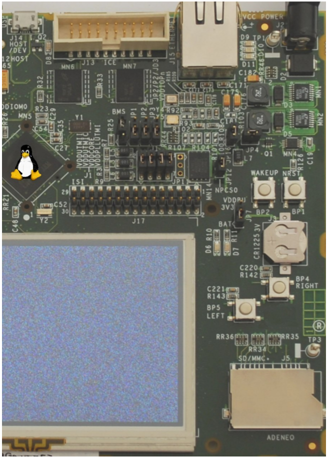
- The traditional init process is strictly synchronous, blocking future tasks until the current one has completed.
- Upstart operates asynchronously, as well as handling the starting of tasks and services during boot and stopping them during shutdown, it supervises them while the system is running.
- Upstart is able to run sysvinit scripts unmodified.
- Others services are configured in `/etc/init/*.conf`
- use the `service` command:
  `service <servicename> start|stop`

- Linux only
- Tries to launch services in parallel and tracks dependencies
- manages socket-activated and bus-activated services
- uses cgroups to monitor services
- `udev`'s sources are now merged in `systemd`

- services (called units) are defined in `/usr/lib/systemd/system/`
- they are linked in runlevels, for example: `/usr/lib/systemd/system/graphical.target` or `/etc/systemd/system/<your target>`
- controlled using `systemclt start|stop <service>`

Hotplugging with udev

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



/dev issues and limitations

- On Red Hat 9, 18000 entries in `/dev`!
  All entries for all possible devices had to be created at system installation.
- Needed an authority to assign major numbers
  `http://lanana.org/`: Linux Assigned Names and Numbers Authority
- Not enough numbers in 2.4, limits extended in 2.6.
- Userspace neither knew what devices were present in the system, nor which real device corresponded to each `/dev` entry.

The udev solution

Takes advantage of **sysfs** introduced by Linux 2.6.

- Created by Greg Kroah Hartman, a huge contributor.
  Other key contributors: Kay Sievers, Dan Stekloff.
- *Entirely* in user space.
- Automatically creates and removes device entries in `/dev/` according to inserted and removed devices.
- Major and minor device transmitted by the kernel.
- Requires no change to driver code.
- Fast: written in C
  Relatively small size: `udevd` version 167: 127 KB

Starting udev (1)

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

```
$ ls -l /lib/udev/devices/
total 12
brw------- 1 root root   7, 0 2011-06-04 10:25 loop0
drwxr-xr-x 2 root root   4096 2011-06-04 10:25 net
crw------- 1 root root 108, 0 2011-06-04 10:25 ppp
drwxr-xr-x 2 root root   4096 2011-04-07 14:43 pts
drwxr-xr-x 2 root root   4096 2011-04-07 14:43 shm
```

Starting udev (2)

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

Example inserting a USB mouse

```
recv(4,
     "add@/class/input/input9/mouse2\0
     ACTION=add\0
     DEVPATH=/class/input/input9/mouse2\0
     SUBSYSTEM=input\0
     SEQNUM=1064\0
     PHYSDEVPATH=/devices/pci0000:00/0000:00:1d.1/usb2/2-2/2-2:1.0\0
     PHYSDEVBUS=usb\0
     PHYSDEVDRIVER=usbhid\0
     MAJOR=13\0
     MINOR=34\0",
     2048,
     0)
= 221
```

udev rules

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

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

Rule files are processed in lexical order.

udev naming capabilities

Device names can be defined

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

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

udev naming rule examples

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

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

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

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

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

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

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

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

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

# Other devices, by name
KERNEL=="null", MODE="0666"
KERNEL=="zero", MODE="0666"
KERNEL=="full", MODE="0666"
```

Identifying device driver modules

Kernel compilation time

Each driver announces which device and vendor IDs it supports. Information stored in the source code.

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

System everyday life

The driver core (USB, PCI, etc.) reads the device ID, vendor ID, and other device attributes

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

A udev event process runs modprobe $MODALIAS

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

Module aliases

- `MODALIAS` environment variable example (USB mouse):
  `MODALIAS=usb:`
  `v046DpC03Ed2000dc00dsc00dp00ic03isc01ip02`
- Matching line in
  `/lib/modules/<version>/modules.alias`:
  `alias usb:v*p*d*dc*dsc*dp*ic03isc01ip02* usbmouse`

Even module loading is done with `udev`!

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

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

SUBSYSTEM!="ide", GOTO="ide_end"
IMPORT{program}="ide_media --export $devpath"
ENV{IDE_MEDIA}=="cdrom",RUN+="/sbin/modprobe -Qba ide-cd"
ENV{IDE_MEDIA}=="disk",RUN+="/sbin/modprobe -Qba ide-disk"
ENV{IDE_MEDIA}=="floppy", RUN+="/sbin/modprobe -Qba ide-floppy"
ENV{IDE_MEDIA}=="tape", RUN+="/sbin/modprobe -Qba ide-tape"
LABEL="ide_end"
SUBSYSTEM=="input", PROGRAM="/sbin/grepmap --udev", \
   RUN+="/sbin/modprobe -Qba $result"
# Load drivers that match kernel-supplied alias
ENV{MODALIAS}=="?*", RUN+="/sbin/modprobe -Q $env{MODALIAS}"
```

Cold-plugging

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

Debugging events - udevmonitor (1)

`udevadm monitor` visualizes the driver core events and the `udev` event processes.

Example event sequence connecting a USB mouse:

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

It gives time information measured in microseconds. You can measure time elapsed between the uevent (`UEVENT` line), and the completion of the corresponding `udev` process (matching `UDEV` line).

Debugging events - udevmonitor (2)

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

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

Misc udev utilities

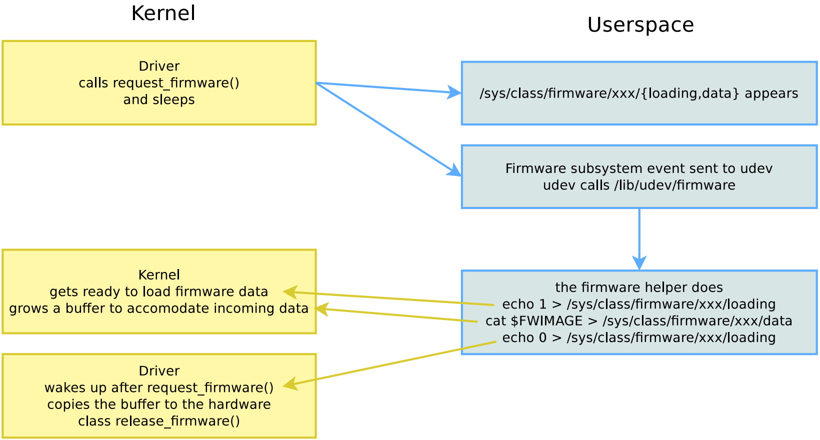
- `udevinfo`
  Lets users query the `udev` database.
- udevtest <sysfs_device_path>
  Simulates a `udev` run to test the configured rules.

Firmware hotplugging

Also implemented with udev!

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

Firmware hotplugging implementation



See `Documentation/firmware_class/` in the kernel sources for a nice overview

udev files

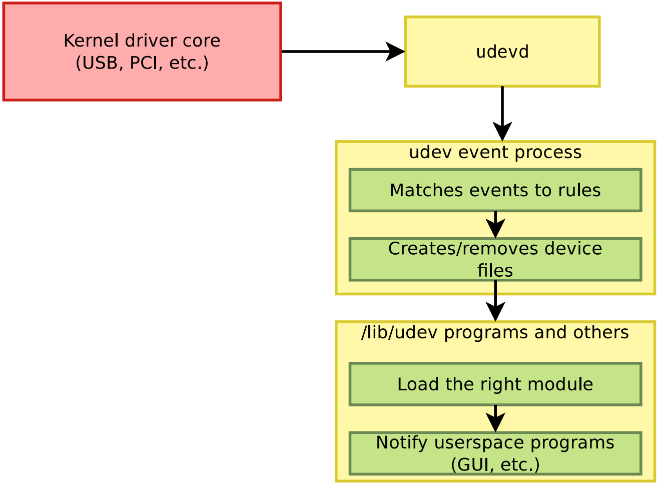
- `/etc/udev/udev.conf`
  `udev` configuration file.
  Mainly used to configure syslog reporting priorities.
  Example setting: `udev_log="err"`
- `/lib/udev/rules.d/`
  Standard `udev` event matching rules, installed by the distribution.
- `/etc/udev/rules.d/*.rules`
  Local (custom) `udev` event matching rules. Best to modify these.
- `/lib/udev/devices/*`
  static `/dev` content (such as `/dev/console`, `/dev/null`...).
- `/lib/udev/*`
  helper programs called from `udev` rules.
- `/dev/*`
  Created device files.

Kernel configuration for udev

- Created for 2.6.19
- Caution: no documentation found, and not tested yet on a minimalistic system. Some settings may still be missing.
- Subsystems and device drivers (USB, PCI, PCMCIA...) should be added too!

```
# General setup
CONFIG_HOTPLUG=y
# Networking, networking options
CONFIG_NET=y
# Unix domain sockets
CONFIG_UNIX=y
CONFIG_NETFILTER_NETLINK=y
CONFIG_NETFILTER_NETLINK_QUEUE=y
# Pseudo filesystems
CONFIG_PROC_FS=y
CONFIG_SYSFS=y
# Needed to manage /dev
CONFIG_TMPFS=y
CONFIG_RAMFS=y
```

udev summary - typical operation



udev resources

- Home page
  `http://kernel.org/pub/linux/utils/kernel/hotplug/udev.html`
- Sources
  `http://kernel.org/pub/linux/utils/kernel/hotplug/`
- The `udev` manual page:
  `man udev`

mdev, the udev for embedded systems

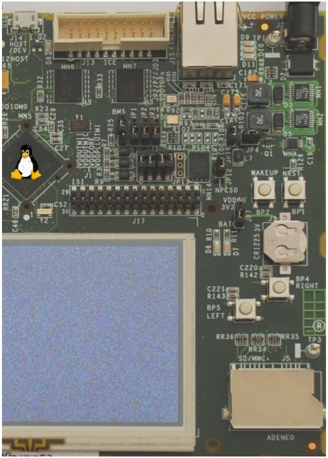
- *udev* might be too heavy-weight for some embedded systems, the `udevd` daemon staying in the background waiting for events.
- BusyBox provides a simpler alternative called `mdev`, available by enabling the `MDEV` configuration option.
- `mdev`'s usage is documented in `doc/mdev.txt` in the BusyBox source code.
- `mdev`mdev is also able to load firmware to the kernel like `udev`

mdev usage

- To use mdev, the proc and sysfs filesystems must be mounted
- mdev must be enabled as the hotplug event manager
  `echo /sbin/mdev > /proc/sys/kernel/hotplug`
- Need to mount /dev as a tmpfs:
  `mount -t tmpfs mdev /dev`
- Tell mdev to create the /dev entries corresponding to the devices detected during boot when mdev was not running:
  `mdev -s`
- The behavior is specified by the /etc/mdev.conf configuration file, with the following format
  `<device regex> <uid>:<gid> <octal permissions> [=path] [@|$|*<command>]`
- Example
  `hd[a-z][0-9]* 0:3 660`

Cross-compiling toolchains

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



C Libraries

glibc

- License: LGPL
- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Of course, actively maintained
- Quite big for small embedded systems: approx 2.5 MB on ARM (version 2.9 - `libc`: 1.5 MB, `libm`: 750 KB)
- `http://www.gnu.org/software/libc/`

uClibc

- License: LGPL
- Lightweight C library for small embedded systems
  - High configurability: many features can be enabled or disabled through a menuconfig interface
  - Works only with Linux/uClinux, works on most embedded architectures
  - No stable ABI, different ABI depending on the library configuration
  - Focus on size rather than performance
  - Small compile time
- `http://www.uclibc.org/`

uClibc (2)

- Most of the applications compile with uClibc. This applies to all applications used in embedded systems.
- Size (arm): 4 times smaller than glibc!
  - uClibc 0.9.30.1: approx. 600 KB (libuClibc: 460 KB, libm: 96KB)
  - glibc 2.9: approx 2.5 MB
- Some features not available or limited: priority-inheritance mutexes, NPTL support is very new, fixed Name Service Switch functionality, etc.
- Used on a large number of production embedded products, including consumer electronic devices
- Actively maintained, large developer and user base
- Supported and used by MontaVista, TimeSys and Wind River.

Honey, I shrunk the programs!

- Executable size comparison on ARM, tested with *glibc* 2.9 and *uClibc* 0.9.30.1
- Plain "hello world" program (stripped)
  - With shared libraries: 5.6 KB with *glibc*, 5.4 KB with *uClibc*
  - With static libraries: 472 KB with *glibc*, 18 KB with *uClibc*
- Busybox (stripped)
  - With shared libraries: 245 KB with *glibc*, 231 KB with *uClibc*
  - With static libraries: 843 KB with *glibc*, 311 KB with *uClibc*

- *Embedded glibc*, under the LGPL
- Variant of the GNU C Library (GLIBC) designed to work well on embedded systems
- Strives to be source and binary compatible with GLIBC
- eglibc's goals include reduced footprint, configurable components, better support for cross-compilation and cross-testing.
- Can be built without support for NIS, locales, IPv6, and many other features.
- Supported by a consortium, with Freescale, MIPS, MontaVista and Wind River as members.
- The Debian distribution has switched to eglibc too, `http://blog.aurel32.net/?p=47`
- `http://www.eglibc.org`

Other smaller C libraries

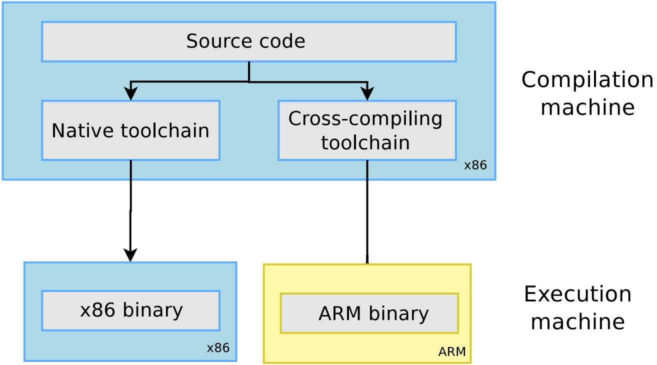
- Several other smaller C libraries have been developed, but none of them have the goal of allowing the compilation of large existing applications
- They need specially written programs and applications
- Choices:
  - Dietlibc, `http://www.fefe.de/dietlibc/`. Approximately 70 KB.
  - Newlib, `http://sourceware.org/newlib/`
  - Klibc, `http://www.kernel.org/pub/linux/libs/klibc/`, designed for use in an *initramfs* or *initrd* at boot time.

Definition and Components

Definition (1)

- The usual development tools available on a GNU/Linux workstation is a **native toolchain**
- This toolchain runs on your workstation and generates code for your workstation, usually x86
- For embedded system development, it is usually impossible or not interesting to use a native toolchain
  - The target is too restricted in terms of storage and/or memory
  - The target is very slow compared to your workstation
  - You may not want to install all development tools on your target.
- Therefore, **cross-compiling toolchains** are generally used. They run on your workstation but generate code for your target.

Definition (2)



Machines in build procedures

- Three machines must be distinguished when discussing toolchain creation
  - The **build** machine, where the toolchain is built.
  - The **host** machine, where the toolchain will be executed.
  - The **target** machine, where the binaries created by the toolchain are executed.
- Four common build types are possible for toolchains

Different toolchain build procedures

Build Host Target

Native build

used to build the normal gcc
of a workstation

Build Host Target

Cross build

used to build a toolchain that runs
on your workstation but generates
binaries for the target

The most common case in embedded development

Build Host Target

Cross-native build

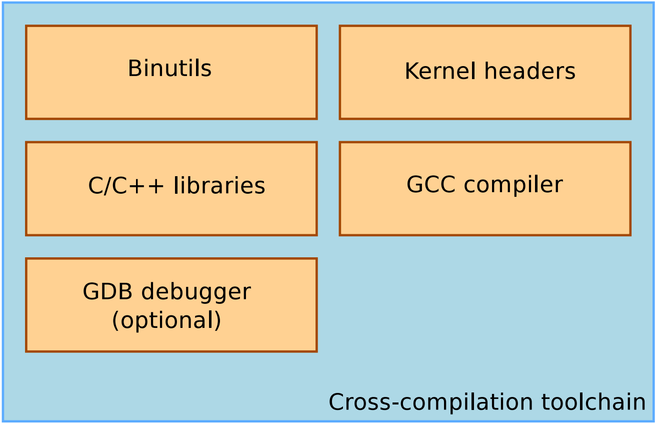
used to build a toolchain that runs on your
target and generates binaries for the target

Build Host Target

Canadian build

used to build on architecture A a
toolchain that runs on architecture B
and generates binaries for architecture C

Components

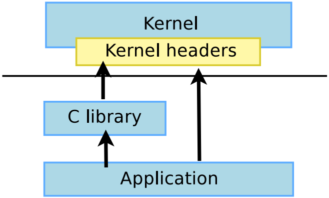

Binutils
Kernel headers
C/C++ libraries
GCC compiler
GDB debugger (optional)
Cross-compilation toolchain

Binutils

- **Binutils** is a set of tools to generate and manipulate binaries for a given CPU architecture
  - `as`, the assembler, that generates binary code from assembler source code
  - `ld`, the linker
  - `ar`, `ranlib`, to generate `.a` archives, used for libraries
  - `objdump`, `readelf`, `size`, `nm`, `strings`, to inspect binaries. Very useful analysis tools!
  - `strip`, to strip useless parts of binaries in order to reduce their size
- `http://www.gnu.org/software/binutils/`
- GPL license

Kernel headers (1)

- The C library and compiled programs needs to interact with the kernel
  - Available system calls and their numbers
  - Constant definitions
  - Data structures, etc.
- Therefore, compiling the C library requires kernel headers, and many applications also require them.
- Available in `<linux/...>` and `<asm/...>` and a few other directories corresponding to the ones visible in `include/` in the kernel sources



Kernel headers (2)

- System call numbers, in <asm/unistd.h>

```
#define __NR_exit           1
#define __NR_fork           2
#define __NR_read           3
```

- Constant definitions, here in <asm-generic/fcntl.h>, included from <asm/fcntl.h>, included from <linux/fcntl.h>

```
#define O_RDWR 00000002
```

- Data structures, here in <asm/stat.h>

```
struct stat {
    unsigned long st_dev;
    unsigned long st_ino;
    [...]
};
```

Kernel headers (3)

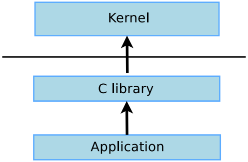
- The kernel-to-userspace ABI is **backward compatible**
  - Binaries generated with a toolchain using kernel headers older than the running kernel will work without problem, but won't be able to use the new system calls, data structures, etc.
  - Binaries generated with a toolchain using kernel headers newer than the running kernel might work on if they don't use the recent features, otherwise they will break
  - Using the latest kernel headers is not necessary, unless access to the new kernel features is needed
- The kernel headers are extracted from the kernel sources using the `headers_install` kernel Makefile target.

