Embedded Linux driver development

Embedded Linux kernel and driver development

Michael Opdenacker

Free Electrons

http://free-electrons.com/

Thanks

- To Jonathan Corbet, for his very useful news and articles on http://lwn.net/, in particular for porting drivers to 2.6.
- To Greg Kroah-Hartman, for his very useful articles on udev
- To the OpenOffice.org project, for their presentation and word processor tools which satisfied all my needs.
- To the Handhelds.org community, for giving me so much help and so many opportunities to help.
- To the members of the whole Free Software and Open Source community, for sharing the best of themselves: their work, their knowledge, their friendship.
- To people who helped, sent corrections or suggestions:
  Phil Blundell, Jeffery Huang, Mohit Mehta, Matti Aaltonen

Copying this document

Document updates available on http://free-electrons.com/training/drivers
Corrections, suggestions and contributions are welcome!

Document history

Unless specified, contributions are from Michael Opdenacker

See http://free-electrons.com/doc/ChangeLog for detailed changes.

- Mar 9, 2005. Latest update. Corrections and minor improvements
- Mar 4, 2005. Many small updates and improvements. A few new slides.
- Dec 18, 2004. Added 84 more pages: character drivers, memory management, interrupts, DMA, device model, sysfs, hotplug, udev.
- Oct 1, 2004. Added jffs2 image mounting instructions.
- Sep 28, 2004. First public release

About this document

- This document is first of all meant to be used as a visual aid by a speaker or a trainer. Hence, this is just a summary or a complement to what is said. Hence, the explanations are not supposed to be exhaustive.
- However, this document is also meant to become a reference for the audience. It also targets readers interested in self-training. So, a bit more details are given, making the document a bit less visually attractive.

Training contents (1)

Introduction

- System overview and role of the kernel
- History and versioning scheme
- Supported hardware architectures
- Legal issues: licensing constraints, software patents
- Kernel user interface

Training contents (2)

Compiling and booting

- Getting the sources
- Using the patch command
- Structure of source files
- Kernel modules
- Kernel configuration
- Compiling
- Cross-compiling
- The bootloader
- Booting parameters
- Debugging through the serial port
- Creation of an initrd ramdisk

Training contents (3)

Driver development

- Linux device drivers
- A simple module
- Programming constraints
- Loading, unloading modules
- Module parameters
- Module dependencies
- Adding sources to the kernel tree
- Kernel debugging

Training contents (4)

Advanced driver development

- Memory management
- I/O register and memory access
- Character device drivers
- Sleeping, interrupts
- mmap, DMA
- New device model, sysfs
- Hotplug
- udev dynamic devices

Training contents (4)

Advice and resources

- Using Ethernet over USB
- Root filesystem on the host through NFS
- Review of the various filesystem types. The MTD subsystem. Advice for making a choice
- Getting help and contributions
- Bug report and patch submission to Linux developers.
- References

Studied kernel version: 2.6

Linux 2.4

- Mature and quite exhaustive
- But developments stopped; very few developers willing to help.
- Will be definitely obsolete when your new product starts.
- Still fine if you get your sources, tools and support from commercial Linux vendors

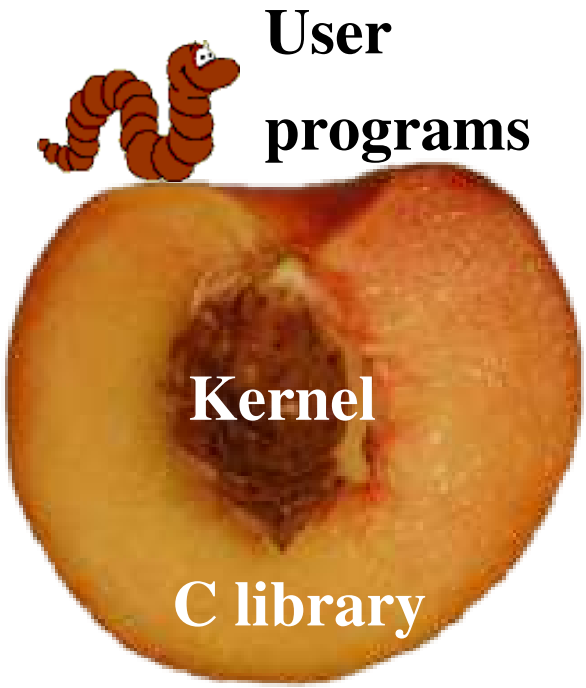
Linux 2.6

- Support from Linux hackers and community
- Now mature and exhaustive. Most drivers upgraded.
- Cutting edge features and increased performance.

Introduction

Role of the kernel

- Linux is the kernel.
  It provides an interface to the hardware
- GNU / Linux is the whole operating system
- Hurd, Mach, BSD are other kernels
- GNU / Hurd, MacOS, FreeBSD are other operating systems



Linux history

- 1991: Linux kernel written from scratch in 6 months by Linus Torvalds in his Helsinki University room, to overcome limitations of his 80386 PC.
- 1991: Linus shares his kernel on the net. Programmers from the whole world join in and contribute to coding and testing
- 1992: Linux released under the GNU General Public License
- 1994: Linux 1.0 released
- 1994: Red Hat founded by Bob Young and Marc Ewing, creating a new business model.
- 1995-: GNU/Linux and free software developing in Internet servers.
- 2001: IBM invests $1 billion in Linux
- 2002-: GNU/Linux wide adoption starts in many industry sectors.

Linux versioning scheme

Releases are versioned as x.y.z

- Stable versions
  - x.y: main release number
  - y: even number
  - z: identifies the exact release version number (use
  - Examples: 2.0.40, 2.2.26, 2.4.27, 2.6.7 ...
- Development versions
  - y: odd number
  - Examples: 2.3.42, 2.5.74 ...