GCC compiler

- GNU C Compiler, the famous free software compiler
- Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, and generate code for a large number of CPU architectures, including ARM, AVR, Blackfin, CRIS, FRV, M32, MIPS, MN10300, PowerPC, SH, v850, i386, x86_64, IA64, Xtensa, etc.
- `http://gcc.gnu.org/`
- Available under the GPL license, libraries under the LGPL.

C library

- The C library is an essential component of a Linux system
  - Interface between the applications and the kernel
  - Provides the well-known standard C API to ease application development
- Several C libraries are available: *glibc*, *uClibc*, *eglibc*, *dietlibc*, *newlib*, etc.
- The choice of the C library must be made at the time of the cross-compiling toolchain generation, as the GCC compiler is compiled against a specific C library.



Obtaining a Toolchain

Building a toolchain manually

Building a cross-compiling toolchain by yourself is a difficult and painful task! Can take days or weeks!

- Lots of details to learn: many components to build, complicated configuration
- Lots of decisions to make (such as C library version, ABI, floating point mechanisms, component versions)
- Need kernel headers and C library sources
- Need to be familiar with current `gcc` issues and patches on your platform
- Useful to be familiar with building and configuring tools
- See the *Crosstool-NG* `docs/` directory for details on how toolchains are built.

Get a pre-compiled toolchain

- Solution that many people choose
  - Advantage: it is the simplest and most convenient solution
  - Drawback: you can't fine tune the toolchain to your needs
- Determine what toolchain you need: CPU, endianism, C library, component versions, ABI, soft float or hard float, etc.
- Check whether the available toolchains match your requirements.
- Possible choices
  - Sourcery CodeBench toolchains
  - Linaro toolchains
  - More references at `http://elinux.org/Toolchains`

Sourcery CodeBench

- *CodeSourcery* was a a company with an extended expertise on free software toolchains: gcc, gdb, binutils and glibc. It has been bought by *Mentor Graphics*, which continues to provide similar services and products
- They sell toolchains with support, but they also provide a "*Lite*" version, which is free and usable for commercial products
- They have toolchains available for
  - ARM
  - MIPS
  - PowerPC
  - SuperH
  - x86
- Be sure to use the GNU/Linux versions. The EABI versions are for bare-metal development (no operating system)

Linaro toolchains

- Linaro contributes to improving mainline gcc on ARM, in particular by hiring CodeSourcery developers.
- For people who can't wait for the next releases of gcc, Linaro releases modified sources of stable releases of gcc, with these optimizations for ARM (mainly for recent Cortex A CPUs).
- As any gcc release, these sources can be used by build tools to build their own binary toolchains (Buildroot, OpenEmbedded...) This allows to support glibc, uClibc and eglibc.
- `https://wiki.linaro.org/WorkingGroups/ToolChain`
- Binary packages are available for Ubuntu users, `https://launchpad.net/~linaro-maintainers/+archive/toolchain`

Installing and using a pre-compiled toolchain

- Follow the installation procedure proposed by the vendor
- Usually, it is simply a matter of extracting a tarball wherever you want.
- Then, add the path to toolchain binaries in your `PATH`:
  `export PATH=/path/to/toolchain/bin/:$PATH`
- Finally, compile your applications
  `PREFIX-gcc -o foobar foobar.c`
- `PREFIX` depends on the toolchain configuration, and allows to distinguish cross-compilation tools from native compilation utilities

Toolchain building utilities

Another solution is to use utilities that **automate the process of building the toolchain**

- Same advantage as the pre-compiled toolchains: you don't need to mess up with all the details of the build process
- But also offers more flexibility in terms of toolchain configuration, component version selection, etc.
- They also usually contain several patches that fix known issues with the different components on some architectures
- Multiple tools with identical principle: shell scripts or Makefile that automatically fetch, extract, configure, compile and install the different components

- **Crosstool-ng**
  - Rewrite of the older Crosstool, with a menuconfig-like configuration system
  - Feature-full: supports uClibc, glibc, eglibc, hard and soft float, many architectures
  - Actively maintained
  - `http://crosstool-ng.org/`

Toolchain building utilities (3)

Many root filesystem building systems also allow the construction of a cross-compiling toolchain

- **Buildroot**
  - Makefile-based, has a Crosstool-NG back-end, maintained by the community
  - `http://www.buildroot.net`
- **PTXdist**
  - Makefile-based, uClibc or glibc, maintained mainly by *Pengutronix*
  - `http://www.pengutronix.de/software/ptxdist/index_en.html`
- **OpenEmbedded**
  - The feature-full, but more complicated building system
  - `http://www.openembedded.org/`

Crosstool-NG: installation and usage

- Installation of Crosstool-NG can be done system-wide, or just locally in the source directory. For local installation:

```
./configure --enable-local
make
make install
```

- Some sample configurations for various architectures are available in samples, they can be listed using

```
./ct-ng list-samples
```

- To load a sample configuration

```
./ct-ng <sample-name>
```

- To adjust the configuration

```
./ct-ng menuconfig
```

- To build the toolchain

```
./ct-ng build
```

Toolchain contents

- The cross compilation tool binaries, in `bin/`
  - This directory can be added to your `PATH` to ease usage of the toolchain
- One or several *sysroot*, each containing
  - The C library and related libraries, compiled for the target
  - The C library headers and kernel headers
- There is one *sysroot* for each variant: toolchains can be *multilib* if they have several copies of the C library for different configurations (for example: ARMv4T, ARMv5T, etc.)
  - CodeSourcery ARM toolchain are multilib, the sysroots are in `arm-none-linux-gnueabi/libc/`, `arm-none-linux-gnueabi/libc/armv4t/`, `arm-none-linux-gnueabi/libc/thumb2`
  - Crosstool-NG toolchains are never multilib, the sysroot is in `arm-unknown-linux-uclibcgnueabi/sysroot`

Toolchain Options

ABI

- When building a toolchain, the ABI used to generate binaries needs to be defined
- ABI, for *Application Binary Interface*, defines the calling conventions (how function arguments are passed, how the return value is passed, how system calls are made) and the organization of structures (alignment, etc.)
- All binaries in a system must be compiled with the same ABI, and the kernel must understand this ABI.
- On ARM, two main ABIs: *OABI* and *EABI*
  - Nowadays everybody uses *EABI*
- On MIPS, several ABIs: *o32, o64, n32, n64*
- `http://en.wikipedia.org/wiki/Application_Binary_Interface`

Floating point support

- Some processors have a floating point unit, some others do not.
  - For example, many ARMv4 and ARMv5 CPUs do not have a floating point unit. Since ARMv7, a VFP unit is mandatory.
- For processors having a floating point unit, the toolchain should generate *hard float* code, in order to use the floating point instructions directly
- For processors without a floating point unit, two solutions
  - Generate *hard float code* and rely on the kernel to emulate the floating point instructions. This is very slow.
  - Generate *soft float code*, so that instead of generating floating point instructions, calls to a userspace library are generated
- Decision taken at toolchain configuration time
- Also possible to configure which floating point unit should be used

CPU optimization flags

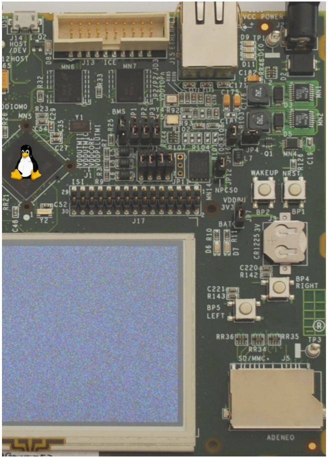
- A set of cross-compiling tools is specific to a CPU architecture (ARM, x86, MIPS, PowerPC)
- However, with the `-march=`, `-mcpu=`, `-mtune=` options, one can select more precisely the target CPU type
  - For example, `-march=armv7 -mcpu=cortex-a8`
- At the toolchain compilation time, values can be chosen. They are used:
  - As the default values for the cross-compiling tools, when no other `-march`, `-mcpu`, `-mtune` options are passed
  - To compile the C library
- Even if the C library has been compiled for armv5t, it doesn't prevent from compiling other programs for armv7

Introduction to Android

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier

Free Electrons, Adeneo Embedded

Corrections, suggestions, contributions and translations are welcome!



History

Early Years

- Began as a start-up in Palo Alto, CA, USA in 2003
- Focused from the start on software for mobile devices
- Very secretive at the time, even though founders achieved a lot in the targeted area before founding it
- Finally bought by Google in 2005

Opening Up

- Google announced the Open Handset Alliance in 2007, a consortium of major actors in the mobile area built around Android
  - Hardware vendors: Intel, Texas Instruments, Qualcomm, Nvidia, etc.
  - Software companies: Google, eBay, etc.
  - Hardware manufacturers: Motorola, HTC, Sony Ericsson, Samsung, etc.
  - Mobile operators: T-Mobile, Telefonica, Vodafone, etc.

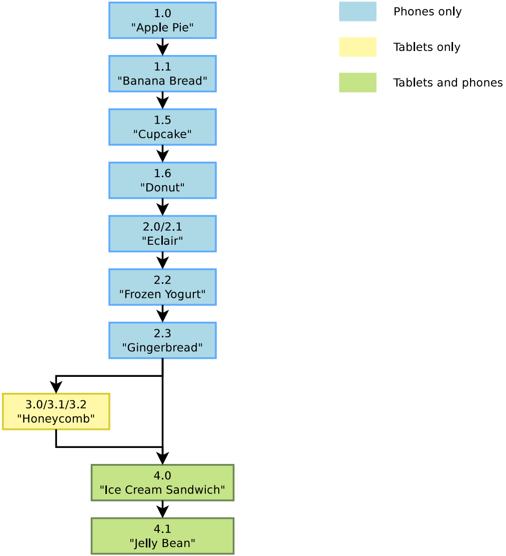
Android Open Source Project (AOSP)

- At every new version, Google releases its source code through this project so that community and vendors can work with it.
  - One major exception: Honeycomb has not been released because Google stated that its source code was not clean enough to release it.
- One can fetch the source code and contribute to it, even though the development process is very locked by Google
- Only a few devices are supported through AOSP though, only the two most recent Android development phones, the Panda board and the Motorola Xoom.

Android Releases

- Each new version is given a dessert name
- Released in alphabetical order
- Last releases:
  - Android 2.3 Gingerbread
  - Android 3.X Honeycomb
  - Android 4.0 Ice Cream Sandwich
  - Android 4.1 Jelly Bean

Android Versions



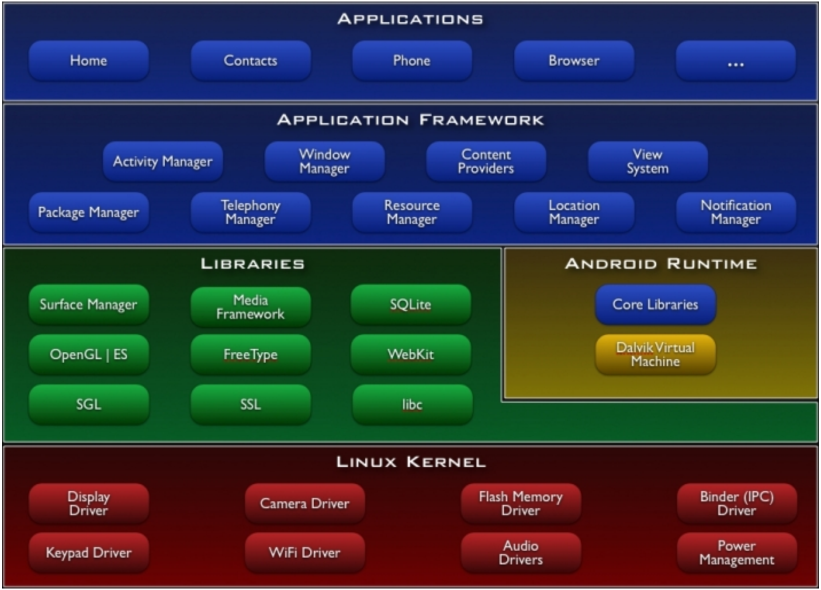
Features

Features

- All you can expect from a modern mobile OS:
  - Application ecosystem, allowing to easily add and remove applications and publish new features across the entire system
  - Support for all the web technologies, with a browser built on top of the well-established WebKit rendering engine
  - Support for hardware accelerated graphics through OpenGL ES
  - Support for all the common wireless mechanisms: GSM, CDMA, UMTS, LTE, Bluetooth, WiFi.

Architecture

Architecture



The Linux Kernel

- Used as the foundation of the Android system
- Numerous additions from the stock Linux, including new IPC (Inter-Process Communication) mechanisms, alternative power management mechanism, new drivers and various additions across the kernel
- These changes are beginning to go into the `staging/` area of the kernel, as of 3.3, after being a complete fork for a long time

Android Libraries

- Gather a lot of Android-specific libraries to interact at a low-level with the system, but third-parties libraries as well
- Bionic is the C library, SurfaceManager is used for drawing surfaces on the screen, etc.
- But also WebKit, SQLite, OpenSSL coming from the free software world

Android Runtime

Handles the execution of Android applications

- Almost entirely written from scratch by Google
- Contains Dalvik, the virtual machine that executes every application that you run on Android, and the core library for the Java runtime, coming from Apache Harmony project
- Also contains system daemons, init executable, basic binaries, etc.

Android Framework

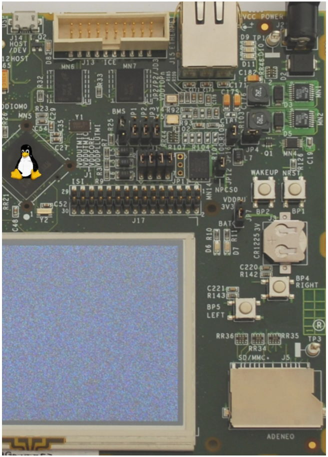
- Provides an API for developers to create applications
- Exposes all the needed subsystems by providing an abstraction
- Allows to easily use databases, create services, expose data to other applications, receive system events, etc.

Android Applications

- AOSP also comes with a set of applications such as the phone application, a browser, a contact management application, an email client, etc.
- However, the Google apps and the Android Market app aren't free software, so they are not available in AOSP. To obtain them, you must contact Google and pass a compatibility test.

Changes introduced in the Android Kernel

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



Anonymous Shared Memory (ashmem)

Shared memory mechanism in Linux

- Shared memory is one of the standard IPC mechanisms present in most OSes
- Under Linux, they are usually provided by the POSIX SHM mechanism, which is part of the System V IPCs
- `ndk/docs/system/libc/SYSV-IPC.html` illustrates all the love Android developers have for these
- The bottom line is that they are flawed by design in Linux, and lead to code leaking resources, be it maliciously or not

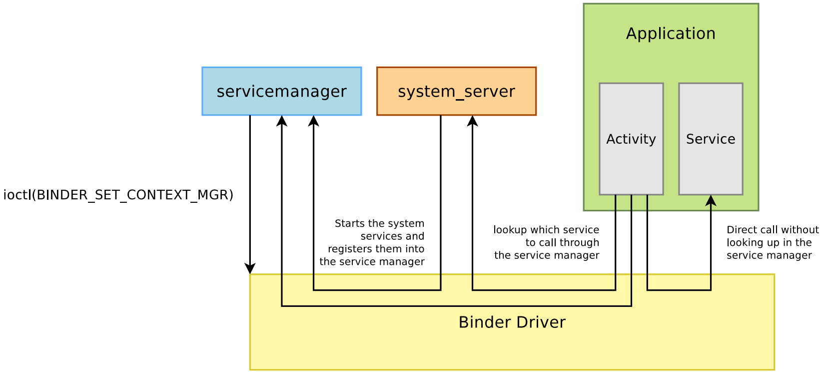
Ashmem

- Ashmem is the response to these flaws
- Notable differences are:
  - Reference counting so that the kernel can reclaim resources which are no longer in use
  - There is also a mechanism in place to allow the kernel to shrink shared memory regions when the system is under memory pressure.
- The standard use of Ashmem in Android is that a process opens a shared memory region and share the obtained file descriptor through Binder.

Binder

Binder

- RPC/IPC mechanism
- Takes its roots from BeOS and the OpenBinder project, which some of the current Android engineers worked on
- Adds remote object invocation capabilities to the Linux Kernel
- One of the very basic functionalities of Android. Without it, Android cannot work.
- Every call to the system servers go through Binder, just like every communication between applications, and even communication between the components of a single application.