Supported hardware architectures

- See the `arch/` directory
- Minimum: 32 bit processors, with or without MMU
- 32 bit architectures:
  alpha, arm, cris, h8300, i386, m68k, m68knommu, mips, parisc, ppc, s390, sh, sparc, um, v850
- 64 bit architectures:
  ia64, mips64, ppc64, sh64, sparc64, x86_64
- See `arch/README` or `Documentation/arch/README` for details

Linux key features

- Portability and hardware support
- Scalability

  Can run on super computers as well as on tiny devices
- Compliance to standards and interoperability
- Networking
- Security
- Stability and reliability
- Modularity

Introduction

Legal issues

Licensing details and constraints

About Free Software

- Linux is *Free Software*
- *Free Software* grants the below 4 freedoms to the user:
  - The freedom to run the program, for any purpose
  - The freedom to study how the program works, and adapt it to one's needs
  - The freedom to redistribute copies to help others
  - The freedom to improve the program, and release one's improvements to the public
- See http://www.gnu.org/philosophy/free-sw.html

The GNU General Public License (GPL)

- *Copyleft* licenses use copyright laws to make sure that modified versions are free software too
- The GNU GPL requires that modifications and derived works are GPL too:
  - Only applies to **released** software
  - Any program using GPLed code (either by static or even dynamic linking) is considered as an extension of this code

Copyleft and GPL links

More details

- Copyleft: http://www.gnu.org/copyleft/copyleft.html
- GPL FAQ: http://www.gnu.org/licenses/gpl-faq.html

Linux kernel licensing constraints (1)

No constraints before you release.

You should share your changes early for your own interest, but you don't have to!

Linux kernel licensing constraints (2)

Constraints at release time

- For any device embedding Linux and Free Software, you have to release sources to the end user. You have no obligation to release them to anybody else!
- Proprietary modules are tolerated (but not recommended) as long as they cannot be considered as derived work of GPLed code.
- Proprietary drivers can't be statically compiled in the kernel.
- No issue with drivers available under a GPL compatible license (details given in the module programming section)

Advantages of free software drivers

From the driver developer / decision maker point of view

- You don't have to write your driver from scratch. You can reuse code from similar free software drivers.
- You get free community contributions, support, code review and testing. Proprietary drivers (even with sources) don't get any.
- Your drivers can be freely shipped by others (mainly by distributions)
- Your drivers can be statically compiled in the kernel
- 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)
- Modules have all privileges. You need the sources to make sure that a module is not a security risk.

Introduction

Legal issues

Software patents

Software patents: the big legal threat

- Software implementations very well protected internationally by Copyright Law. This is automatic, no paperwork.
- However, in countries like the USA or Japan, it is now legal to patent what the software does, instead of protecting only the implementation.
- Patents can be used to prevent anyone from re-using or even improving an algorithm or an idea!
- Deadly for software competition and innovation: can't write any program without reusing any technique or idea from anyone.

Software patents hall of shame

- The progression bar
- Amazon 1-click, Amazon gift ordering
- Electronic shopping cart
- Compressing and decompressing text files
- Compression in mobile communication
- Digital signature with extra info
- Hypermedia linking

See http://swpat.ffii.org/patents/samples/index.en.html for more examples

How to avoid patent issues

- Applies too when you develop in software patent free areas. You may not be able to export your products.
- Kernel drivers with patents: always check driver description in kernel configuration. Known patent issues are always documented.
- Always prefer patent free alternatives (PNG instead of GIF, Linux RTAI instead of RTLinux, etc.)
- Don't file patents on your technologies at your turn. This may expose you more to patent risk. You will lose against software giants.

How to deal with patent issues

When patent lawyers are after you, you may get help from:

- In the USA
  - The Electronic Frontier Foundation
    http://eff.org/
- In the European Union
  - The Foundation for a Free Information Infrastructure
    http://ffii.org/index.en.html
- In other areas
  - *Note to readers: any references are welcome!*

Introduction

Kernel user interface

Kernel userspace interface

A few examples:

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

... and much more! See by yourself!

Userspace interface documentation

Very detailed pieces of information about the `/proc` interface are available in `Documentation/filesystems/proc.txt` (almost 2000 lines) in the kernel sources.

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

Compiling and booting Linux

Getting the sources

Access to kernel sources

- Download sources from http://kernel.org/:

```
wget http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.7.tar.bz2
wget http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.7.tar.bz2.sign
```

- Or get a patch vs the x.y.<z-1> version:

```
wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/patch-2.6.7.bz2
wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/patch-2.6.7.bz2.sign
```


Checking the integrity of sources

- Check the integrity of sources:

```
gpg --verify linux-2.6.7.tar.bz2.sign linux-2.6.7.tar.bz2
```

  - GnuPG details: http://www.gnupg.org/gph/en/manual.html
  - Kernel source signature details: http://www.kernel.org/signature.html

Using the patch command

- `patch` command: uses the output of the `diff` command to apply a set of changes to a source tree.
- `patch` usage examples
  - `patch -p<n> < diff_file`
  - `cat diff_file | patch -p<n>`
  - `bunzip2 -c diff_file.bz2 | patch -p<n>`
  - `gunzip -c diff_file.gz | patch -p<n>`

  n: number of directory levels to skip (example next page)

patch usage example

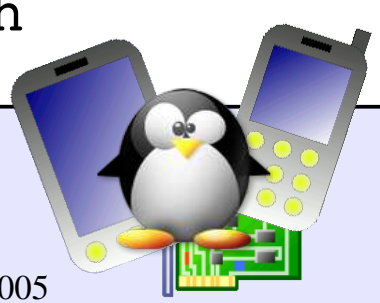
- Patch file (`hardware.diff`)

```
--- linux-2.6.8.1/include/asm-arm/hardware.h    2004-08-14 12:54:48.000000000 +0200
+++ linux-2.6.8.1_modified/include/asm-arm/hardware.h   2004-08-17 12:42:06.119650556
+0200
@@ -15,13 +15,4 @@
#include <asm/arch/hardware.h>
-#ifndef __ASSEMBLY__
-
-struct platform_device;
-
-extern int platform_add_devices(struct platform_device **, int);
-extern int platform_add_device(struct platform_device *);
-
-#endif
-
#endif
```

- Commands:

```
cd linux-2.6.8.1
patch -p1 < hardware.diff
```

- Applies changes to `include/asm-arm/hardware.h`



Applying a Linux patch

Linux patches

- Always to apply to the `x.y.<z-1>` version
- Always produced for `n=1` (that's what everybody does... do it too!)
- Official mainstream patches always `bzip2` compressed

Linux patch command line example

```
wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.10.bz2
wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.10.bz2.sign
gpg --verify patch-2.6.10.bz2.sign
cd linux-2.6.9
bunzip2 -c ../patch-2.6.10.bz2 | patch -p1
```

- Keep patch files compressed: useful to check their signature later

Useful kernel source links

Difficult to find if you don't know them!

- Direct view access to the Linux source repository, useful to create patches against the latest versions:
  http://linux.bkbits.net:8080/linux-2.6/src
- Linux daily source snapshots:
  http://www.kernel.org/pub/linux/kernel/v2.6/snapshots/old/
  http://www.kernel.org/pub/linux/kernel/v2.6/snapshots/

Compiling and booting Linux

Structure of source files

Linux sources structure (1)

`arch/`	Architecture dependent code
`COPYING`	Linux copying conditions (GNU GPL)
`CREDITS`	Linux main contributors
`crypto/`	Cryptographic libraries
`Documentation/`	Kernel documentation. Don't miss it!
`drivers/`	All device drivers (`drivers/usb/`, etc.)
`fs/`	Filesystems (`fs/ext3/`, etc.)
`include/`	Kernel headers
`include/asm-<arch>`	Architecture dependent headers
`include/linux`	Linux kernel core headers
`init/`	Linux initialization (including `main.c`)
`ipc/`	Code used for process communication

Linux sources structure (2)

`kernel/`	Linux kernel core (very small!)
`lib/`	Misc library routines (`zlib`, `crc32`...)
`MAINTAINERS`	Maintainers of each kernel part. Very useful!
`Makefile`	Top Linux makefile (sets arch and version)
`mm/`	Memory management code (small too!)
`net/`	Network support code (not drivers)
`README`	Overview and building instructions
`REPORTING-BUGS`	Bug report instructions
`scripts/`	Scripts for internal or external use
`security/`	Security model implementations (SELinux...)
`sound/`	Sound support code and drivers
`usr/`	Early user-space code (`initramfs`)

Compiling and booting Linux

Kernel modules

Loadable kernel modules (1)

- Modules: add a given functionality to the kernel (drivers, filesystem support, and many others)
- Can be loaded and unloaded at any time, only when their functionality is need. Once loaded, have full access to the whole kernel. No particular protection.
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).

Loadable kernel modules (2)

- Useful to support incompatible drivers (either load one or the other, but not both)
- Useful to deliver binary-only drivers (bad idea) without having to rebuild the kernel.
- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...
- Modules can also be compiled statically into the kernel.

Compiling and booting Linux

Kernel configuration

Kernel configuration overview

- Makefile edition
  Setting the version and target architecture if needed
- Kernel configuration: defining what features to include in the kernel:

```
make [config|xconfig|gconfig|menuconfig|oldconfig]
```

  - Kernel configuration file (Makefile syntax) stored in the `.config` file at the root of the kernel sources
  - Distribution kernel config files usually released in `/boot/`