Application
servicemanager
system_server
Activity
Service
ioctl(BINDER_SET_CONTEXT_MGR)
Starts the system services and registers them into the service manager
lookup which service to call through the service manager
Direct call without looking up in the service manager
Binder Driver

klogger

Logging

- Logs are very important to debug a system, either live or after a fault occurred
- In a regular Linux distribution, two components are involved in the system's logging:
  - Linux' internal mechanism, accessible with the `dmesg` command and holding the output of all the calls to `printk()` from various parts of the kernel.
  - A syslog daemon, which handles the userspace logs and usually stores them in the `/var/log` directory
- From Android developers' point of view, this approach has two flaws:
  - As the calls to `syslog()` go through as socket, they generate expensive task switches
  - Every call writes to a file, which probably writes to a slow storage device or to a storage device where writes are expensive

Logger

- Android addresses these issues with *logger*, which is a kernel driver, that uses 4 circular buffers in the kernel memory area.
- The buffers are exposed in the `/dev/log` directory and you can access them through the *liblog* library, which is in turn, used by the Android system and applications to write to logger, and by the *logcat* command to access them.
- This allows to have an extensive level of logging across the entire AOSP

Low Memory Killer

Low Memory Killer

- When the system goes out of memory, Linux throws the OOM Killer to cleanup memory greedy processes
- However, this behaviour is not predictable at all, and can kill very important components of a phone (Telephony stack, Graphic subsystem, etc) instead of low priority processes (Angry Birds)
- The main idea is to have another process killer, that kicks in before the OOM Killer and takes into account the time since the application was last used and the priority of the component for the system
- It uses various thresholds, so that it first notifies applications so that they can save their state, then begins to kill non-critical background processes, and then the foreground applications
- As it is run to free memory before the OOM Killer, the latter will never be run, as the system will never run out of memory

Various Drivers and Fixes

Various additions

- Android also has a lot of minor features added to the Linux kernel:
  - RAM Console, a RAM-based console that survives a reboot to hold kernel logs
  - *pmem*, a physically contiguous memory allocator, written specifically for the HTC G1, to allocate heaps used for 2D hardware acceleration
  - ADB
  - YAFFS2
  - Timed GPIOs

Network Security

Paranoid Network

- In the standard Linux kernel, every application can open sockets and communicate over the Network
- However, Google was willing to apply a more strict policy with regard to network access
- Access to the network is a permission, with a per application granularity
- Filtered with the GID
- You need it to access IP, Bluetooth, raw sockets or RFCOMM

Wakelocks

Power management basics

- Every CPU has a few states of power consumption, from being almost completely off, to working at full capacity.
- These different states are used by the Linux kernel to save power when the system is run
- For example, when the lid is closed on a laptop, it goes into "suspend", which is the most power conservative mode of a device, where almost nothing but the RAM is kept awake
- While this is a good strategy for a laptop, it is not necessarily good for mobile devices
- For example, you don't want your music to be turned off when the screen is off

Wakelocks

- Android's answer to these power management constraints is wakelocks
- One of the most famous Android changes, because of the flame wars it spawned
- The main idea is instead of letting the user decide when the devices need to go to sleep, the kernel is set to suspend as soon and as often as possible.
- In the same time, Android allows applications and kernel drivers to voluntarily prevent the system from going to suspend, keeping it awake (thus the name wakelock)
- This implies to write the applications and drivers to use the wakelock API.
  - Applications do so through the abstraction provided by the API
  - Drivers must do it themselves, which prevents to directly submit them to the vanilla kernel

Wakelocks API

- Kernel Space API

```
#include <linux/wakelock.h>
void wake_lock_init(struct wakelock *lock,
                    int type,
                    const char *name);
void wake_lock(struct wake_lock *lock);
void wake_unlock(struct wake_lock *lock);
void wake_lock_timeout(struct wake_lock *lock, long timeout);
void wake_lock_destroy(struct wake_lock *lock);
```

- User-Space API

```
$ echo foobar > /sys/power/wake_lock
$ echo foobar > /sys/power/wake_unlock
```

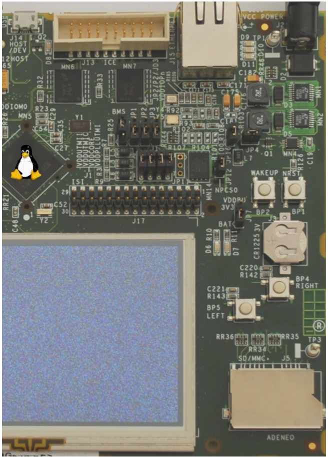
Alarm Timers

The alarm driver

- Once again, the timer mechanisms available in Linux were not sufficient for the power management policy that Android was trying to set up
- High Resolution Timers can wake up a process, but don't fire when the system is suspended, while the Real Time Clock can wake up the system if it is suspended, but cannot wake up a particular process.
- Developed the alarm timers on top of the Real Time Clock and High Resolution Timers already available in the kernel
- These timers will be fired even if the system is suspended, waking up the device to do so
- Obviously, to let the application do its job, when the application is woken up, a wakelock is grabbed

Android Native Layer

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



Bionic

Bionic 1/2

- Google developed another C library for Android: `Bionic`. They didn't start from scratch however, they based their work on the BSD standard C library.
- The most remarkable thing about Bionic is that it doesn't have full support for the POSIX API, so it might be a hurdle when porting an already developed program to Android.
- Among other things, are lacking:
  - Full pthreads API
  - No locales and wide chars support
  - No `openpty()`, `syslog()`, `crypt()`, functions
  - Removed dependency on the `/etc/resolv.conf` and `/etc/passwd` files and using Android's own mechanisms instead
  - Some functions are still unimplemented (see `getprotobyname()`

Bionic 2/2

- However, Bionic has been created this way for a number of reasons
  - Keep the libc implementation as simple as possible, so that it can be fast and lightweight (Bionic is a bit smaller than uClibc)
  - Keep the (L)GPL code out of the userspace. Bionic is under the BSD license
- And it implements some Android-specifics functions as well:
  - Access to system properties
  - Logging events in the system logs
- In the `prebuilt/` directory, Google provides a prebuilt toolchain that uses Bionic
- See `http://androidxref.com/4.0.4/xref/ndk/docs/system/libc/OVERVIEW.html` for details about Bionic.

Toolbox

Why Toolbox?

- A Linux system needs a basic set of programs to work
  - An init program
  - A shell
  - Various basic utilities for file manipulation and system configuration
- In normal Linux systems, those programs are provided by different projects
  - `coreutils`, `bash`, `grep`, `sed`, `tar`, `wget`, `modutils`, etc. are all different projects
  - Many different components to integrate
  - Components not designed with embedded systems constraints in mind: they are not very configurable and have a wide range of features
- Busybox is an alternative solution, extremely common on embedded systems

General purpose toolbox: BusyBox

- Rewrite of many useful Unix command line utilities
  - Integrated into a single project, which makes it easy to work with
  - Designed with embedded systems in mind: highly configurable, no unnecessary features
- All the utilities are compiled into a single executable, `/bin/busybox`
  - Symbolic links to `/bin/busybox` are created for each application integrated into Busybox
- For a fairly featureful configuration, less than 500 KB (statically compiled with uClibc) or less than 1 MB (statically compiled with glibc).
- `http://www.busybox.net/`

BusyBox commands!

Commands available in BusyBox 1.13

[, [[, addgroup, adduser, adjtimex, ar, arp, arping, ash, awk, basename, bbconfig, bbsh, brctl, bunzip2, busybox, bzcat, bzip2, cal, cat, catv, chat, chattr, chcon, chgrp, chmod, chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm, cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devfsd, df, dhcprelay, diff, dirname, dmesg, dnsd, dos2unix, dpkg, dpkg_deb, du, dumpkmap, dumpleases, e2fsck, echo, ed, egrep, eject, env, envdir, envuidgid, ether_wake, expand, expr, fakeidentd, false, fbset, fbsplash, fdflush, fdformat, fdisk, fetchmail, fgrep, find, findfs, fold, free, freeramdisk, fsck, fsck_minix, ftpget, ftpput, fuser, getenforce, getopt, getsebool, getty, grep, gunzip, gzip, halt, hd, hdparm, head, hexdump, hostid, hostname, httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifup, inetd, init, inotifyd, insmod, install, ip, ipaddr, ipcalc, ipcrm, ipcs, iplink, iproute, iprule, iptunnel, kbd_mode, kill, killall, killall5, klogd, lash, last, length, less, linux32, linux64, linuxrc, ln, load_policy, loadfont, loadkmap, logger, login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lzmacat, makedevs, man, matchpathcon, md5sum, mdev, mesg, microcom, mkdir, mke2fs, mkfifo, mkfs_minix, mknod, mkswap, mktemp, modprobe, more, mount, mountpoint, msh, mt, mv, nameif, nc, netstat, nice, nmeter, nohup, nslookup, od, openvt, parse, passwd, patch, pgrep, pidof, ping, ping6, pipe_progress, pivot_root, pkill, poweroff, printenv, printf, ps, pscan, pwd, raidautorun, rdate, rdev, readahead, readlink, readprofile, realpath, reboot, renice, reset, resize, restorecon, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake, run_parts, runcon, runlevel, runsv, runsvdir, rx, script, sed, selinuxenabled, sendmail, seq, sestatus, setarch, setconsole, setenforce, setfiles, setfont, setkeycodes, setlogcons, setsebool, setsid, setuidgid, sh, sha1sum, showkey, slattach, sleep, softlimit, sort, split, start_stop_daemon, stat, strings, stty, su, sulogin, sum, sv, svlogd, swapoff, swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar, taskset, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, top, touch, tr, traceroute, true, tty, ttysize, tune2fs, udhcpc, udhcpd, udpsvd, umount, uname, uncompress, unexpand, uniq, unix2dos, unlzma, unzip, uptime, usleep, uudecode, uuencode, vconfig, vi, vlock, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat, zcip

Toolbox

- As Busybox is under the GPL, Google developed an equivalent tool, under the BSD license
- Much fewer UNIX commands implemented than Busybox, but other commands to use the Android-specifics mechanism, such as `alarm`, `getprop` or a modified `log`

Commands available in Toolbox in Gingerbread

`alarm, cat, chmod, chown, cmp, date, dd, df, dmesg, exists, getevent, getprop, hd, id, ifconfig, iftop, insmod, ioctl, ionice, kill, ln, log, ls, lsmod, lsof, mkdir, mount, mv, nandread, netstat, newfs_msdos, notify, powerd, printenv, ps, r, readtty, reboot, renice, rm, rmdir, rmmod, rotatefb, route, schedtop, sendevent, setconsole, setkey, setprop, sleep, smd, start, stop, sync, syren, top, umount, uptime, vmstat, watchprops, wipe`

Init

Init

- `init` is the name of the first userspace program
- It is up to the kernel to start it, with PID 1, and the program should never exit during system life
- The kernel will look for init at `/sbin/init`, `/bin/init`, `/etc/init` and `/bin/sh`. You can tweak that with the `init=` kernel parameter
- The role of init is usually to start other applications at boot time, a shell, mount the various filesystems, etc.
- Init also manages the shutdown of the system by undoing all it has done at boot time

Android's init

- Once again, Google has developed his own instead of relying on an existing one.
- However, it has some interesting features, as it can also be seen as a daemon on the system
  - it manages device hotplugging, with basic permissions rules for device files, and actions at device plugging and unplugging
  - it monitors the services it started, so that if they crash, it can restart them
  - it monitors system properties so that you can take actions when a particular one is modified

Init part

- For the initialization part, init mounts the various filesystems (`/proc`, `/sys`, `data`, etc.)
- This allows to have an already setup environment before taking further actions
- Once this is done, it reads the `init.rc` file and executes it

init.rc file interpretation

- Uses a unique syntax, based on events
- There usually are several init configuration files, `init.rc` itself, and `init.<platform_name>.rc`
- While `init.rc` is always taken into account, `init.<platform_name>.rc` is only interpreted if the platform currently running the system reports the same name
- This name is either obtained by reading the file `/proc/cpuinfo` or from the `androidboot.hardware` kernel parameter
- Most of the customizations should therefore go to the platform-specific configuration file rather than to the generic one

Syntax

- Unlike most init script systems, the configuration relies on system event and system property changes, allowed by the daemon part of it
- This way, you can trigger actions not only at startup or at run-level changes like with traditional init systems, but also at a given time during system life

Actions

```
on <trigger>
  command
  command
```

- Here are a few trigger types:
  - `boot`
    - Triggered when init is loaded
  - `<property>=<value>`
    - Triggered when the given property is set to the given value
  - `device-added-<path>`
    - Triggered when the given device node is added or removed
  - `service-exited-<name>`
    - Triggered when the given service exits

Init triggers

- Commands are also specific to Android, with sometimes a syntax very close to the shell one (just minor differences):
- The complete list of triggers, by execution order is:
  - `early-init`
  - `init`
  - `early-fs`
  - `fs`
  - `post-fs`
  - `early-boot`
  - `boot`

Example

```
on boot
   export PATH /sbin:/system/sbin:/system/bin
   export LD_LIBRARY_PATH /system/lib

   mkdir /dev
   mkdir /proc
   mkdir /sys

   mount tmpfs tmpfs /dev
   mkdir /dev/pts
   mkdir /dev/socket
   mount devpts devpts /dev/pts
   mount proc proc /proc
   mount sysfs sysfs /sys

   write /proc/cpu/alignment 4
```

Services

```
service <name> <pathname> [ <argument> ]*
   <option>
   <option>
```

- Services are like daemons
- They are started by init, managed by it, and can be restarted when they exit
- Many options, ranging from which user to run the service as, rebooting in recovery when the service crashes too frequently, to launching a command at service reboot.

Example

```
on device-added-/dev/compass
  start akmd

on device-removed-/dev/compass
  stop akmd

service akmd /sbin/akmd
  disabled
  user akmd
  group akmd
```

Uevent

- Init also manages the runtime events generated by the kernel when hardware is plugged in or removed, like `udev` does on a standard Linux distribution
- This way, it dynamically creates the devices nodes under `/dev`
- You can also tweak its behavior to add specific permissions to the files associated to a new event.
- The associated configuration files are `/ueventd.rc` and `/ueventd.<platform>.rc`

uevent.rc syntax

```
<path>   <permission>  <user>  <group>
```

- Example

```
/dev/bus/usb/*             0660   root      usb
```

Properties

- Init also manages the system properties
- Properties are a way used by Android to share values across the system that are not changing quite often
- Quite similar to the Windows Registry
- These properties are splitted into several files:
  - `/system/build.prop` which contains the properties generated by the build system, such as the date of compilation
  - `/default.prop` which contains the default values for certain key properties, mostly related to the security and permissions for ADB.
  - `/data/local.prop` which contains various properties specific to the device
  - `/data/property` is a folder which purpose is to be able to edit properties at run-time and still have them at the next reboot. This folder is storing every properties prefixed by `persist.` in separate files containing the values.

Modifying Properties

- You can add or modify properties in the build system by using either the `PRODUCT_PROPERTY_OVERRIDES` makefile variable, or by defining your own `system.prop` file in the device directory. Their content will be appended to `/system/build.prop` at compilation time
- Modify the `init.rc` file so that at boot time it exports these properties using the `setprop` command
- Using the API functions such as the Java function `SystemProperties.set`

Permissions on the Properties

- Android, by default, only allows any given process to read the properties.
- You can set write permissions on a particular property or a group of them using the file `system/core/init/property_service.c`

```
/* White list of permissions for setting property services. */
struct {
    const char *prefix;
    unsigned int uid;
    unsigned int gid;
} property_perms[] = {
    { "net.rmnet0.",      AID_RADIO,    0 },
    { "net.dns",          AID_RADIO,    0 },
    { "net.",             AID_SYSTEM,   0 },
    { "dhcp.",            AID_SYSTEM,   0 },
    { "log.",             AID_SHELL,    0 },
    { "service.adb.root", AID_SHELL,    0 },
    { "persist.security.", AID_SYSTEM,   0 },
    { NULL, 0, 0 }
```

Special Properties

- `ro.*` properties are read-only. They can be set only once in the system life-time. You can only change their value by modifying the property files and reboot.
- `persist.*` properties are stored on persistent storage each time they are set.
- `ctl.start` and `ctl.stop` properties used instead of storing properties to start or stop the service name passed as the new value
- `net.change` property holds the name of the last `net.*` property changed.

Various daemons

Vold

- The VOLume Daemon
- Just like init does, monitors new device events
- While init was only creating device files and taking some configured options, `vold` actually only cares about storage devices
- Its roles are to:
  - Auto-mount the volumes
  - Format the partitions on the device
- There is no `/etc/fstab` in Android, but `/system/etc/vold.fstab` has a somewhat similar role

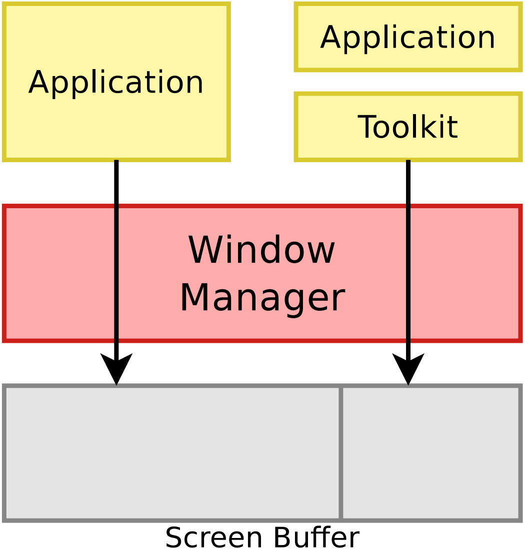
rild

- `rild` is the Radio Interface Layer Daemon
- This daemon drives the telephony stack, both voice and data communication
- When using the voice mode, talks directly to the baseband, but when issuing data transfers, relies on the kernel network stack
- It can handle two types of commands:
  - *Solicited commands*: commands that originate from the user: dial a number, send an SMS, etc.
  - *Unsolicited commands*: commands that come from the baseband: receiving an SMS, a call, signal strength changed, etc.

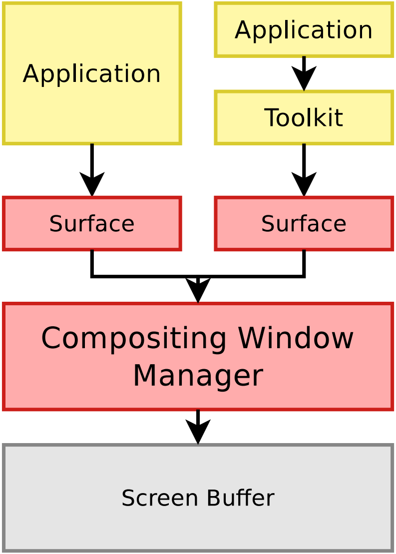
Others

- netd
  - `netd` manages the various network connections: Bluetooth, Wifi, USB
  - Also takes any associated actions: detect new connections, set up the tethering, etc.
  - It really is an equivalent to NetworkManager
  - On a security perspective, it also allows to isolate network-related privileges in a single process
- installd
  - Handles package installation and removal
  - Also checks package integrity, installs the native libraries on the system, etc.

SurfaceFlinger and PixelFlinger


Application
Application
Toolkit
Window
Manager
Screen Buffer

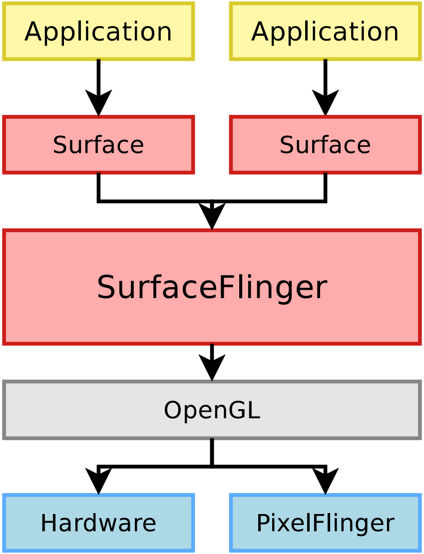
Compositing window managers



SurfaceFlinger

- This difference in design adds some interesting features:
  - Effects are easy to implement, as it's up to the window manager to mangle the various surfaces at will to display them on the screen. Thus, you can add transparency, 3d effects, etc.
  - Improved stability. With a regular window manager, a message is sent to every window to redraw its part of the screen, for example when a window has been moved. But if an application fails to redraw, the windows will become glitchy. This will not happen with a compositing WM, as it will still display the untouched surface.
- SurfaceFlinger is the compositing window manager in Android, providing surfaces to applications and rendering all of them with hardware acceleration.