Makefile changes

- To identify your kernel image with others build from the same sources, use the `EXTRAVERSION` variable:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 7
EXTRAVERSION = -acme1
```

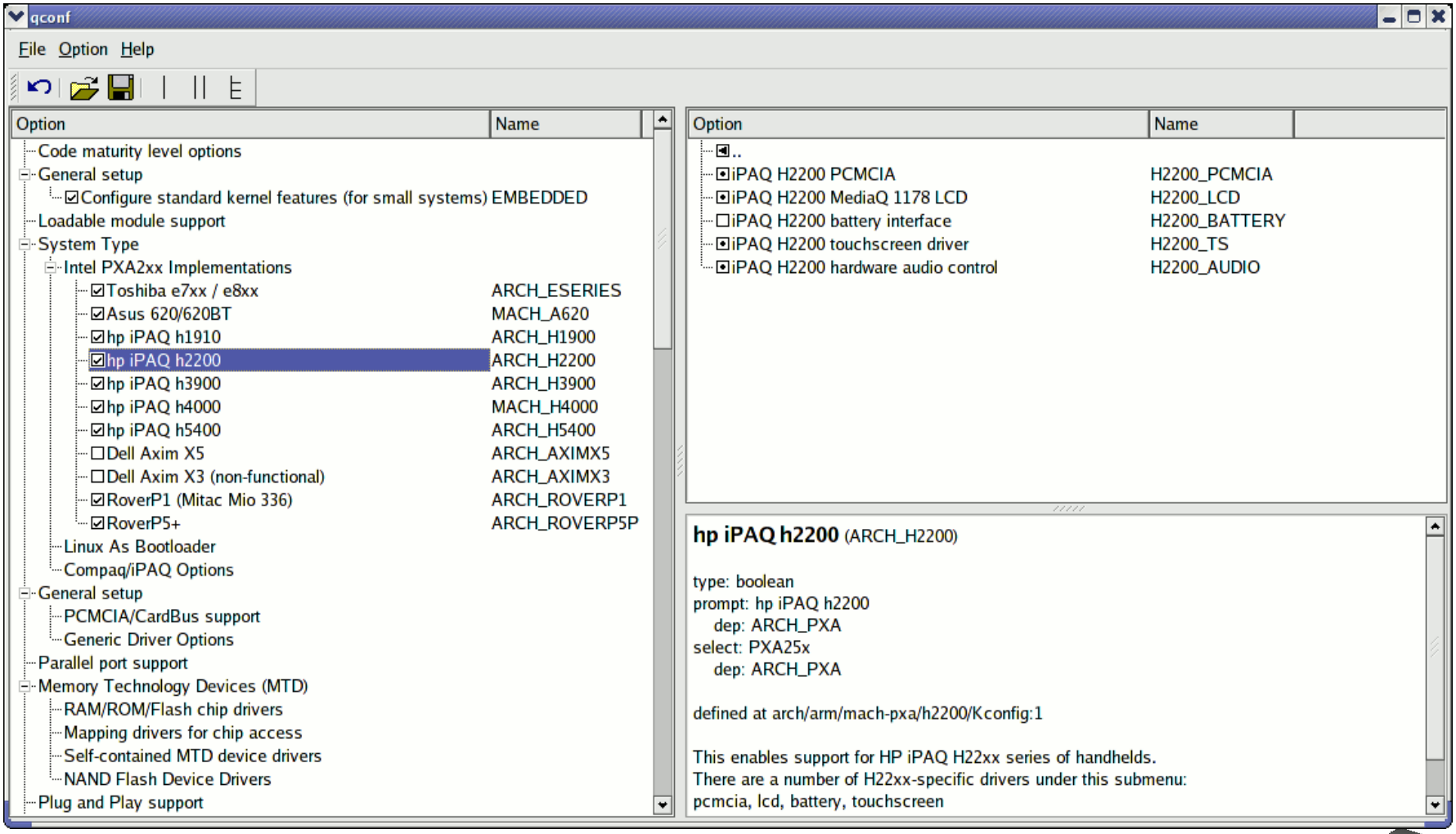
  - `uname -r` will return: `2.6.7-acme1`

make xconfig

`make xconfig`

- `qconf`: new qt configuration interface for Linux 2.6. Much easier to use!
- Make sure you read `help -> introduction: useful options!`
- File browser: easier to load configuration files

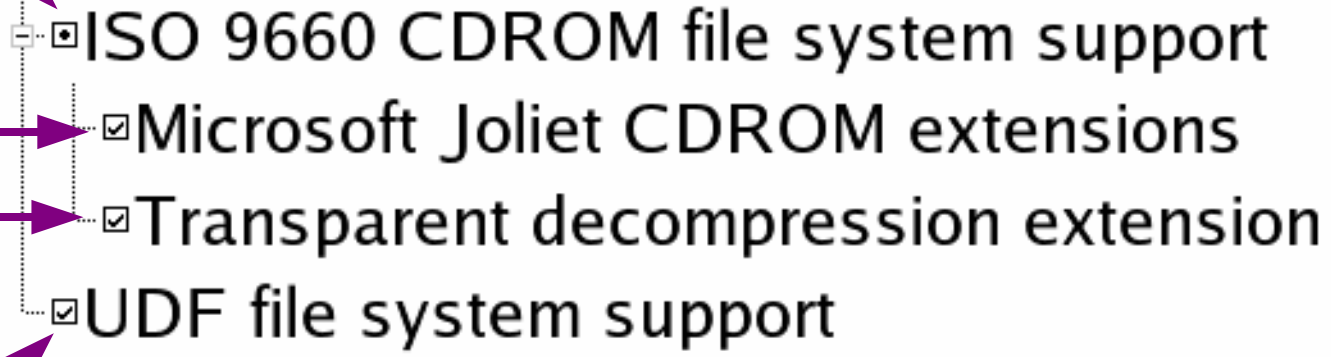
qconf screenshot



Compiling statically or as a module

Compiled as a separate module
`CONFIG_ISO9660_FS=m`

Driver options
`CONFIG_JOLIET=y`
`CONFIG_ZISOFS=y`



Compiled statically in the kernel
`CONFIG_UDF_FS=y`

make config / menuconfig / gconfig

`make config`

- Asks you the questions 1 by 1. Extremely long!

`make menuconfig`

- Same old text interface as in Linux 2.4. Rarely useful.

`make gconfig`

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

make oldconfig

`make oldconfig`

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

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

make help

`make help`

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

Compiling and booting Linux

Compiling the kernel

Compiling and installing the kernel

Compiling steps:

- `make`

Install steps (logged as root!)

- `make install`
- `make modules_install`

The following commands are no longer needed:

- `make depends`
- `make modules` (done by `make`)

Compiling faster

- Spend a bit more time in kernel configuration and just compile the modules needed on your hardware. This can divide compile time by 30 and save hundreds of MB!
- Compile several files in parallel:
  `make -j <number>`
  Runs several targets in parallel, whenever possible
  - `make -j 4`
    Much faster even on uniprocessor workstations! Less time wasted in reading or writing files (the other jobs keep the CPU busy)
  - Not really useful going further than 4 (too much context switching)
  - `make -j <4*number_of_processors>`
    On a multiprocessor machine. Beware of not disturbing other users, if any!

Kernel compiling tips

- View the full (gcc, ld) command line:

```
make V=1
```

- Remove all generated files (to create patches...):