SurfaceFlinger and PixelFlinger

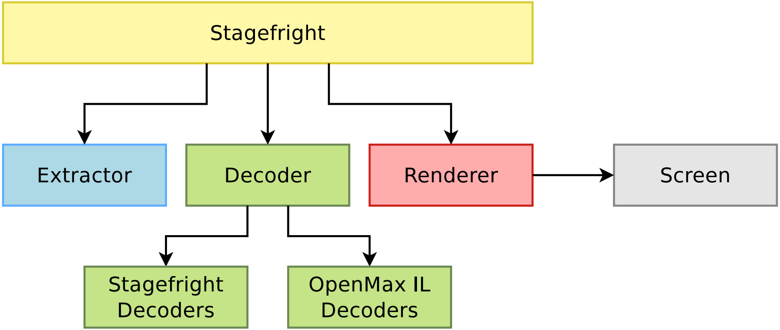


Stagefright

Stagefright

- StageFright is the multimedia playback engine in Android since Eclair
- In its goals, it is quite similar to Gstreamer: Provide an abstraction on top of codecs and libraries to easily play multimedia files
- It uses a plugin system, to easily extend the number of formats supported, either software or hardware decoded

StageFright Architecture



- To add support for a new format, you need to:
  - Develop a new Extractor class, if the container is not supported yet.
  - Develop a new Decoder class, that implements the interface needed by the StageFright core to read the data.
  - Associate the mime-type of the files to read to your new Decoder in the `OMXCodec.cpp` file, in the `kDecoderInfo` array.
    - → No runtime extension of the decoders, this is done at compilation time.

```
static const CodecInfo kDecoderInfo[] = {
    { MEDIA_MIMETYPE_AUDIO_AAC, "OMX.TI.AAC.decode" },
    { MEDIA_MIMETYPE_AUDIO_AAC, "AACDecoder" },
};
```

Dalvik and Zygote

Dalvik

- Dalvik is the virtual machine, executing Android applications
- It is an interpreter written in C/C++, and is designed to be portable, lightweight and run well on mobile devices
- It is also designed to allow several instances of it to be run at the same time while consuming as little memory as possible
- Two execution modes
  - `portable`: the interpreter is written in C, quite slow, but should work on all platforms
  - `fast`: Uses the *mterp* mechanism, to define routines either in assembly or in C optimized for a specific platform. Instruction dispatching is also done by computing the handler address from the opcode number
- It uses the *Apache Harmony* Java framework for its core libraries

Zygote

- Dalvik is started by `Zygote`
- `frameworks/base/cmds/app_process`
- At boot, Zygote is started by init, it then
  - Initializes a virtual machine in its address space
  - Loads all the basic Java classes in memory
  - Starts the system server
  - Waits for connections on a UNIX socket
- When a new application should be started:
  - Android connects to Zygote through the socket to request the start of a new application
  - Zygote forks
  - The child process loads the new application and start executing it

Hardware Abstraction Layer

Hardware Abstraction Layers

- Usually, the kernel already provides a HAL for userspace
- However, from Google's point of view, this HAL is not sufficient and suffers some restrictions, mostly:
  - Depending on the subsystem used in the kernel, the userspace interface differs
  - All the code in the kernel must be GPL-licensed
- Google implemented its HAL with dynamically loaded userspace libraries

Library naming

- It follows the same naming scheme as for init: the generic implementation is called `libfoo.so` and the hardware-specific one `libfoo.hardware.so`
- The name of the hardware is looked up with the following properties:
  - `ro.hardware`
  - `ro.product.board`
  - `ro.board.platform`
  - `ro.arch`
- The libraries are then searched for in the directories:
  - `/vendor/lib/hw`
  - `/system/lib/hw`

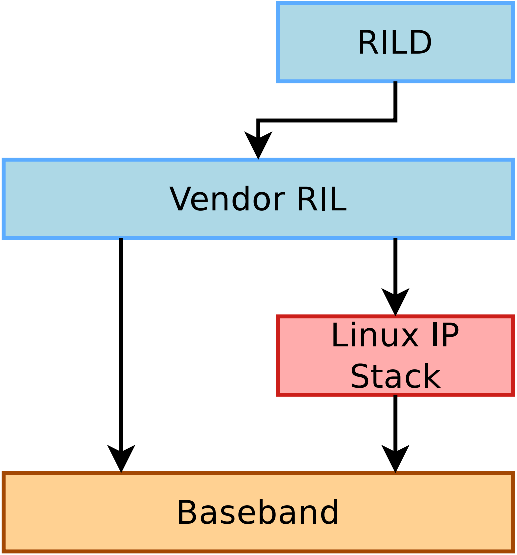
Various layers

- Audio (`libaudio.so`) configuration, mixing, noise cancellation, etc.
  - `hardware/libhardware_legacy/include/hardware_legacy/AudioHardwareInterface.h`
- Graphics (`gralloc.so`, `copybit.so`, `libhgl.so`) handles graphic memory buffer allocations, OpenGL implementation, etc.
  - `libhgl.so` should be provided by your vendor
  - `hardware/libhardware/include/gralloc.h`
  - `hardware/libhardware/include/copybit.h`
- Camera (`libcamera.so`) handles the camera functions: autofocus, take a picture, etc.
  - `frameworks/base/include/camera/CameraHardwareInterface.h`

Various layers

- GPS (`libgps.so`) configuration, data acquisition
  - `hardware/libhardware/include/hardware/gps.h`
- Lights (`liblights.so`) Backlight and LEDs management
  - `hardware/libhardware/include/lights.h`
- Sensors (`libsensors.so`) handles the various sensors on the device: Accelerometer, Proximity Sensor, etc.
  - `hardware/libhardware/include/sensors.h`
- Radio Interface (`libril-vendor-version.so`) manages all communication between the baseband and `rild`
  - You can set the name of the library with the `rild.lib` and `rild.libargs` properties to find the library
  - `hardware/ril/include/telephony/ril.h`

Example: rild



JNI

What is JNI?

- A Java framework to call and be called by native applications written in other languages
- Mostly used for:
  - Writing Java bindings to C/C++ libraries
  - Accessing platform-specific features
  - Writing high-performance sections
- It is used extensively across the Android userspace to interface between the Java Framework and the native daemons
- Since Gingerbread, you can develop apps in a purely native way, possibly calling Java methods through JNI

C Code

```c
#include "jni.h"

JNIEXPORT void JNICALL Java_com_example_Print_print(JNIEnv *env,
                                                    jobject obj,
                                                    jstring javaString)
{
    const char *nativeString = (*env)->GetStringUTFChars(env,
                                                         javaString,
                                                         0);
    printf("%s", nativeString);
    (*env)->ReleaseStringUTFChars(env, javaString, nativeString);
}
```

JNI arguments

- Function prototypes are following the template:

```
JNIEXPORT jstring JNICALL Java_ClassName_MethodName
          (JNIEnv*, jobject)
```

- `JNIEnv` is a pointer to the JNI Environment that we will use to interact with the virtual machine and manipulate Java objects within the native methods
- `jobject` contains a pointer to the calling object. It is very similar to `this` in C++

Types

- There is no direct mapping between C Types and JNI types
- You must use the JNI primitives to convert one to his equivalent
- However, there are a few types that are directly mapped, and thus can be used directly without typecasting:

Native Type	JNI Type
unsigned char	jboolean
signed char	jbyte
unsigned short	jchar
short	jshort
long	jint
long long	jlong
float	jfloat
double	jdouble

Java Code

```java
package com.example;

class Print
{
        private static native void print(String str);

        public static void main(String[] args)
        {
                Print.print("HelloWorld!");
        }

        static
        {
                System.loadLibrary("print");
        }
}
```

Calling a method of a Java object from C

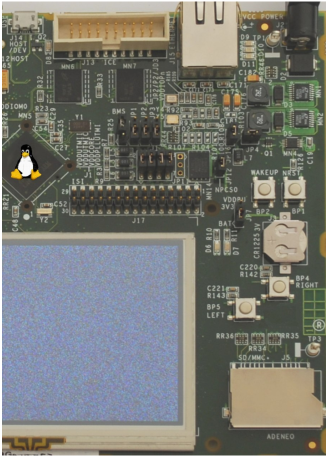
```
JNIEXPORT void JNICALL Java_ClassName_Method(JNIEnv *env,
                                             jobject obj)
{
  jclass cls = (*env)->GetObjectClass(env, obj);
  jmethodID hello = (*env)->GetMethodID(env,
                                        cls,
                                        "hello",
                                        "(V)V");
  if (!hello)
    return;
  (*env)->CallVoidMethod(env, obj, hello);
}
```

Instantiating a Java object from C

```
JNIEXPORT jobject JNICALL Java_ClassName_Method(JNIEnv *env,
                                                jobject obj)
{
    jclass cls = env->FindClass("java/util/ArrayList");
    jmethodID init = env->GetMethodID(cls,
                                      "<init>",
                                      "()V");
    jobject array = env->NewObject(cls, init);

    return array;
}
```

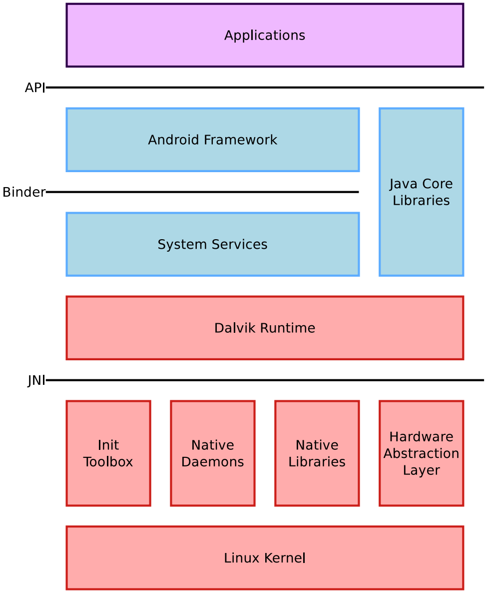
Android Framework and Applications

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded

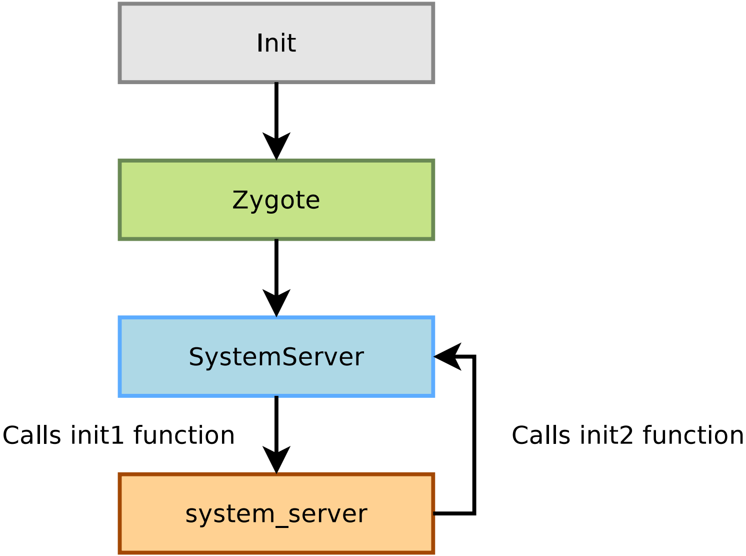


Service Manager and Various Services

Whole Android Stack



System Server boot



The first step: `system_server.c`

- Located in `frameworks/base/cmds/system_server`
- Started by `Zygote` through the SystemServer
- Starts all the various native services:
  - `SurfaceFlinger`
  - `SensorService`
  - `AudioFlinger`
  - `MediaPlayerService`
  - `CameraService`
  - `AudioPolicyService`
- It then calls back the SystemServer object's `init2` function to go on with the initialization

Java Services Initialization

- Located in `frameworks/base/services/java/com/android/server/SystemServer.java`
- Starts all the different Java services in a different thread by registering them into the Service Manager
- `PowerManager`, `ActivityManager` (also handles the ContentProviders), `PackageManager`, `BatteryService`, `LightsService`, `VibratorService`, `AlarmManager`, `WindowManager`, `BluetoothService`, `DevicePolicyManager`, `StatusBarManager`, `InputMethodManager`, `ConnectivityService`, `MountService`, `NotificationManager`, `LocationManager`, `AudioService`, `...`
- If you wish to add a new system service, you will need to add it to one of these two parts to register it at boot time

Inter-Process Communication, Binder and AIDLs

- On modern systems, each process has its own address space, allowing to isolate data
- This allows for better stability and security: only a given process can access its address space. If another process tries to access it, the kernel will detect it and kill this process.
- However, interactions between processes are sometimes needed, that's what IPCs are for.
- On classic Linux systems, several IPC mechanisms are used:
  - Signals
  - Semaphores
  - Sockets
  - Message queues
  - Pipes
  - Shared memory
- Android, however, uses mostly:
  - Binder
  - Ashmem and Sockets

Binder 1/2

- Uses shared memory for high performance
- Uses reference counting to garbage collect objects no longer in use
- Data are sent through *parcels*, which is some kind of serialization
- Used across the whole system, e.g., clients connect to the window manager through Binder, which in turn connects to SurfaceFlinger using Binder
- Each object has an *identity*, which does not change, even if you pass it to other processes.

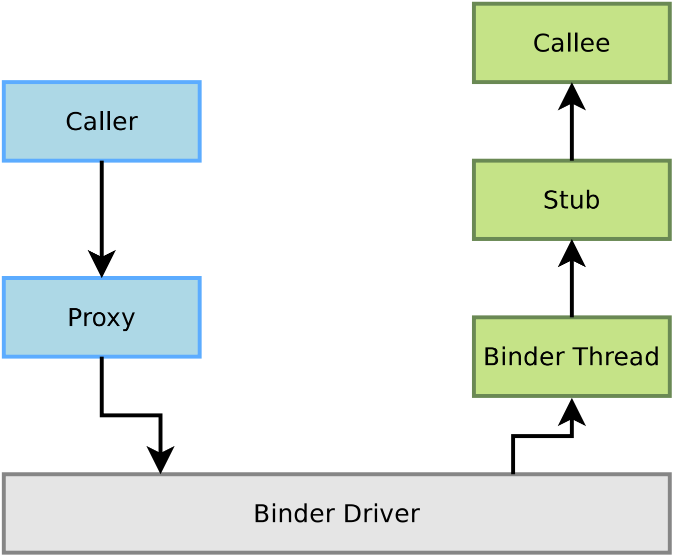
Binder 2/2

- This is useful if you want to separate components in distinct processes, or to manage several components of a single process (i.e. Activity's Windows).
- Object identity is also used for security. Some token passed correspond to specific permissions. Another security model to enforce permissions is for every transaction to check on the calling UID.
- Binder also supports one-way and two-way messages

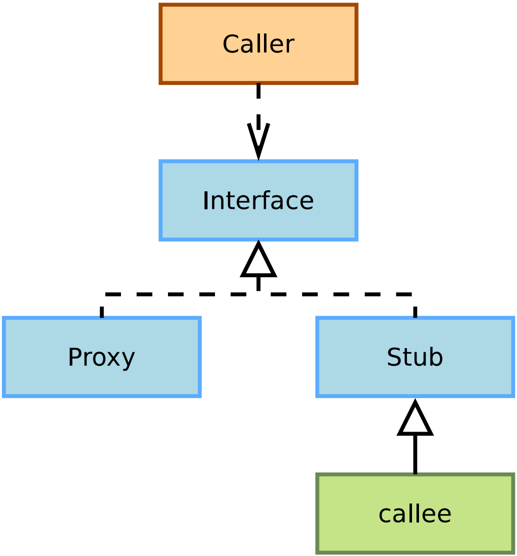
Binder terminology

- *The Binder*
  - The overall Binder Architecture
- *Binder Interface*
  - A well-defined set of methods and properties other can call, and that should be implemented by *a* binder
- *A Binder*
  - A particular implementation of a Binder interface
- *Binder Object*
  - An instance of a class that implements a Binder interface

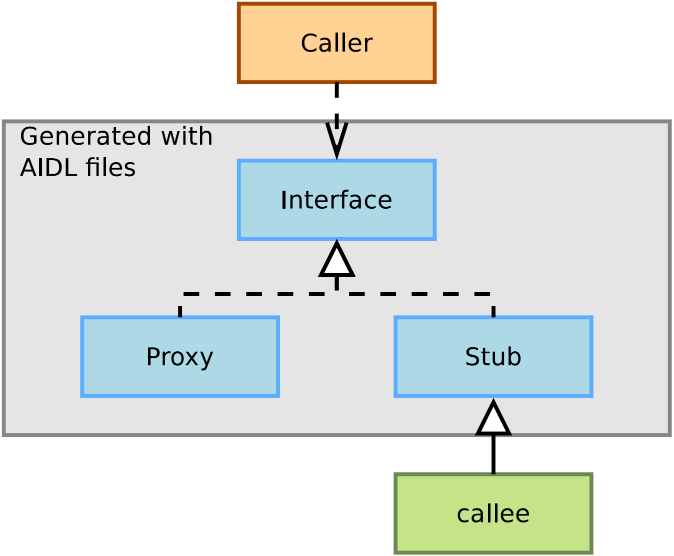
Binder Mechanism



Binder Implementation 1/2



Binder Implementation 2/2



Android Interface Definition Language (AIDL)

- Very similar to any other Interface Definition Language you might have encountered
- Describes a programming interface for the client and the server to communicate using IPCs
- Looks a lot like Java interfaces. Several types are already defined, however, and you can't extend this like what you can do in Java:
  - All Java primitive types (`int`, `long`, `boolean`, etc.)
  - `String`
  - `CharSequence`
  - `Parcelable`
  - `List` of one of the previous types
  - `Map`

```
package com.example.android;

interface IRemoteService {
    void HelloPrint(String aString);
}
```

Parcelable Objects

- If you want to add extra objects to the AIDLs, you need to make them implement the `Parcelable` interface
- Most of the relevant Android objects already implement this interface.
- This is required to let Binder know how to serialize and deserialize these objects
- However, this is not a general purpose serialization mechanism. Underlying data structures may evolve, so you should not store parcelled objects to persistent storage
- Has primitives to store basic types, arrays, etc.
- You can even serialize file descriptors!

Implement Parcelable Classes

- To make an object parcelable, you need to:
  - Make the object implement the `Parcelable` interface
  - Implement the `writeToParcel` function, which stores the current state of the object to a `Parcel` object
  - Add a static field called `CREATOR`, which implements the `Parcelable.Creator` interface, and takes a `Parcel`, deserializes the values and returns the object
  - Create an AIDL file that declares your new parcelable class
- You should also consider `Bundles`, that are type-safe key-value containers, and are optimized for reading and writing values

Intents

- Intents are a high-level use of Binder
- They describe the intention to do something
- They are used extensively across Android
  - Activities, Services and BroadcastReceivers are started using intents
- Two types of intents:

  explicit — The developer designates the target by its name

  implicit — There is no explicit target for the Intent. The system will find the best target for the Intent by itself, possibly asking the user what to do if there are several matches

Various Java Services

Android Java Services

- There are lots of services implemented in Java in Android
- They abstract most of the native features to make them available in a consistent way
- You get access to the system services using the `Context.getSystemService()` call
- You can find all the accessible services in the documentation for this function

ActivityManager

- Manages everything related to Android applications
  - Starts Activities and Services through Zygote
  - Manages their lifecycle
  - Fetches content exposed through content providers
  - Dispatches the implicit intents
  - Adjusts the Low Memory Killer priorities
  - Handles non responding applications

PackageManager

- Exposes methods to query and manipulate already installed packages, so you can:
  - Get the list of packages
  - Get/Set permissions for a given package
  - Get various details about a given application (name, uids, etc)
  - Get various resources from the package
- You can even install/uninstall an apk
  - `installPackage`/`uninstallPackage` functions are hidden in the source code, yet `public`.
  - You can't compile code that is calling directly these functions and they are not documented anywhere except in the code
  - But you can call them through the Java `Reflection` API, if you have the proper permissions of course

- Abstracts the Wakelocks functionality
- Defines several states, but when a wakelock is grabbed, the CPU will always be on
  - `PARTIAL_WAKE_LOCK`
    - Only the CPU is on, screen and keyboard backlight are off
  - `SCREEN_DIM_WAKE_LOCK`
    - Screen backlight is partly on, keyboard backlight is off
  - `SCREEN_BRIGHT_WAKE_LOCK`
    - Screen backlight is on, keyboard backlight is off
  - `FULL_WAKE_LOCK`
    - Screen and keyboard backlights are on

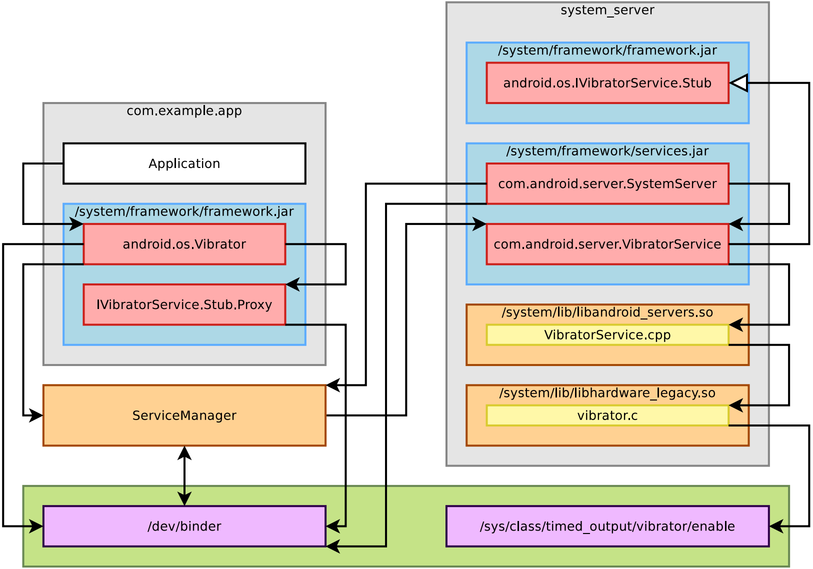
AlarmManager

- Abstracts the Android timers
- Allows to set a one time timer or a repetitive one
- When a timer expires, the AlarmManager grabs a wakelock, sends an Intent to the corresponding application and releases the wakelock once the Intent has been handled

ConnectivityManager and WifiManager

- ConnectivityManager
  - Manages the various network connections
    - Falls back to other connections when one fails
    - Notifies the system when one becomes available/unavailable
    - Allows the applications to retrieve various information about connectivity
- WifiManager
  - Provides an API to manage all aspects of WiFi networks
    - List, modify or delete already configured networks
    - Get information about the current WiFi network if any
    - List currently available WiFi networks
    - Sends Intents for every change in WiFi state

Example: Vibrator Service



Extend the framework

Why extend it?

- You might want to extend the existing Android framework to add new features or allow other applications to use specific devices available on your hardware
- As you have the code, you could just hack the source to make the framework suit your needs
- This is quite problematic however:
  - You might break the API, introduce bugs, etc
  - Google requires you not to modify the Android public API
  - It is painful to track changes across the tree, to port the changes to new versions
  - You don't always want to have such extensions for all your products
- As usual with Android, there's a device-specific way of extending the framework: `PlatformLibraries`

PlatformLibraries

- The modifications are just plain Java libraries
- You can declare any namespace you want, do whatever code you want.
- However, they are bundled as raw Java archives, so you cannot embed resources in the modifications
- If you would still do this, you can add them to `frameworks/base/res`, but you have to hide them
- When using the Google Play Store, all the libraries including these ones are submitted to Google, so that it can filter out apps relying on libraries not available on your system
- To avoid any application to link to any jar file, you have to declare both in your application and in your library that you will use and add a custom library
- The library's xml permission file should go into the `/system/etc/permissions` folder

PlatformLibrary Makefile

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES := \
            $(call all-subdir-java-files)

LOCAL_MODULE_TAGS := optional

LOCAL_MODULE:= com.example.android.pl

include $(BUILD_JAVA_LIBRARY)
```

PlatformLibrary permissions file

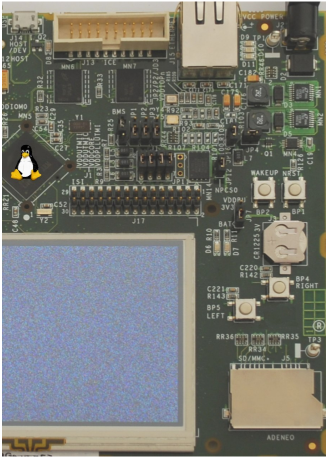
```xml
<?xml version="1.0" encoding="utf-8"?>
<permissions>
    <library name="com.example.android.pl"
        file="/system/framework/com.example.android.pl.jar"/>
</permissions>
```

PlatformLibrary Client Makefile

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional

LOCAL_PACKAGE_NAME := PlatformLibraryClient

LOCAL_SRC_FILES := $(call all-java-files-under, src)

LOCAL_JAVA_LIBRARIES := com.example.android.pl

include $(BUILD_PACKAGE)
```

Android Filesystem

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



Contents

Filesystem organization on GNU/Linux

- On most Linux based distributions, the filesystem layout is defined by the Filesystem Hierarchy Standard
- The FHS defines the main directories and their contents

/bin	Essential command binaries
/boot	Bootloader files, i.e. kernel images and related stuff
/etc	Host-specific system-wide configuration files.

- Android follows an orthogonal path, storing its files in folders not present in the FHS, or following it when it uses a defined folder

Filesystem organization on Android

- Instead, the two main directories used by Android are

  /system
  : Immutable directory coming from the original build. It contains native binaries and libraries, framework jar files, configuration files, standard apps, etc.

  /data
  : is where all the changing content of the system are put: apps, data added by the user, data generated by all the apps at runtime, etc.

- These two directories are usually mounted on separate partitions, from the root filesystem originating from a kernel RAM disk.
- Android also uses some usual suspects: `/proc`, `/dev`, `/sys`, `/etc`, `sbin`, `/mnt` where they serve the same function they usually do

/system

./app	All the pre-installed apps
./bin	Binaries installed on the system (toolbox, vold, surfaceflinger)
./etc	Configuration files
./fonts	Fonts installed on the system
./framework	Jar files for the framework
./lib	Shared objects for the system libraries
./modules	Kernel modules
./xbin	External binaries

Other directories

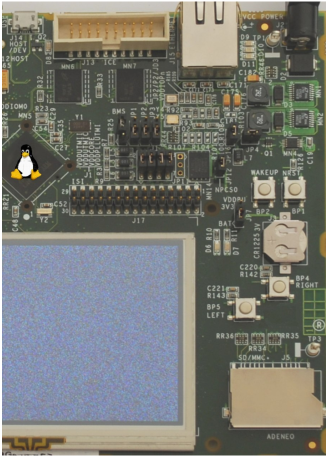
- Like we said earlier, Android most of the time either uses directories not in the FHS, or directories with the exact same purpose as in standard Linux distributions (`/dev`, `/proc`), therefore avoiding collisions. `/sys`)
- There is some collision though, for `/etc` and `/sbin`, which are hopefully trimmed down
- This allows to have a full Linux distribution side by side with Android with only minor tweaks

android_filesystem_config.h

- Located in `system/core/include/private/`
- Contains the full filesystem setup, and is written as a C header
  - UID/GID
  - Permissions for system directories
  - Permissions for system files
- Processed at compilation time to enforce the permissions throughout the filesystem
- Useful in other parts of the framework as well, such as ADB

Developing and Debugging with ADB

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



Corrections, suggestions, contributions and translations are welcome!

Introduction

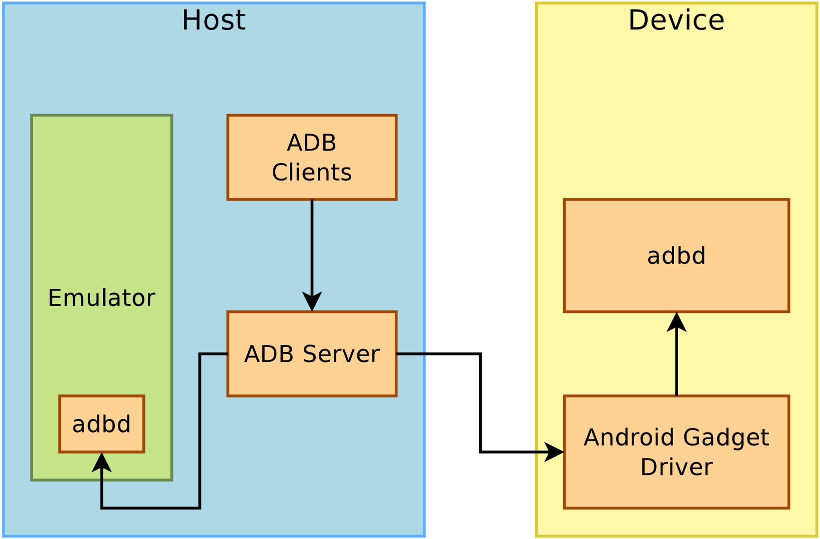
ADB

- Usually on embedded devices, debugging and is done either through a serial port on the device or JTAG for low-level debugging
- This setup works well when developing a new product that will have a static system. You develop and debug a system on a product with serial and JTAG ports, and remove these ports from the final product.
- For mobile devices, where you will have applications developers that are not in-house, this is not enough.
- To address that issue, Google developed ADB, that runs on top of USB, so that another developer can still have debugging and low-level interaction with a production device.

Implementation

- The code is split in 3 components:
  - ADBd, the part that runs on the device
  - ADB server, which is run on the host, acts as a proxy and manages the connection to ADBd
  - ADB clients, which are also run on the host, and are what is used to send commands to the device
- ADBd can work either on top of TCP or USB.
  - For USB, Google has implemented a driver using the USB gadget and the USB composite frameworks as it implements either the ADB protocol and the USB Mass Storage mechanism.
  - For TCP, ADBd just opens a socket
- ADB can also be used as a transport layer between the development platform and the device, disregarding whether it uses USB or TCP as underneath layer

ADB Architecture



Use of ADB

ADB commands: Basics

start-server	Starts the ADB server on the host
kill-server	Kills the ADB server on the host
devices	Lists accessible devices
connect	Connects to a remote ADBd using TCP port 5555 by default
disconnect	Disconnects from a connected device
help	Prints available commands with help information
version	Prints the version number

ADB commands: Files and applications

push Copies a local file to the device

pull Copies a remote file from the device

sync There are three cases here:

- If no argument is passed, copies the local directories `system` and `data` if they differ from `/system` and `/data` on the target.
- If either `system` or `data` is passed, syncs this directory with the associated partition on the device
- Else, syncs the given folder

install Installs the given Android package (apk) on the device

uninstall Uninstalls the given package name from the device

ADB commands: Debugging

logcat Prints the device logs. You can filter either on the source of the logs or their on their priority level

shell Runs a remote shell with a command line interface. If an argument is given, runs it as a command and prints out the result

bugreport Gets all the relevant information to generate a bug report from the device: logs, internal state of the device, etc.

jdwp Lists the processes that support the JDWP protocol

ADB commands: Scripting 1/2

wait-for-device Blocks until the device gets connected to ADB. You can also add additional commands to be run when the device becomes available.

get-state Prints the current state of the device, `offline`, `bootloader` or `device`

get-serialno Prints the serial number of the device

remount Remounts the `/system` partition on the device in read/write mode

ADB commands: Scripting 2/2

reboot
: Reboots the device. `bootloader` and `recovery` arguments are available to select the operation mode you want to reboot to.

reboot-bootloader
: Reboots the device into the bootloader

root
: Restarts ADBd with root permissions on the device
  - Only if the `ro.secure` property is to 1 to force ADB into user mode, and `ro.debuggable` is set to 1 to allow to restart ADB as root

usb
: Restarts ADBd listening on USB

tcpip
: Restarts ADBd listening on TCP on the given port

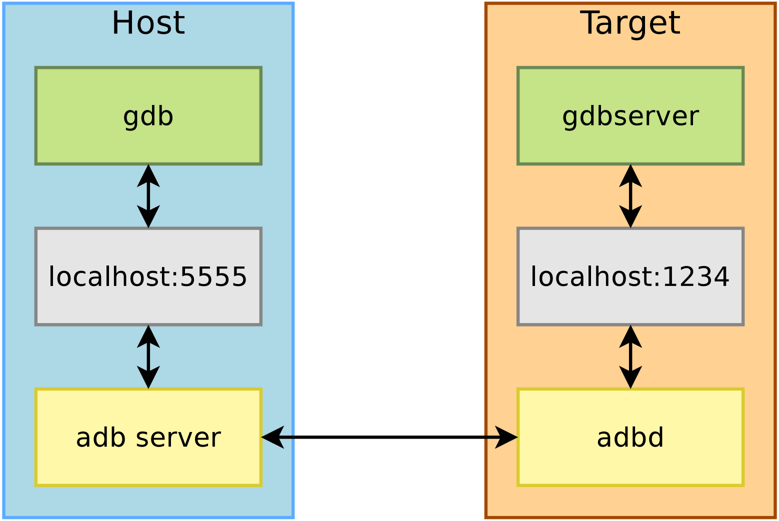
ADB commands: Easter eggs

lolcat
: Alias to `adb logcat`

hell
: Equivalent to `adb shell`, with a different color scheme

Examples

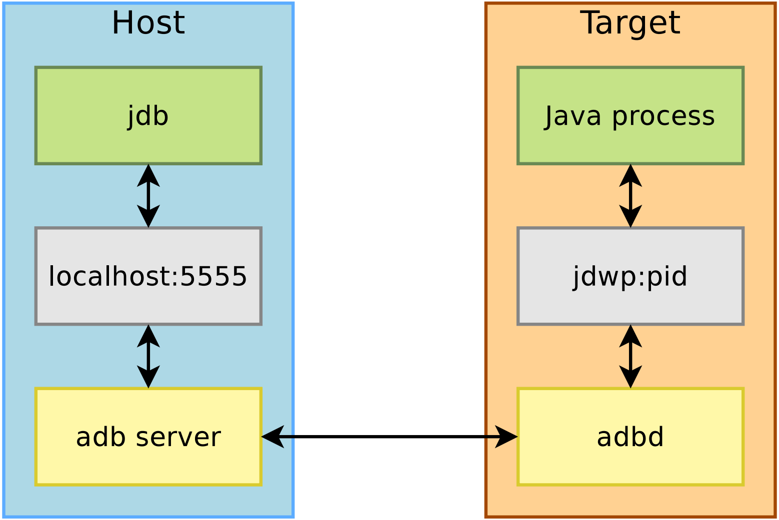
ADB forward and gdb



`adb forward tcp:5555 tcp:1234`

See also `gdbclient`

ADB forward and jdb

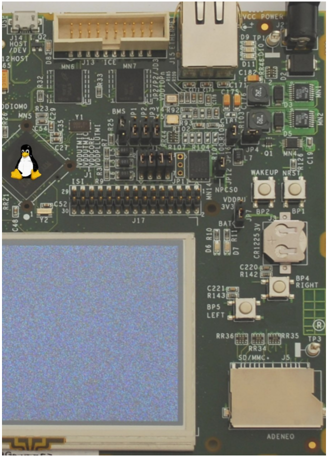


`adb forward tcp:5555 jdwp:4242`

Various commands

- Wait for a device and install an application
  - `adb wait-for-device install foobar.apk`
- Test an application by sending random user input
  - `adb shell monkey -v -p com.free-electrons.foobar 500`
- Filter system logs
  - `adb logcat ActivityManager:I FooBar:D *:S`
  - You can also set the `ANDROID_LOG_TAGS` environment variable on your workstation
- Access other log buffers
  - `adb logcat -b radio`

Android Application Development



Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded

Basics

Android applications

- Android applications are written mostly in Java using Google's SDK
- Applications are bundled into an Android PacKage (`.apk` files) which are archives containing the compiled code, data and resources for the application, so applications are completely self-contained
- You can install applications either through a market (Google Play Store, Amazon Appstore, F-Droid, etc) or manually (through ADB or a file manager)
- Of course, everything we have seen so far is mostly here to provide a nice and unified environment to application developers

Applications Security

- Once installed, applications live in their own sandbox, isolated from the rest of the system
- The system assigns a Linux user to every application, so that every application has its own user/group
- It uses this UID and files permissions to allow the application to access only its own files
- Each process has its own instance of Dalvik, so code is running isolated from other applications
- By default, each application runs in its own process, which will be started/killed during system life
- Android uses the *principle of least privilege*. Each application by default has only access to what it requires to work.
- However, you can request extra permissions, make several applications run in the same process, or with the same UID, etc.