```
make mrproper
```

Generated files

- `vmlinux`

  Raw Linux kernel image, non compressed

- `arch/<arch>/boot/zImage`

  zlib compressed kernel image

  Default image on arm

- `arch/<arch>/boot/bzImage`

  bzip2 compressed kernel image. Usually small enough to fit on a floppy disk!

  Default image on i386

Installed files (1)

- `/boot/vmlinuz-<version>`
  Kernel image
- `/boot/System.map-<version>`
  Stores kernel symbol addresses
- `/boot/initrd-<version>.img`
  Initial RAM disk, storing the modules you need to mount your root filesystem. `make install` runs `mkinitrd` for you!
- `/etc/grub.conf` or `/etc/lilo.conf`
  make install updates your bootloader configuration files to support your new kernel! It reruns `/sbin/lilo` if LILO is your bootloader.

Installed files (2)

- `/lib/modules/<version>/`

  Kernel modules + extras

  - `build/`

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

  - `kernel/`

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

Installed files (3)

- `/lib/modules/<version>/` (continued)
  - `modules.alias`

    Module aliases for insmod and modprobe. Example line:
    `alias sound-service-?-0 snd_mixer_oss`
  - `modules.dep`

    Module dependencies for insmod and modprobe. Also useful to copy only the required modules to a minimum filesystem.
  - `modules.symbols`

    Tells which module a given symbol belongs to.

  All the files in this directory are text files. Don't hesitate to have a look by yourself!

Compiling the kernel in a nutshell

- Edit version information in the `Makefile` file
- `make xconfig`
- `make`
- `make install`
- `make modules_install`

Compiling and booting Linux

Cross-compiling the kernel

Makefile changes

Makefile changes

- Update the version as usual
- You should change the default target platform, e.g.:

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

  (prefix of the cross-compiler executable)

- or run (ARM example):
  `make ARCH=arm CROSS_COMPILE=arm-linux-`
  (Useful when you compile for several platforms)

See comments in `Makefile` for details

Configuring the kernel

- Same as native compilation
- Don't forget to set the right architecture
- Useful way of sharing your configuration file:
  `cp .config arch/<arch>/configs/acme_defconfig`

  To get your standard configuration file, the other people working on the ACME embedded system and using your kernel will just have to run:
  `make acme_defconfig`

Cross-compiling setup

Example

- You have an ARM cross-compiling toolchain in `/usr/local/arm/3.3.2/`
- You just have to add it to your Unix PATH:
  `export PATH=/usr/local/arm/3.3.2/bin:$PATH`

See the `Documentation/Changes` file in the sources for details about minimum tool versions requirements .

Building the kernel

- Run

  `make` (if you have modified your Makefile)

  or otherwise (ARM example)

  ```
  make ARCH=arm CROSS_COMPILE=arm-linux-
  ```

- Copy

  `arch/<platform>/boot/zImage`

  to the target storage

- `make modules_install`

  and copy `/lib/modules/<version>` to the target storage

- You can customize `arch/<arch>/boot/install.sh` so that `make install` does this automatically for you.

Compiling and booting Linux

The bootloader

The bootloader's job

One main mission: load the operating system(s). Tasks:

- Initialize the machine properly (the kernel can do part of this later too).
- Access the kernel and initrd files in their storage medium (need to support the corresponding filesystem too)
- Because of the above 2 tasks, bootloaders are often platform specific!
- Load the kernel and initrd files
- Execute the kernel file with the right command line

2-stage bootloaders

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

A few bootloaders (1)

- LILO: LInux LOad. Original Linux bootloader. Still in use!
  http://freshmeat.net/projects/lilo/
  Supports: x86
- GRUB: GRand Unified Bootloader from GNU. More powerful.
  http://www.gnu.org/software/grub/
  Supports: x86
- LinuxBIOS: Linux based BIOS replacement
  http://www.linuxbios.org/
  Supports: x86

A few bootloaders (2)

- sh-boot: LinuxSH project bootloader
  http://cvs.sourceforge.net/viewcvs.py/linuxsh/sh-boot/
  Supports: sh
- bootldr: Handhelds.org's bootloader for iPAQs
  ftp://ftp.handhelds.org/bootldr/
  Supports: arm
- LAB: Linux As Bootloader, from Handhelds.org
  Part of Handhelds.org's Linux kernel.
  See http://handhelds.org/moin/moin.cgi/Linux26ToolsAndSources
  Supports: arm (experimental)

A few bootloaders (3)

- U-Boot: Universal Bootloader. The most used on arm.
  http://u-boot.sourceforge.net/
  Supports: arm, ppc, mips, x86
- RedBoot: eCos based bootloader from Red-Hat
  http://sources.redhat.com/redboot/
  Supports: x86, arm, ppc, mips, sh, m68k...

Kernel command line parameters

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

- Useful to configure the kernel at boot time, without having to recompile it.
- Example (used for the HP iPAQ h2200 PDA)

```
root=/dev/ram0 rw init=/linuxrc \
console=ttyS0,115200n8 console=tty0 \
ramdisk_size=8192 cachepolicy=writethrough \
```

Most common command line parameters

- `root`

  Identifies the root filesystem

- `init`

  Script to run at the end of kernel initialization

  Default: `/sbin/init`

- `console`

  Console for booting messages

- `ro` / `rw`

  Mount root device as read-only / read-write

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

Compiling and booting Linux

Debugging through the serial port

Usefulness of a serial port

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

When you don't have a serial port

On the host

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

On the target

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

Compiling and booting Linux

Creation of an initrd ramdisk

Initrd

Initrd = Initial RAM disk

- Very first, minimalistic root (/) filesystem in RAM
- Traditionally used to minimize the number of device drivers built into the kernel.
  Example: contains an ext3 module to mount the final ext3 root filesystem.
- Also useful to run complex initialization scripts
- Useful to load proprietary modules (can't be statically compiled into the kernel)

How to create an initrd

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

<Populate: busybox, modules, linuxrc script
More details in the Tools for Embedded Linux Systems training!>

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

More about initrds

- Just read `Documentation/initrd.txt` in the kernel sources!
- A very nice piece of documentation indeed.
  Also covers pivot rooting.

Driver development

Linux device drivers

Character drivers

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