Applications Components

- Components are the basic blocks of each application
- You can see them as entry points for the system in the application
- There is four types of components:
  - Activities
  - Broadcast Receivers
  - Content Providers
  - Services
- Every application can start any component, even located in other applications. This allows to share components easily, and have very little duplication. However, for security reasons, you start it through an Intent and not directly
- When an application requests a component, the system starts the process for this application, instantiates the needed class and runs that component. We can see that there is no single point of entry in an application like `main()`

Application Manifest

- To declare the components present in your application, you have to write a XML file, `AndroidManifest.xml`
- This file is used to:
  - Declare available components
  - Declare which permissions these components need
  - Revision of the API needed
  - Declare hardware features needed
  - Libraries required by the components

Manifest HelloWorld

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.example.android">
    <application>
        <activity android:name=".ExampleActivity"
                  android:label="@string/example_label">
            <intent-filter>
                <action android:name="android.intent.actio
                <category android:name="android.intent.cate
            </intent-filter>
        </activity>
        <uses-library android:name="com.example.android.pl"
    </application>
</manifest>
```

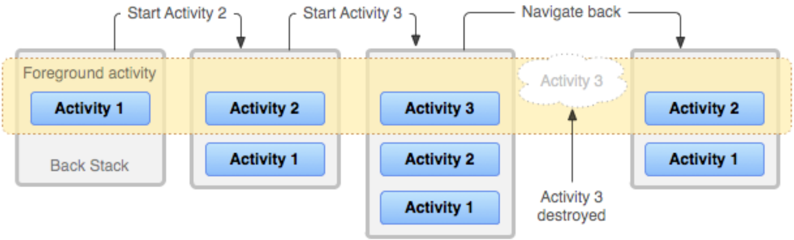
NDK

- Google also provides a NDK to allow developers to write native code
- While the code is not run by Dalvik, the security guarantees are still there
- Allows to write faster code or to port existing C code to Android more easily
- Since Gingerbread, you can even code a whole application without writing a single line of Java
- It is still packaged in an apk, with a manifest, etc.
- However, there are some drawbacks, the main one being that you can't access the resources mechanism available from Java

Activities

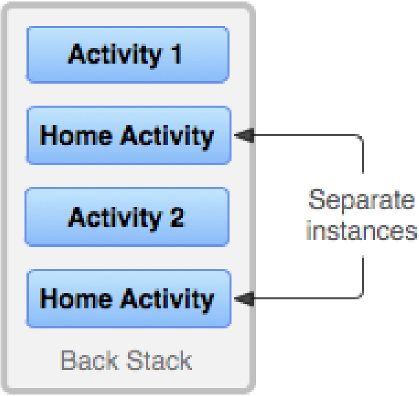
- Activities are a single screen of the user interface of an application
- They are assembled to provide a consistent interface. If we take the example of an email application, we will have:
  - An activity listing the received mails
  - An activity to compose a new mail
  - An activity to read a mail
- Other applications might need one of these activities. To continue with this example, the Camera application might want to start the composing activity to share the just-shot picture
- It is up to the application developer to advertise available activities to the system
- When an activity starts a new activity, the latter replaces the former on the screen and is pushed on the *back stack* which holds the last used activities, so when the user is done with the newer activity, it can easily go back to the previous one

Back Stack



Credits: `http://developer.android.com`

Back Stack



Credits: `http://developer.android.com`

- As there is no single entry point and as the system manages the activities, activities have to define callbacks that the system can call at some point in time
- Activities can be in one of the three states on Android

  Running The activity is on the foreground and has focus

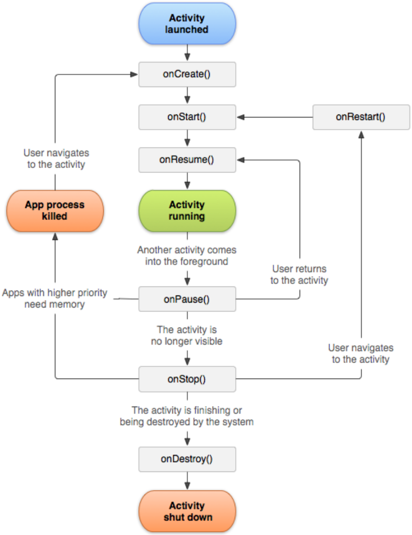
  Paused The activity is still visible on the screen but no longer has focus. It can be destroyed by the system under very heavy memory pressure

  Stopped The activity is no longer visible on the screen. It can be killed at any time by the system

Activity Lifecycle 2/3

- There are callbacks for every change from one of these states to another
- The most important ones are `onCreate` and `onPause`
- All components of an application run in the same thread. If you do long operations in the callbacks, you will block the entire application (UI included). You should always use threads for every long-running task.

Activity Lifecycle 3/3



Credits: `http://developer.android.com`

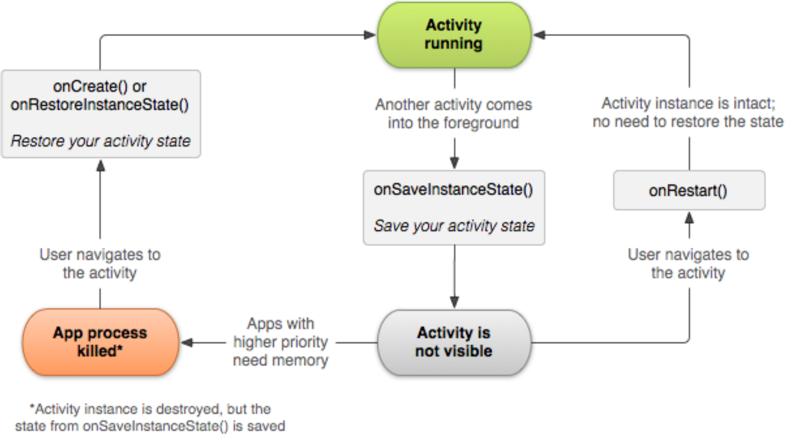
Saving Activity State 1/2

- As applications tend to be killed and restarted quite often, we need a way to store our internal state when killed and reload it when restarted
- Once again, this is done through callbacks
- Before killing the application, the system calls the `onSaveInstanceState` callback and when restarting it, it calls `onRestoreInstanceState`
- In both cases, it provides a Bundle as argument to allow the activity to store what's needed and reload it later, with little overhead

Saving Activity State 2/2

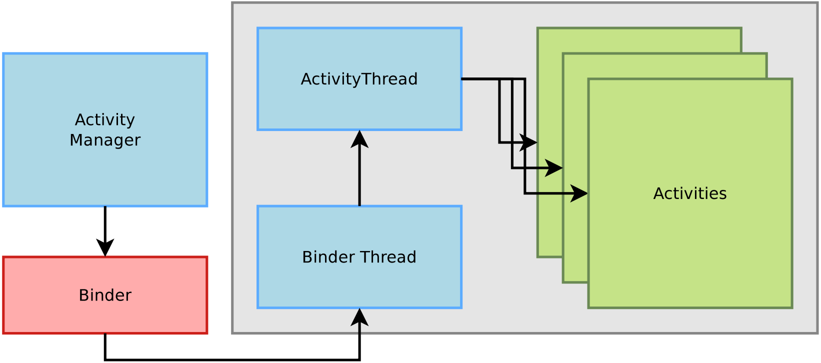
- This make the creation/suppression of activities flawless for the user, while allowing to save as much memory as we need
- These callbacks are not always called though. If the activity is killed because the user left it in a permanent way (through the back button), it won't be called
- By default, these activities are also called when rotating the device, because the activity will be killed and restarted by the system to load new resources

Activity Lifecycle



Credits: `http://developer.android.com`

Activity Callbacks



Credits: `http://developer.android.com`

Activity HelloWorld

```java
public class ExampleActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.i("ExampleActivity", "Activity created!");
    }
    protected void onStart() {
        super.onStart();
    }
    protected void onResume() {
        super.onResume();
    }
    protected void onPause() {
        super.onPause();
    }
    protected void onStop() {
        super.onStop();
    }
    protected void onDestroy() {
        super.onDestroy();
    }
}
```

Services

Services

- Services are components running in the background
- They are used either to perform long running operations or to work for remote processes
- A service has no user interface, as it is supposed to run when the user does something else
- From another component, you can either work with a service in a synchronous way, by *binding* to it, or asynchronous, by *starting* it

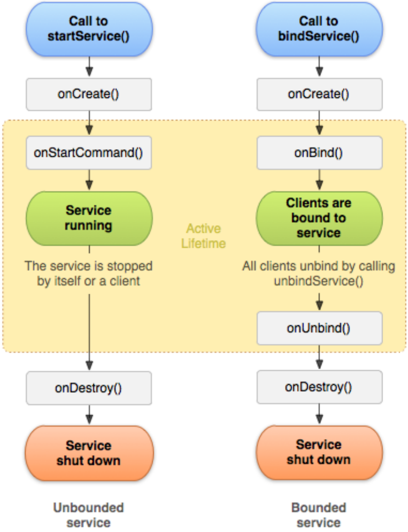
Service Manifest

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.example.android">
    <application>
        <service android:name=".ExampleService"/>
    </application>
</manifest>
```

Services Types

- We can see services as a set including:
  - Started Services, that are created when other components call `startService`. Such a service runs as long as needed, whether the calling component is still alive or not, and can stop itself or be stopped. When the service is stopped, it is destroyed by the system
    - You can also subclass `IntentService` to have a started service. However, while much easier to implement, this service will not handle multiple requests simultaneously.
  - Bound Services, that are bound to by other components by calling `bindService`. They offer a client/server like interface, interacting with each other. Multiple components can bind to it, and a service is destroyed only when no more components are bound to it
- Services can be of both types, given that callbacks for these two do not overlap completely
- Services are started by passing Intents either to the `startService` or `bindService` commands

Services Lifecycle

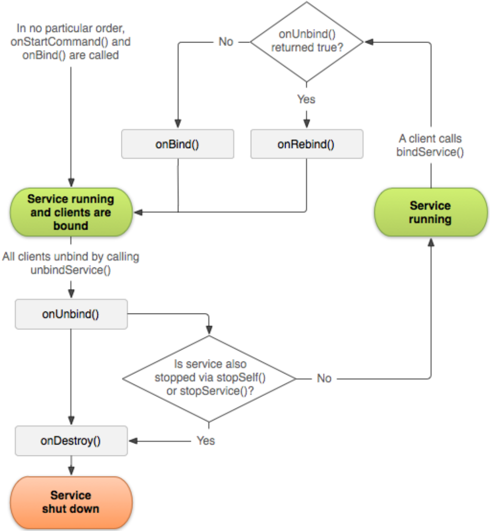


Credits: `http://developer.android.com`

Bound Services

- There are three possible ways to implement a bound service:
  - By extending the `Binder` class. It works only when the clients are local and run in the same process though.
  - By using a `Messenger`, that will provide the interface for your service to remote processes. However, it does not perform multi-threading, all requests are queued up.
  - By writing your own AIDL file. You will then be able to implement your own interface and write thread-safe code, as you are very likely to receive multiple requests at once

Bound Services and Started Lifecycle



Credits: `http://developer.android.com`

Content Providers

Content Providers

- They provide access to organized data in a manner quite similar to relational databases
- They allow to share data with both internal and external components and centralize them
- Security is also enforced by permissions like usual, but they also do not allow remote components to issue arbitrary requests like what we can do with relational databases
- Instead, Content Providers rely on URIs to allow for a restricted set of requests with optional parameters, only permitting the user to filter by values and by columns
- You can use any storage back-end you want, while exposing a quite neutral and consistent interface to other applications

Content URIs

- URIs are often built with the following pattern:
  - `content://<package>.provider/<path>` to access particular tables
  - `content://<package>.provider/<path>/<id>` to access single rows inside the given table
- Facilities are provided to deal with these
  - On the application side:
    - `ContentUri` to append and manage numerical IDs in URIs
    - `Uri.Builder` and `Uri` classes to deal with URIs and strings
  - On the provider side:
    - `UriMatcher` associates a pattern to an ID, so that you can easily match incoming URIs, and use switch over them.

Implementing a Content Provider

```java
public class ExampleProvider extends ContentProvider {
    private static final UriMatcher sUriMatcher;

    static {
        sUriMatcher.addURI("com.example.android.provider", "table1", 1);
        sUriMatcher.addURI("com.example.android.provider", "table1/#", 2);
    }

    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {

        switch (sUriMatcher.match(uri)) {
        default:
            System.out.println("Hello World!");
            break;
        }
    }
```

```
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }

    public int update(Uri uri, ContentValues values, String selection,
                      String[] selectionArgs) {
        return 0;
    }

    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }

    public boolean onCreate() {
        return true;
    }
}
```

Managing the Intents

- Intents are basically a bundle of several pieces of information, mostly
  - `Component Name`
    - Contains both the full class name of the target component plus the package name defined in the Manifest
  - `Action`
    - The action to perform or that has been performed
  - `Data`
    - The data to act upon, written as a URI, like `tel://0123456789`
  - `Category`
    - Contains additional information about the nature of the component that will handle the intent, for example the launcher or a preference panel
- The component name is optional. If it is set, the intent will be explicit. Otherwise, the intent will be implicit

Intent Resolution

- When using explicit intents, dispatching is quite easy, as the target component is explicitly named. However, it is quite rare that a developer knows the component name of external applications, so it is mostly used for internal communication.
- Implicit intents are a bit more tricky to dispatch. The system must find the best candidate for a given intent.
- To do so, components that want to receive intents have to declare them in their manifests *Intent filters*, so that the system knows what components it can respond to.
- Components without intent filters will never receive implicit intents, only explicit ones

Intent Filters 1/2

- They are only about notifying the system about handled implicit intents
- Filters are based on matching by category, action and data. Filtering by only one of these three (by category for example) is fine.
  - A filter can list several actions. If an intent action field corresponds to one of the actions listed here, the intent will match
  - It can also list several categories. However, if none of the categories of an incoming intent are listed in the filter, then intent won't match.

Intent Filters 2/2

- You can also use intent matching from your application by using the `query*` methods from the PackageManager to get a matching component from an Intent.
- For example, the launcher application does that to display only activities with filters that specify the category `android.intent.category.LAUNCHER` and the action `android.intent.action.MAIN`

Real Life Manifest Example: Notepad

```xml
<manifest package="com.example.android.notepad">
  <application android:icon="@drawable/app_notes"
               android:label="@string/app_name" >

    <activity android:name="NotesList"
              android:label="@string/title_notes_list">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="android.intent.action.PICK" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Broadcasted intents

- Intents can also be broadcast thanks to two functions:
  - `sendBroadcast` that broadcasts an intent that will be handled by all its handlers at the same time, in an undefined order
  - `sendOrderedBroadcast` broadcasts an intent that will be handled by one handler at a time, possibly with propagation of the result to the next handler, or the possibility for a handler to cancel the broadcast
- Broadcasts are used for system wide notification of important events: booting has completed, a package has been removed, etc.

Broadcast Receivers

- Broadcast receivers are the fourth type of components that can be integrated into an application. They are specifically designed to deal with broadcast intents.
- Their overall design is quite easy to understand: there is only one callback to implement: `onReceive`
- The lifecycle is quite simple too: once the onReceive callback has returned, the receiver is considered no longer active and can be destroyed at any moment
- Thus you must not use asynchronous calls (Bind to a service for example) from the onReceive callback, as there is no way to be sure that the object calling the callback will still be alive in the future.

Processes and Threads

Process Management in Android

- By default in Android, every component of a single application runs in the same process.
- When the system wants to run a new component:
  - If the application has no running component yet, the system will start a new process with a single thread of execution in it
  - Otherwise, the component is started within that process
- If you happen to want a component of your application to run in its own process, you can still do it through the `android:process` XML attribute in the manifest.
- When the memory constraints are high, the system might decide to kill a process to get some memory back. This is done based on the importance of the process to the user. When a process is killed, all the components running inside are killed.

Processes priority

- *Foreground processes* have the topmost priority. They host either
  - An activity the user is interacting with
  - A service bound to such an activity
  - A service running in the foreground (started with `startForeground`)
  - A service running one of its lifecycle callbacks
  - A broadcast receiver running its `onReceive` method
- *Visible processes* host
  - An activity that is no longer in the foreground but still is visible on the screen
  - A service that is bound to a visible activity
- *Service Processes* host a service that has been started by `startService`
- *Background Processes* host activities that are no longer visible to the user
- *Empty Processes*

Threads

- As there is only one thread of execution, both the application components and UI interactions are done in sequential order
- So a long computation, I/O, background tasks cannot be run directly into the main thread without blocking the UI
- If your application is blocked for more than 5 seconds, the system will display an "*Application Not Responding*" dialog, which leads to poor user experience
- Moreover, UI functions are not thread-safe in Android, so you can only manipulate the UI from the main thread.
- So, you should:
  - Dispatch every long operation either to a service or a worker thread
  - Use messages between the main thread and the worker threads to interact with the UI.

- There are two ways of implementing worker threads in Android:
  - Use the standard Java threads, with a class extending `Runnable`
    - This works, of course, but you will need to do messaging between your worker thread and the main thread, either through handlers or through the `View.post` function
  - Use Android's `AsyncTask`
    - A class that has four callbacks: `doInBackground`, `onPostExecute`, `onPreExecute`, `onProgressUpdate`
    - Useful, because only `doInBackground` is called from a worker thread, others are called by the UI thread

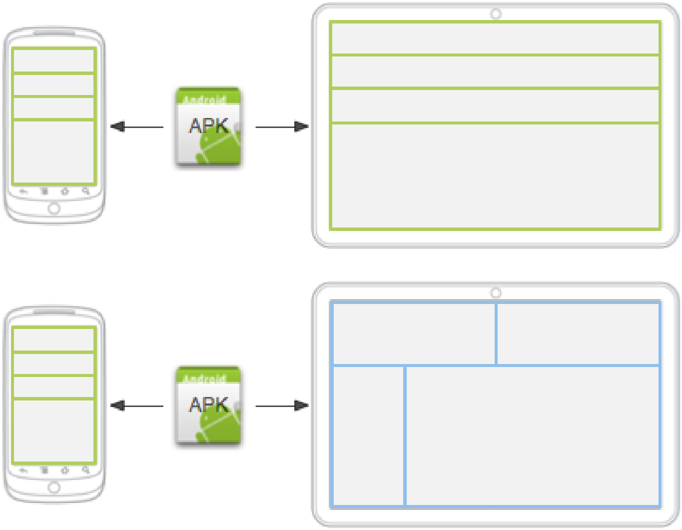
Resources

Applications Resources

- Applications contain more than just compiled source code: images, videos, sound, etc.
- In Android, anything related to the visual appearance of the application is kept separate from the source code: activities layout, animations, menus, strings, etc.
- Resources should be kept in the `res/` directory of your application.
- At compilation, the build tool will create a class `R`, containing references to all the available resources, and associating an ID to it
- This mechanism allows you to provide several alternatives to resources, depending on locales, screen size, pixel density, etc. in the same application, resolved at runtime.

Resources Directory

- All resources are located in the `res/` subdirectory
  - `anim/` contains animation definitions
  - `color/` contains the color definitions
  - `drawable/` contains images, "9-patch" graphics, or XML-files defining drawables (shapes, widgets, relying on a image file)
  - `layout/` contains XML defining applications layout
  - `menu/` contains XML files for the menu layouts
  - `raw/` contains files that are left untouched
  - `values/` contains strings, integers, arrays, dimensions, etc
  - `xml/` contains arbitrary XML files
- All these files are accessed by applications through their IDs. If you still want to use a file path, you need to use the `assets/` folders

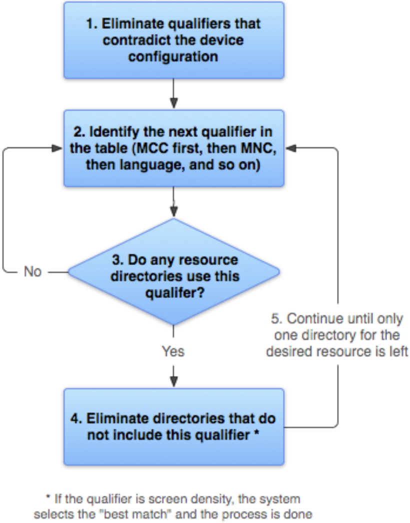


Credits: http://developer.android.com

Alternative Resources

- Alternative resources are provided using extended sub-folder names, that should be named using the pattern <folder_name>-<qualifier>
- There is a number of qualifiers, depending on which case you want to provide an alternative for. The most used ones are probably:
  - locales (`en`, `fr`, `fr-rCA`, ...)
  - screen orientation (`land`, `port`)
  - screen size (`small`, `large`,...)
  - screen density (`mdpi`, `ldpi`, ...)
  - and much others
- You can specify multiple qualifiers by chaining them, separated by dashes. If you want layouts to be applied only when on landscape on high density screens, you will save them into the directory `layout-land-hdpi`

Resources Selection



Credits: `http://developer.android.com`

Data Storage

Data Storage on Android

- An application might need to write to arbitrary files and read from them, for caching purposes, to make settings persistent, etc.
- But the system can't just let you read and write to any random file on the system, this would be a major security flaw
- Android provides some mechanisms to address the two following concerns: allow an application to write to files, while integrating it into the Android security model
- There are four major mechanisms:
  - Preferences
  - Internal data
  - External data
  - Databases

Shared Preferences

- Shared Preferences allows to store and retrieve data in a persistent way
- They are stored using key-value pairs, but can only store basic types: int, float, string, boolean
- They are persistent, so you don't have to worry about them disappearing when the activity is killed
- You can get an instance of the class managing the preferences through the function `getPreferences`
- You may also want several set of preferences for your application and the function `getSharedPreferences` for that
- You can edit them by calling the method `edit` on this instance. Don't forget to call `commit` when you're done!

Internal Storage

- You can also save files directly to the internal storage device
- These files are not accessible by default by other applications
- Such files are deleted when the user removes the application
- You can request a `FileOutputStream` class to such a new file by calling the method `openFileOutput`
- You can pass extra flags to this method to either change the way the file is opened or its permissions
- These files will be created at runtime. If you want to have files at compile time, use resources instead
- You can also use internal storage for caching purposes. To do so, call `getCacheDir` that will return a File object allowing you to manage the cache folder the way you want to. Cache files may be deleted by Android when the system is low on internal storage.

External Storage

- External storage is either the SD card or an internal storage device
- Each file stored on it is world-readable, and the user has direct access to it, since that is the device exported when USB mass storage is used.
- Since this storage may be removable, your application should check for its presence, and that it behaves correctly
- You can either request a sub-folder created only for your application using the `getExternalFilesDir` method, with a tag giving which type of files you want to store in this directory. This folder will be removed at un-installation.
- Or you can request a public storage space, shared by all applications, and never removed by the system, using `getExternalStoragePublicDirectory`
- You can also use it for caching, with `getExternalCacheDir`

SQLite Databases

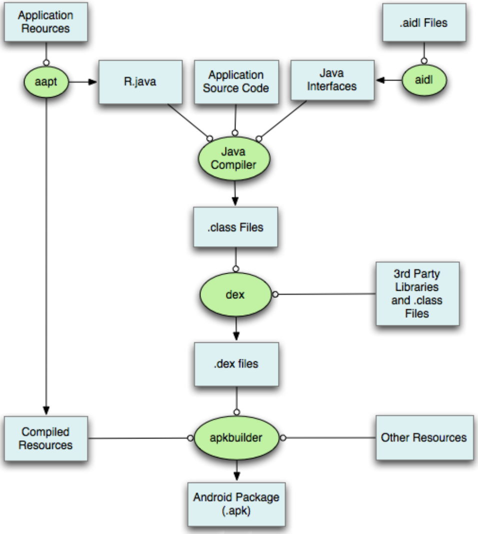
- Databases are often abstracted by Content Providers, that will abstract requests, but Android adds another layer of abstraction
- Databases are managed through subclasses of `SQLiteOpenHelper` that will abstract the structure of the database
- It will hold the requests needed to build the tables, views, triggers, etc. from scratch, as well as requests to migrate to a newer version of the same database if its structure has to evolve.
- You can then get an instance of `SQLiteDatabase` that allows to query the database
- Databases created that way will be only readable from your application, and will never be automatically removed by the system
- You can also manipulate the database using the `sqlite3` command in the shell

Android Packages (apk)

Content of an APK

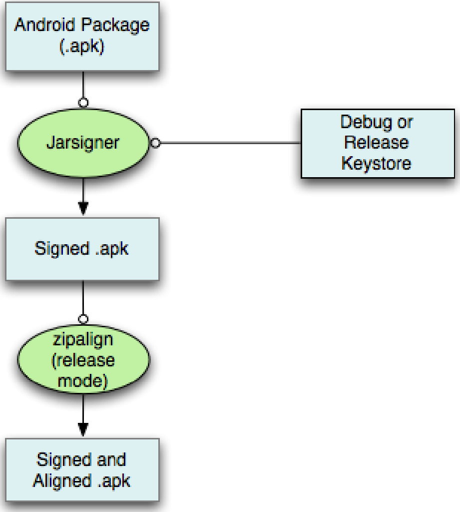
- `META-INF` a directory containing all the Java metadata
  - `MANIFEST.MF` the Java Manifest file, containing various metadata about the classes present in the archive
  - `CERT.RSA` Certificate of the application
  - `CERT.SF` List of resources present in the package and associated SHA-1 hash
- `AndroidManifest.xml`
- `res` contains all the resources, compiled to binary xml for the relevant resources
- `classes.dex` contains the compiled Java classes, to the *Dalvik EXecutable* format, which is a uncompressed format, containing Dalvik instructions
- `resources.arsc` is the resources table. It keeps track of the package resources, associated IDs and packages

APK Building



Credits: `http://developer.android.com`

APK Building

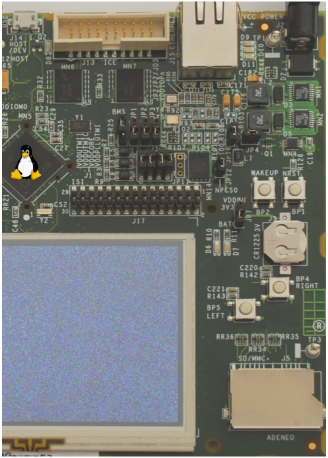


Credits: `http://developer.android.com`

- Write an Android application
- Integrate an application in the Android build system

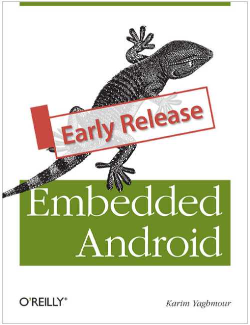
Advices and Resources

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



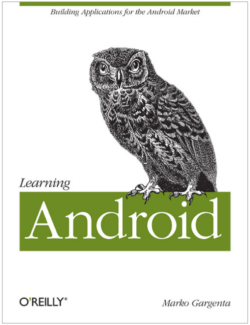
Embedded Android: Porting, Extending, and Customizing, January 2013 (expected)

- By Karim Yaghmour, O'Reilly
- Should be a good reference book and guide on all hidden and undocumented Android internals
- Early version available on-line at O'Reilly
- Our rating: 3 stars



Learning Android, March 2011

- By Marko Gargenta, O'Reilly
- A good reference book and guide on Android application development
- Our rating: 2 stars



Websites

- Android API reference:
  `http://developer.android.com/reference`
- Android Documentation:
  `http://developer.android.com/guide/`
- A good overview on how the various parts of the system are put together to maintain a highly secure system
  `http://source.android.com/tech/security/`

Conferences

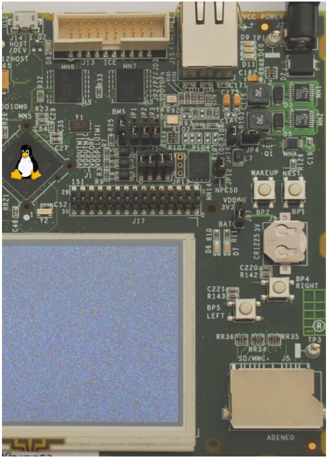
Useful conferences featuring Android topics:

- Android Builders Summit:
  `https://events.linuxfoundation.org/events/android-builders-summit`
  Organized by the Linux Foundation in California (in the Silicon Valley) in early Spring. Many talks about the whole Android stack. Presentation slides are freely available on the Linux Foundation website.
- Embedded Linux Conference:
  `http://embeddedlinuxconference.com/`
  Organized by the Linux Foundation: California (Silicon Valley, Spring), in Europe (Fall). Mostly about kernel and userspace Linux development in general, but always some talks about Android. Presentation slides freely available
- Don't miss our free conference videos on `http://free-electrons.com/community/videos/conferences/`!

Embedded Linux driver development

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier

Free Electrons, Adeneo Embedded



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`

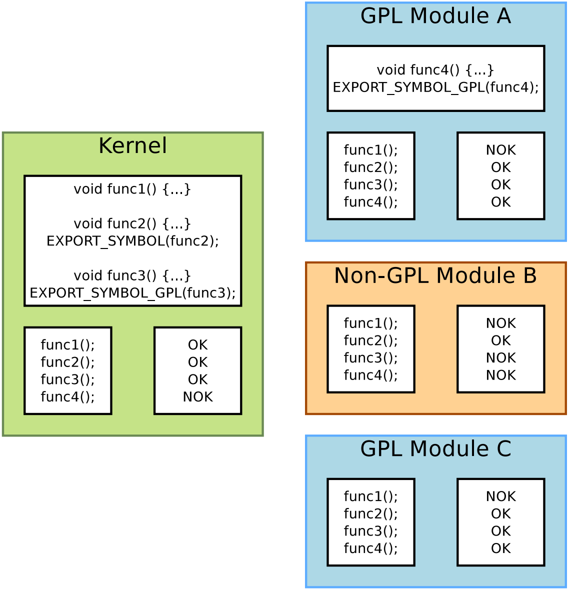
Hello Module Explanations

- 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

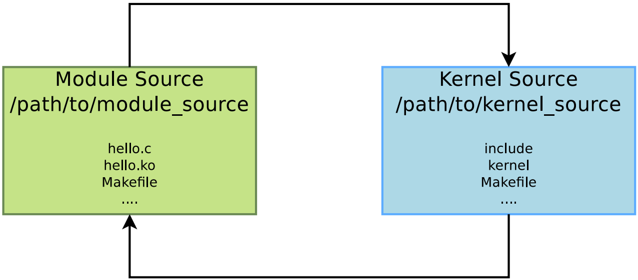
Compiling an out-of-tree Module 1/2

- 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

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

MODULE_LICENSE("GPL");

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

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

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

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`

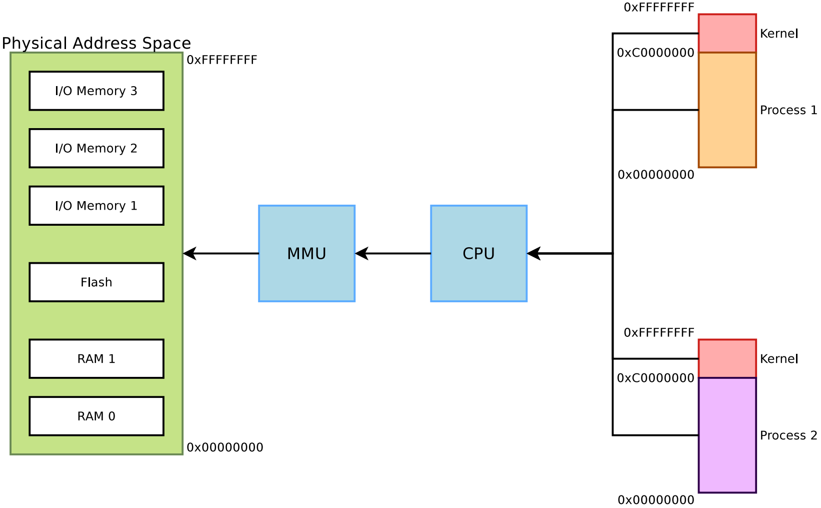
Declaring a module parameter

```
#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);
```

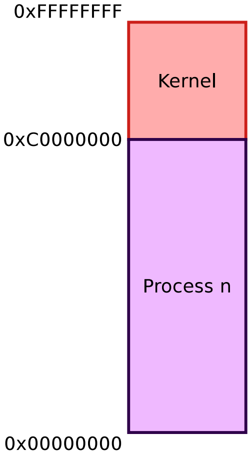
Modules parameter arrays are also possible with `module_param_array()`, but they are less common.

Memory Management

Physical and Virtual Memory

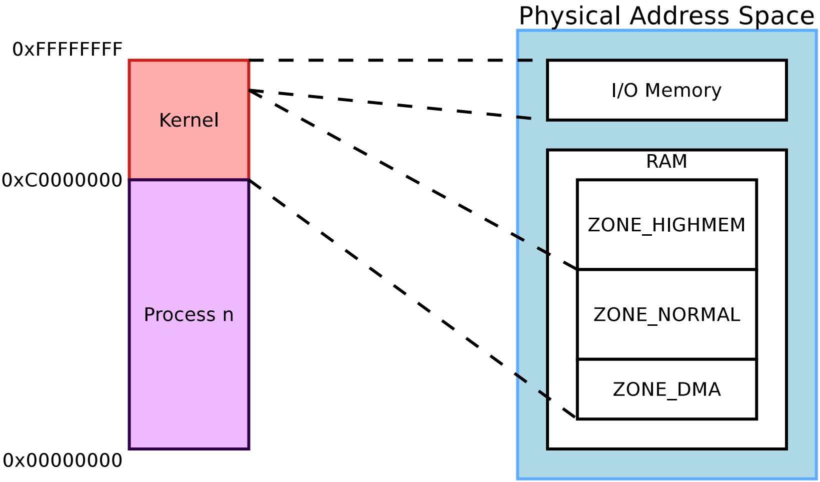


Virtual Memory Organization



- 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

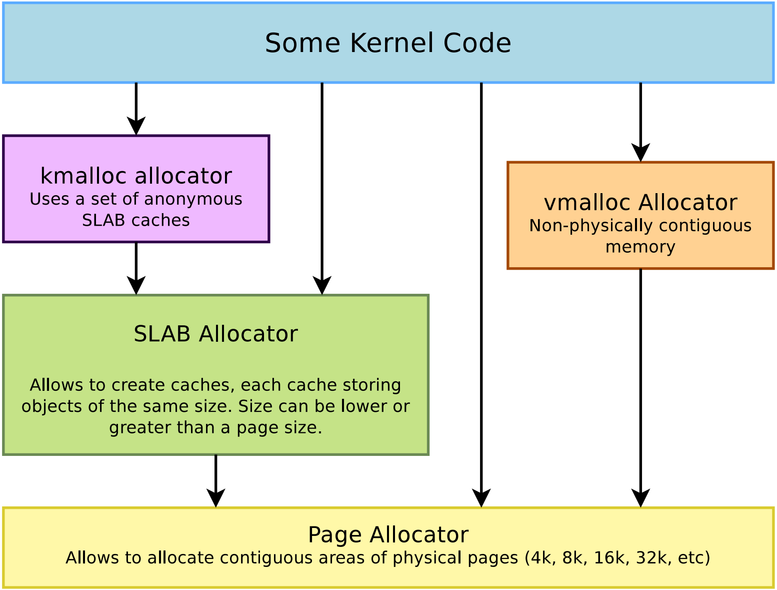
Notes on user-space memory

- 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.

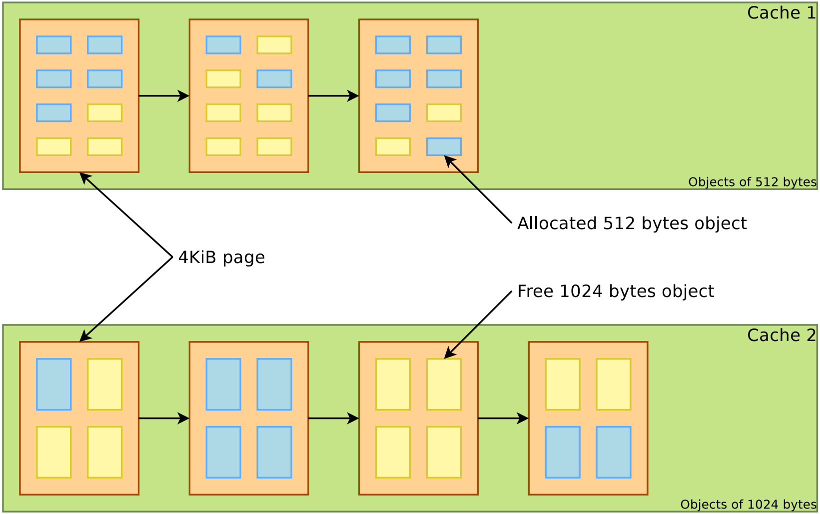
Page Allocator Flags

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

kmalloc API 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, 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

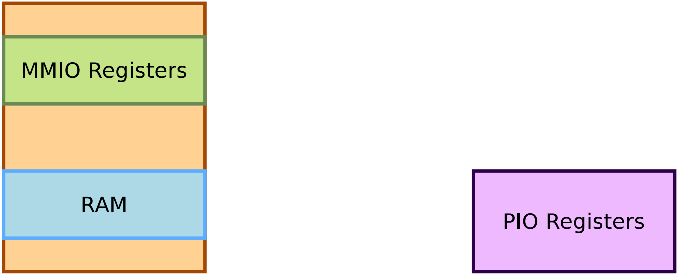
```
/* Define the global list */
static LIST_HEAD(tc_list);

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
...
```

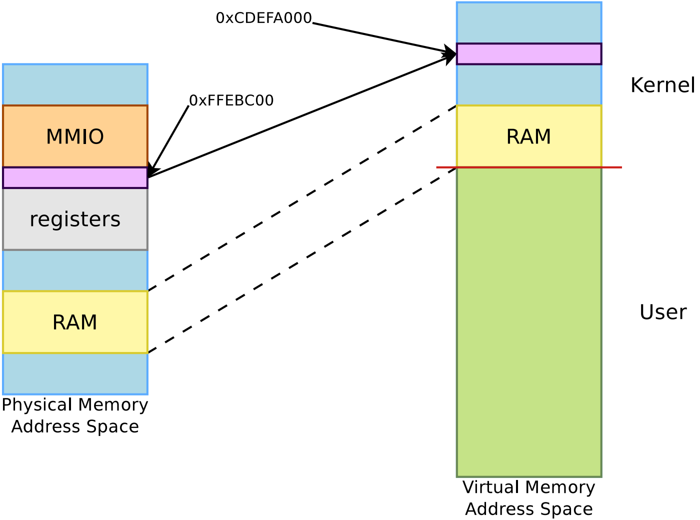
Mapping I/O memory in virtual memory

- 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).

Character drivers

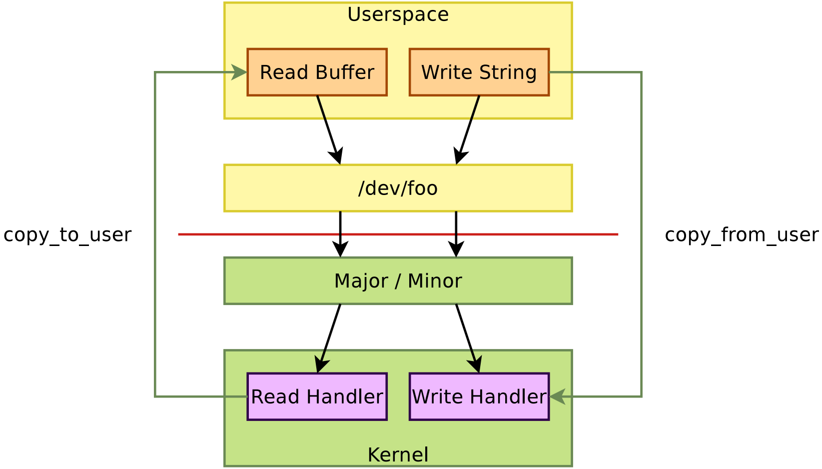
Usefulness of character drivers

- 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.

- 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

- `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.

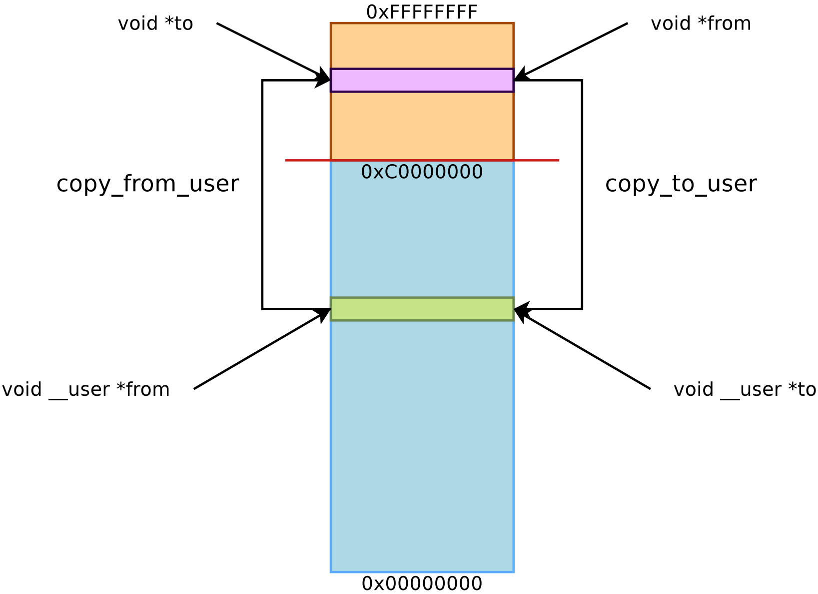
Exchanging data with user-space 1/3

- 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

```c
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;
}
```

```
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

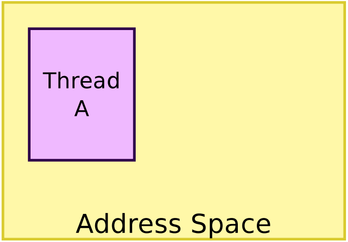
Processes and scheduling

Process, thread?