```
crw-rw----   1 root uucp 4, 64 Feb 23  2004 /dev/ttyS0
crw--w----   1 jdoe tty 136, 1 Sep 13 06:51 /dev/pts/1
crw-------   1 root root 13, 32 Feb 23  2004 /dev/input/mouse0
crw-rw-rw-   1 root root 1, 3 Feb 23  2004 /dev/null
```

- Examples: keyboards, mice, parallel port, IrDA, Bluetooth port, consoles, terminals...

Block drivers

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

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

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

Device major and minor numbers

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

- First number: major number
  Uniquely associated to each driver
- Second number: minor number
  Uniquely associated to each device

To find out which driver a device corresponds to, or when the device name is too cryptic, see `Documentation/devices.txt`

Device file creation

- Device files are not created when a driver is loaded.
- They have to be created in advance:
  `mknod /dev/<device> [c|b] <major> <minor>`
- Examples:
  `mknod /dev/ttyS0 c 4 64`
  `mknod /dev/hda1 b 3 1`

Other driver types

They don't have any corresponding `/dev` entry you could read or write through a regular Unix command.

- Network drivers

  They are represented by a network device such as `ppp0`, `eth1`, `usbnet`, `irda0` (listed by `ifconfig -a`)

- Other drivers

  Often, intermediate drivers just interfacing with other ones.

Driver development

A simple module

hello module

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

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

Thanks to Jonathan Corbet for the example!

Module license

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

- `GPL`
  GNU Public License v2 or later
- `GPL v2`
  GNU Public License v2
- `GPL and additional rights`
- `Dual BSD/GPL`
  GNU Public License v2 or BSD license choice
- `Dual MPL/GPL`
  GNU Public License v2 or Mozilla license choice
- `Proprietary`
  Non free products

Module license usefulness

- Used by kernel developers to identify issues coming from proprietary drivers, which they can't do anything about.
- Useful for users to check that their system is 100% free
- Useful for GNU/Linux distributors for their release policy checks.

Module coding guidelines (1)

- C includes: you can't use standard C library functions (`printf()`, `strcat()`, etc.). The C library is implemented on top of the kernel, not the opposite.
- Linux provides some C functions for your convenience, like `printk()`, which interface is pretty similar to `printf()`.

So, only kernel header includes are allowed.

Module coding guidelines (2)

- Never use floating point numbers in kernel code. Your code may be run on a processor without a floating point unit (like on ARM). Floating point can be emulated by the kernel, but this is very slow.
- Define all symbols as static, except exported ones (avoid namespace pollution)
- See `Documentation/CodingStyle` for more guidelines
- It's also good to follow or at least read GNU coding standards: http://www.gnu.org/prep/standards.html

Compiling a module

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

```
# Makefile for the hello module

obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

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

Tab!
(no spaces)

Using the module

- Logged as root, run
  `tail -f /var/log/messages`
- Logged as root in another terminal, load the module:
  `insmod ./hello.ko`
- You will see the following in `/var/log/messages`:
  `Sep 13 22:02:30 localhost kernel: Hello, world`
- Now remove the module:
  `rmmod hello`
- You will see:
  `Sep 13 22:02:37 localhost kernel: Goodbye, cruel world`

Module utilities (1)

- `modinfo <module_name>`
  `modinfo <module_path>.ko`

  Gets information about a module: parameters, license, description.

  Very useful before deciding to load a module or not.

- `insmod <module_name>`
  `insmod <module_path>.ko`

  Tries to load the given module, if needed by searching for its `.ko` file throughout the default locations (can be redefined by the `MODPATH` environment variable.

Module utilities (2)

- `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.

- `rmmod <module_name>`

  Tries to remove the given module

- `lsmod`

  Displays the list of loaded modules

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

Driver development

Defining and passing module parameters

hello module with parameters

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

MODULE_LICENSE("GPL");

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

static char *whom = "world";
module_param(whom, charp, 0);
static int howmany = 1;

module_param(howmany, int, 0);

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

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

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

Thanks to Jonathan Corbet for the example!

Using the hello_param module

- Load the module. For example:
```
insmod ./hello_param.ko howmany=2 whom=universe
```
- You will see the following in `/var/log/messages`:
```
Sep 13 23:04:30 localhost kernel: (0) Hello, universe
Sep 13 23:04:30 localhost kernel: (1) Hello, universe
```
- Now remove the module:
```
rmmod hello_param
```
- You will see:
```
Sep 13 23:04:38 localhost kernel: Goodbye, cruel
universe
```


Declaring module parameters (1)

- `module_param(name, type, perm);`

  `name`: regular name symbol

  `type`: either `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool` or `invbool` (checked at compile time!)

  `perm`: permissions for the corresponding entry in `/sys/module/<module_name>/<param>`. Safe to use `0`.

- `module_param_named(name, value, type, perm);`

  To make the `name` variable available outside the module and the `value` variable inside.

Declaring module parameters (2)

- `module_param_string(name, string, len, perm);`
  To defined name as `charp`, string prefilled with string of length `len`, usually `sizeof(string)`
- `module_param_array(name, type, num, perm);`
  To declare an array of parameters

Passing module parameters

- Through `insmod` or `modprobe`:

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

- Through `modprobe`
  after changing the `/etc/modprobe.conf` file:

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

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

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

Driver development

Module dependencies

Module dependencies

- Module dependencies don't have to be described by the module writer.
- They are automatically computed during kernel building from module exported symbols. `module2` depends on `module1` if `module2` uses a symbol exported by `module1`.
- Module dependencies stored in `/lib/modules/<version>/modules.dep`
- You can update this file by running (as root) `depmod -a [<version>]`

Driver development

Adding sources to the kernel tree

New directory in kernel sources (1)

To add an `acme_drivers/` directory to the kernel sources:

- Move the `acme_drivers/` directory to the appropriate location in kernel sources
- Create an `acme_driver/Kconfig` directory
- Create an `acme_driver/Makefile` file based on the Kconfig variables
- In the parent directory Kconfig file, add `source "acme_driver/Kconfig"`

New directory in kernel sources (2)

- In the parent directory Makefile file, add

  `"obj-$(CONFIG_ACME) += acme_driver/"` (just 1 condition)

  or

  `"obj-y += acme_driver/"` (several conditions)

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

Driver development

Kernel debugging

Debugging with printk

- Universal debugging technique used since the beginning of programming (first found in cavemen drawings)
- Printed or not in console or `/var/log/messages` according to the priority (give details and kernel config switches)
- Available priorities (`include/linux/kernel.h`):

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

ksymoops

- Can help decrypting oops messages, by converting addresses and code to useful text
- Easy to use: just copy/paste the oops text to a file
- Command line example:

```
ksymoops --no-ksyms -m System.map -v vmlinux
oops.txt
```

- See `Documentation/oops-tracing.txt` and then `man ksymoops` for details.

Debugging with Kprobes

http://www-124.ibm.com/developerworks/oss/linux/projects/kprobes/

- Fairly simple way of inserting breakpoints in kernel routines
- Unlike printk debugging, you neither have to recompile nor reboot your kernel. You only need to compile and load a dedicated module to declare the address of the routine you want to probe.
- Non disruptive, based on the kernel interrupt handler
- Kprobes even lets you modify register and global kernel internals.
- Supported architectures: only i386 so far.

See http://www-106.ibm.com/developerworks/library/l-kprobes.html for a nice overview

Kernel debugging tips

- If your kernel doesn't boot yet, useful to activate Low Level debugging (Kernel Hacking section):

```
CONFIG_DEBUG_LL=y
```


Advanced driver development

Character devices

Linux error codes

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

- Generic error codes:
  `include/asm-generic/errno-base.h`
- Platform specific error codes:
  `include/asm/errno.h`

Device registration

- First need to create the corresponding device(s) in `/dev`
- Driver initialization: must register a device with a major number
- Registration (`linux/fs.h`)

```
int register_chrdev(
                unsigned it major,
                const char *name,
                struct file_operations *fops);
```

- Freeing the device

```
int unregister_chrdev(
                 unsigned it major,
                const char *name);
```

- If calling these functions fail, they return a strictly negative value.

Registered devices

- Registered devices are visible in `/proc/devices` with the given major number and name:

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

```
Block devices:
1 ramdisk
3 ide0
8 sd
9 md
22 ide1
65 sd
66 sd
67 sd
68 sd
69 sd
...
```

Finding a free major number

- Less and less majors number available
- Unsafe to hardcode one which may conflict with another driver (standard or custom)
- Solution: let `register_chrdev` dynamically find a free one for you!

```
major = register_chrdev (0, "foo", &name_fops);
```

- Issue: you can't create `/dev` entries in advance!

  However, the script loading the module can then use `/proc/devices`:

```
module=foo; device=foo
insmod $module.ko
major=`awk "\\$2==\"$module\" {print \\$1}" /
proc/devices`
mknod /dev/foo0 c $major 0
```


file operations (1)

As you've just seen, when you call `register_chrdev()`, you have to declare file_operations (called *fops*). Here are the main ones:

- `loff_t (*llseek) (struct file *, loff_t, int);`
- `ssize_t (*read) (struct file *, char *, size_t, loff_t *);`
- `ssize_t (*write) (struct file *, const char *, size_t, loff_t *);`
- `int (*open) (struct inode *, struct file *);`
- `int (*release) (struct inode *, struct file *);`

file operations (2)

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

  Can be used to send specific commands to the device, which are neither reading or writing (e.g. formatting a disk, configuration changes).

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

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

- `struct module *owner;`

  Used by the kernel to keep track of who's using this structure and count the number of users of the module.

The `file` structure

Is created by the network during the `open()` call. Represents open files. Pointers to this structure are usually called "*fips*".

- `mode_t f_mode;`
  The file opening mode (FMODE_READ and/or FMODE_WRITE)
- `loff_t f_pos;`
  Current offset in the file.
- `struct file_operations *f_op;`
  Can still be changed!
- `struct dentry *f_dentry`
  Useful to get access to the inode: `filp->f_dentry->d_inode`.

Character driver summary

- Define your file operations (`fops`)
- Define your `module_init` function and call `register_chrdev()`
  - Give a major number, or 0 (automatic)
  - Give your fops
- Define the `module_exit` function, and call `unregister_chrdev()`
- Load your module
- Find the major number (if needed) and create the `/dev/` entry.
- Use the device at last!!!

Advanced driver development

Memory management

kmalloc and kfree

- Basic allocators, kernel equivalents of glibc's `malloc` and `free`.
- `static inline void *kmalloc(size_t size, int flags);`

  `size`: number of bytes to allocate

  `flags`: priority (see next page)
- `void kfree (const void *objp);`
- Example:

```
data = kmalloc(sizeof(*data), GFP_KERNEL);
```


kmalloc features

- Quick (unless it's blocked waiting for pages)
- Doesn't initialize the allocated area
- The allocated area is contiguous in physical RAM
- Allocates by 2^n-k sizes (k: a few management bits)
  Don't ask for 1024 when you need 1000! You'd get 2048!

kmalloc flags (1)

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

- `GFP_KERNEL`
  Standard kernel memory allocation. May block. Fine for most needs.
- `GFP_ATOMIC`
  Allocated RAM from interrupt handlers or code not triggered by user processes. Never blocks.
- `GFP_USER`
  Allocates memory for user processes. May block. Lowest priority.
- `GFP_NOIO`
  May block, but no I/O will be performed.
- `GFP_NOFS`
  May block, but no filesystem operations will be performed.
- `GFS_HIGHUSER`
  Allocating user-space pages where high memory may be used. May block. Low priority.

kmalloc flags (2)

Extra flags (can be added with |)

- `__GFP_DMA`
  Allocate in DMA zone
- `__GFP_HIGHMEM`
  Allocate in high memory (x86 and sparc only)
- `__GFP_REPEAT`
  Ask to try harder. May still block, but less likely.
- `__GFP_NOFAIL`
  Must not fail. Never gives up. Caution: use only when mandatory!
- `__GFP_NORETRY`
  If allocation fails, doesn't try to get free pages.