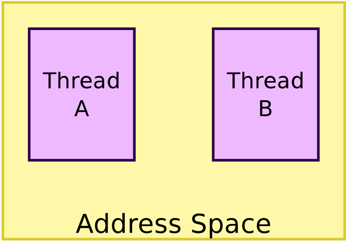
- Confusion about the terms *process*, *thread* and *task*
- In Unix, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - One thread, that starts executing the `main()` function.
  - Upon creation, a process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

Process, thread: kernel point of view

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

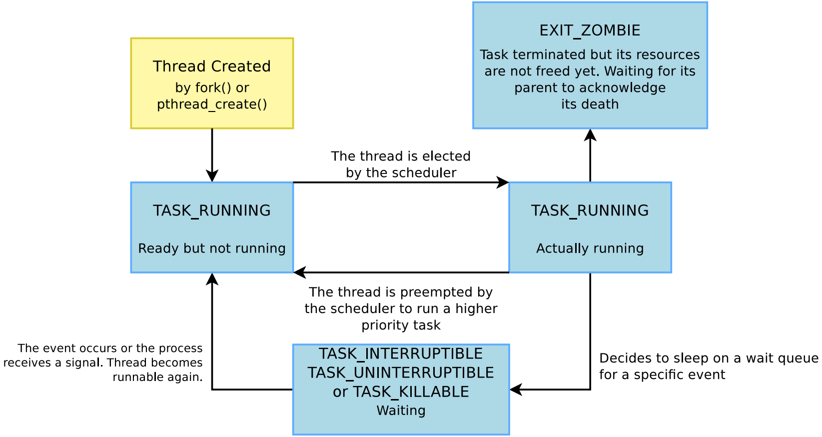


Process after fork()

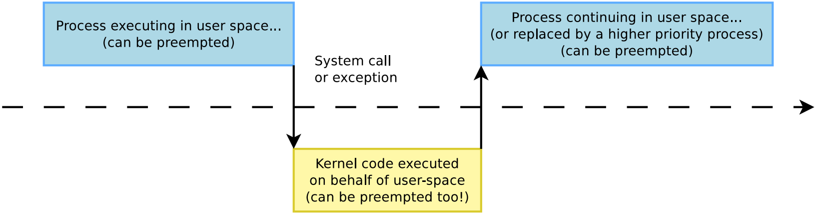


Same process after pthread_create()

A thread life



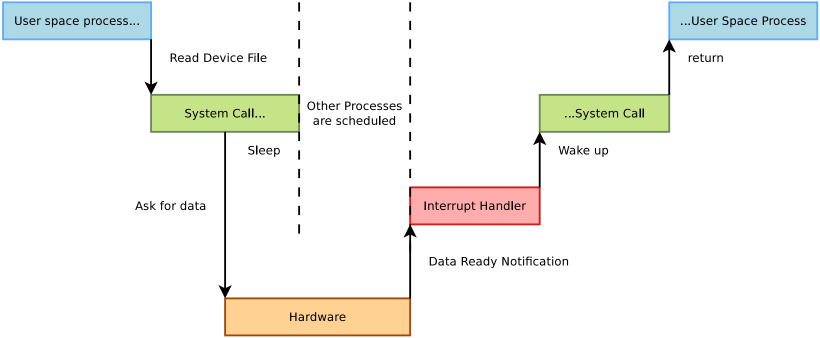
Execution of system calls



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

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

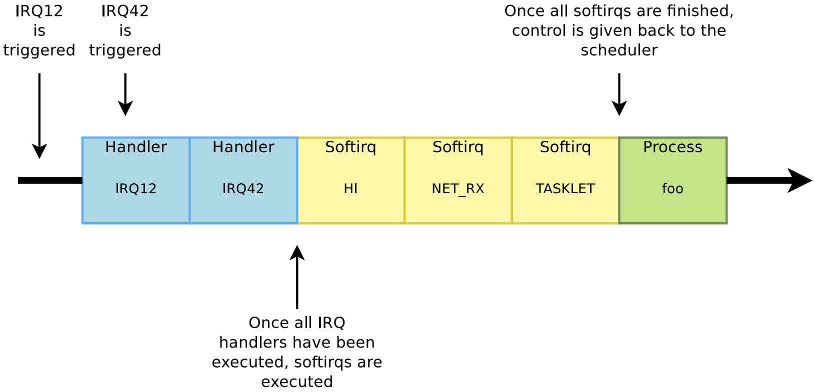
Registering an interrupt handler 1/2

- 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);
        [...]
}
```

```
/* 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

Workqueues

- To create a task statically, you can use:
  `DECLARE_WORK(name, void (*function)(void *), void *data);`
  or dynamically:
  `INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);`
  `PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);`
- You can then submit your work to the shared queue using:
  `int schedule_work(struct work_struct *work);`

Workqueues

- You can also create your own threads:
  `struct workqueue_struct *create_workqueue(const char *name);`
  `struct workqueue_struct *create_singlethread_workqueue(const char *name);`
- Tosubmit your work int those threads, use:
  `int queue_work(struct workqueue_struct *queue, struct work_struct *work);`
  `int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);`
- The complete API, in `include/linux/workqueue.h` provides many other possibilities (creating its own workqueue threads, etc.)

Interrupt management summary

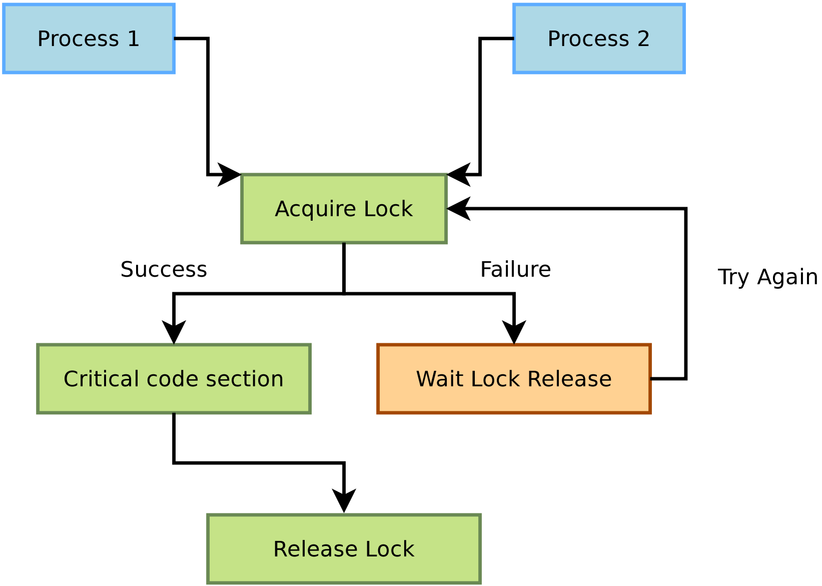
- Device driver
  - When the device file is first opened, register an interrupt handler for the device's interrupt channel.
- Interrupt handler
  - Called when an interrupt is raised.
  - Acknowledge the interrupt
  - If needed, schedule a tasklet taking care of handling data. Otherwise, wake up processes 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.

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);`

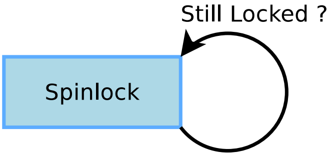
Locking and Unlocking Mutexes 1/2

- `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.

- 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

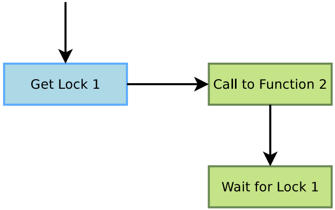
```
struct uart_port {
        spinlock_t lock;
        /* Other fields */
};
```

- Spinlock taken/released with protection against interrupts

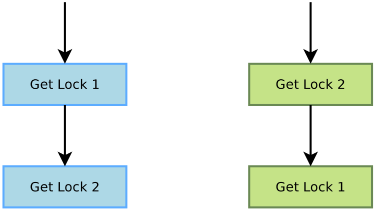
```
static unsigned int ulite_tx_empty
    (struct uart_port *port) {
        unsigned long flags;

        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(...);`

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);`
    - `u` for decimal representation
    - `x` for hexadecimal representation
- 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);`
- 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

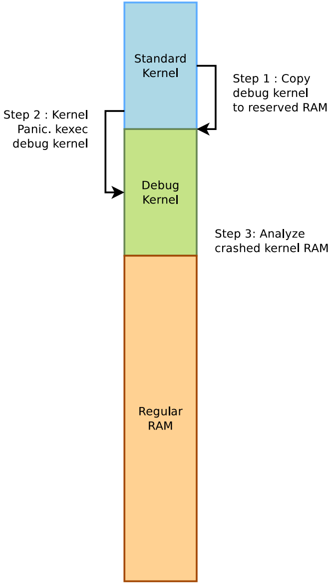
- Two types of JTAG dongles
  - Those offering a `gdb` compatible interface, over a serial port or an Ethernet connexion. `gdb` can directly connect to them.
  - Those not offering a gdb compatible interface are generally supported by OpenOCD (Open On Chip Debugger): `http://openocd.sourceforge.net/`
    - 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

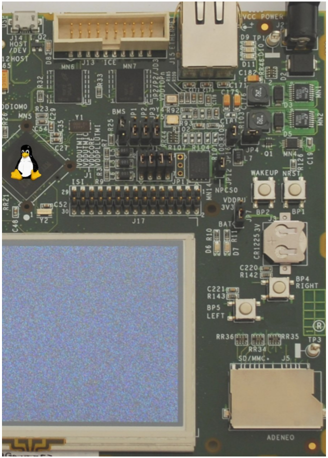
- http://lttng.org
  - The successor of the Linux Trace Toolkit (LTT)
  - Toolkit allowing to collect and analyze tracing information from the kernel, based on kernel markers and kernel tracepoints.
  - So far, based on kernel patches, but doing its best to use in-tree solutions, and to be merged in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

LTTV

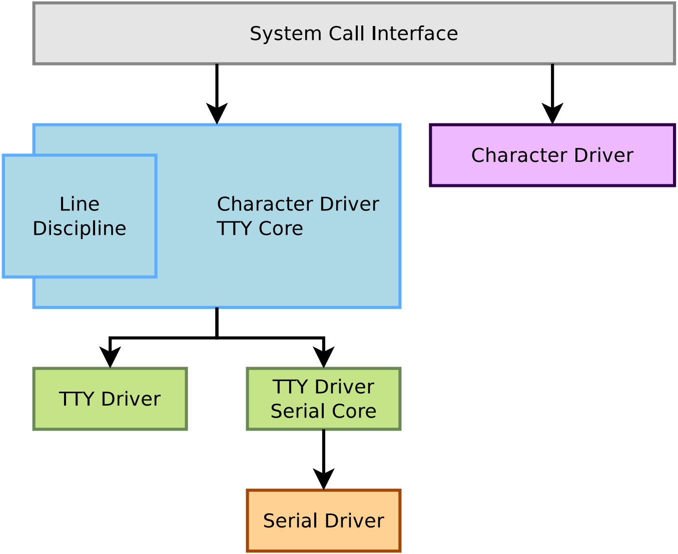
- Viewer for LTTng traces
  - Support for huge traces (tested with 15 GB ones)
  - Can combine multiple tracefiles in a single view.
  - Graphical or text interface
- See `http://lttng.org/files/lttv-doc/user_guide/`

Serial Drivers

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



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 later

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

```c
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

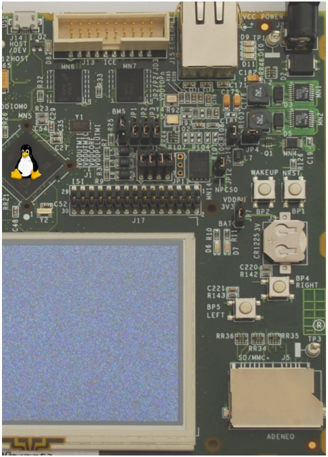
```c
static int __init serial_txx9_console_setup(struct console *co,
    char *options)
{
    struct uart_port *port;
    struct uart_txx9_port *up;
    int baud = 9600;
    int bits = 8;
    int parity = 'n';
    int flow = 'n';

    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);
}
```

Porting the Linux Kernel to an ARM Board

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



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

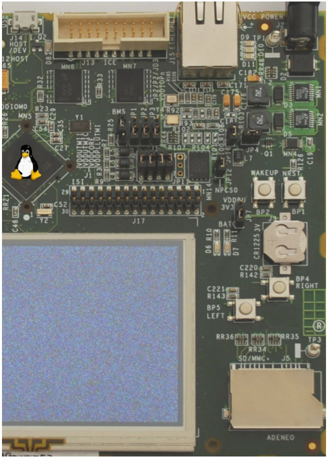
- The *device tree source* (`.dts`) is compiled into a *device tree blob* (`.dtb`) using a *device tree compiler* (`.dtc`)
  - The `dtb` is an efficient binary data structure
  - The `dtb` is either appended to the kernel image, or better, passed by the bootloader to the kernel
- At runtime, the kernel 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, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



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.

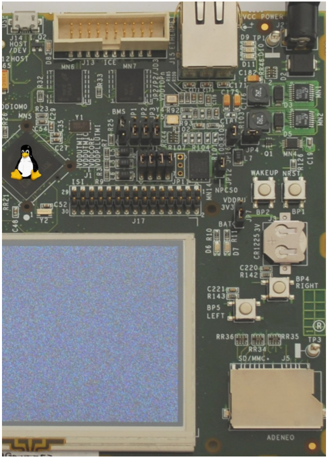
Useful Resources

- 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

Introduction to Git

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier

Free Electrons, Adeneo Embedded



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

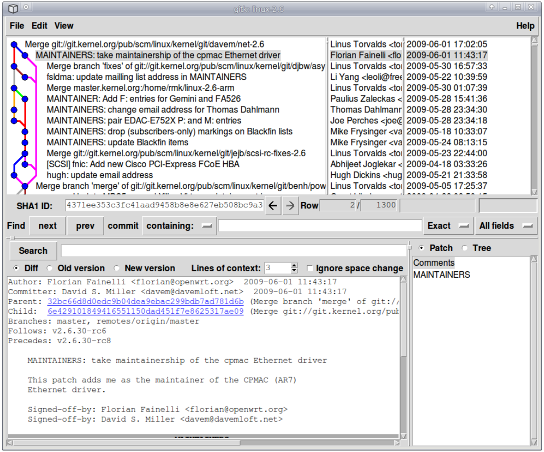
- `git log` will list all the commits. The latest commit is the first.

```
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)

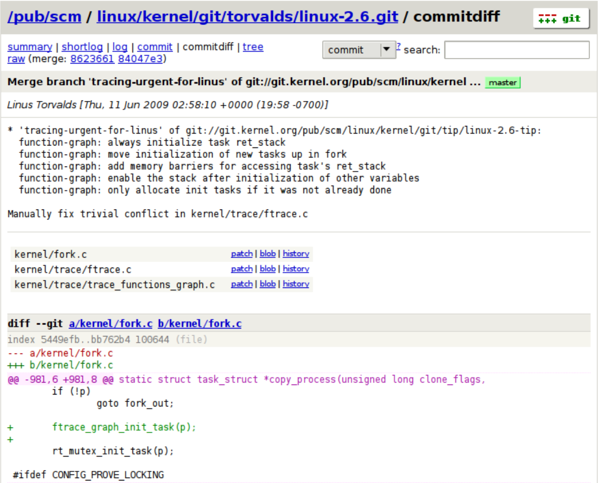
Visualize the History: gitk

- `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)

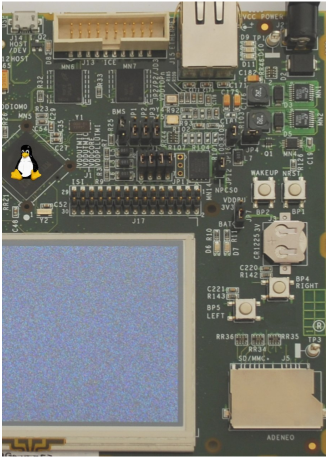
- If you use `git format-patch` to produce your patches, you will need to update your branch and may need to group your changes in a different way (one patch per commit).
- Here's what we recommend
  - Update your master branch
    - `git checkout master; git pull`
  - Back to your branch, implement 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`

Kernel Advice and Resources

Grégory Clément, Michael Opdenacker, Maxime Ripard, Sébastien Jan, Thomas Petazzoni, Alexandre Belloni, Grégory Lemercier
Free Electrons, Adeneo Embedded



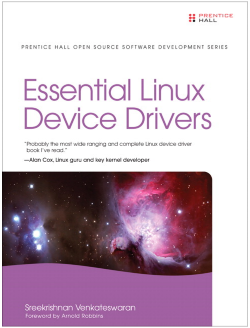
References

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.

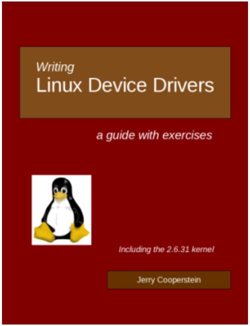
Useful Reading (1)

- 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



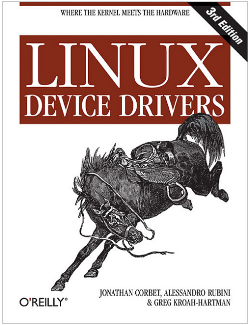
Useful Reading (2)

- 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.



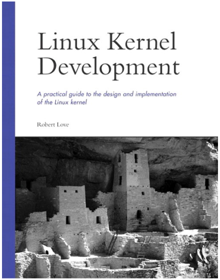
Useful Reading (3)

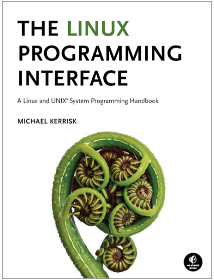
- Linux Device Drivers, 3rd edition, Feb 2005
  - `http://www.oreilly.com/catalog/linuxdrive3/`
  - By Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, O'Reilly
  - Freely available on-line! Great companion to the printed book for easy electronic searches!
  - `http://lwn.net/Kernel/LDD3/` (1 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

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.

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

Thank you!

And may the Source be with you