Allocating by pages

More appropriate than `kmalloc` when you need big slices of RAM:

- `unsigned long get_zeroed_page(int flags);`
  Returns a pointer to a free page and fills it up with zeros
- `unsigned long __get_free_page(int flags);`
  Same, but doesn't initialize the contents
- `unsigned long __get_free_pages(int flags, unsigned long order);`
  Returns a pointer on a memory zone of several contiguous pages in physical RAM
  `order`: $\log_2$(number_of_pages)
- `unsigned long __get_dma_pages(int flags, unsigned long order);`
  Same, but with the `__GFP_DMA` added flag.

Freeing pages

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

Need to use the same order as in allocation.

Mapping physical addresses

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

- `void *vmalloc(unsigned long size);`
- `void vfree(void *addr);`
- `void *ioremap(unsigned long phys_addr, unsigned long size);`

  Doesn't allocate. Maps the given segment in physical RAM to virtual address space.
- `void iounmap(void *address);`

Memory utilities

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

  Fills a region of memory with the given value.

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

  Copies one area of memory to another.

  Use `memmove` with overlapping areas.

- Lots of functions equivalent to glibc ones defined in `include/linux/string.h`


  ```

Advanced driver development

I/O memory and ports

Requesting I/O ports

/proc/ioports example

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

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

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

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

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

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

Reading / writing on I/O ports

The implementation of the below functions and the *unsigned* type can vary from platform to platform!

bytes

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

words

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

"long" integers

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

Reading / writing string on I/O ports

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

byte strings

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

word strings

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

long strings

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


Requesting I/O memory

/proc/iomem example

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

- Equivalent functions with the same interface
- ```
  struct resource * request_mem_region(
      unsigned long start,
      unsigned long len,
      char *name);
  ```
- ```
  void release_mem_region(
      unsigned long start,
      unsigned long len);
  ```

Choosing I/O ranges

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

Differences with standard memory

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

Avoiding I/O access issues

- Caching on I/O ports or memory already disabled, either by the hardware or by Linux init code.
- Memory barriers are supplied to avoid reordering by the compiler:

Architecture dependent

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

Hardware independent

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

Directly mapped memory

- In some architectures (mainly MIPS), I/O memory can be directly mapped in physical address space.
- In this case, I/O pointers shouldn't be dereferenced.
- To avoid portability issues across architectures, the below functions should be used:

```
unsigned read[b|w|l](address);
void writeb[b|w|l](unsigned value, address);

void memset_io(address, value, count);
void memcpy_fromio(dest, source, num);
void memcpy_toio(dest, source, num);
```

Mapping I/O memory in virtual memory

- To access I/O memory, drivers need to have a virtual address that the processor can handle.
- The `ioremap` functions satisfy this need:

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

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

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

/dev/mem

- Used to provide access user-space applications with direct access to physical addresses.
- Actually only works with addresses that are non-RAM (I/O memory) or with addresses that have some special flag set in the kernel's data structures. Fortunately, doesn't provide access to any address in physical RAM!
- Used by applications such as the X server to write directly to device memory.

Advanced driver development

Sleeping

How to sleep

Sleeping is needed when a user process is waiting for data which are not ready yet.

The process is then put in a waiting queue.

Declaring the queue

- `DECLARE_WAIT_QUEUE_HEAD (module_queue);`

Several ways to make a process sleep

- `sleep_on`
  Can't be interrupted!
- `interruptible_sleep_on`
  Can be interrupted by a signal
- `sleep_on_timeout`
  `interruptible_sleep_on_timeout`
  Same as above, with a timeout.
- `wait_event`
  `wait_event_interruptible`
  Sleep until a condition
  is true.

Use only the interruptible commands!

The other ones are rarely needed.

Waking up!

- `wake_up(&queue);`
  Wakes up all the waiting processes on the given queue
- `wake_up_interruptible(&queue);`
  Only wakes up the interruptible processes
- `wake_up_sync(&queue);`
  Doesn't reschedule when you know that another process is about to sleep, in which case rescheduling will happen anyway.

Advanced driver development

Interrupt management

Need for interrupts

- Internal processor interrupts used by the processor, for example for multi-task scheduling.
- External interrupts needed because most internal and external devices are slower than the processor. Better not keep the processor waiting for input data to be ready or data to be output. When the device is ready again, it sends an interrupt to get the processor attention again.

Registering an interrupt handler (1)

Defined in `include/linux/interrupt.h`

- ```
  int request_irq(
      unsigned int irq, /*requested irq channel */
      irqreturn_t (*handler) (int, void *,
          struct pt_regs *), /*interrupt handler */
      unsigned long irq_flags, /* option mask */
      const char * devname, /* registered name */
      void *dev_id) /* used by shared interrupt channels */
  ```
- ```
  void free_irq(
      unsigned int irq,
      void *dev_id);
  ```

Registering an interrupt handler (2)

`irq_flags` bit values (can be combined!)

- `SA_INTERRUPT`
  "Quick" interrupt handler. Run with disabled interrupts.
  Shouldn't need to be used except in specific cases (such as timer interrupts)
- `SA_SHIRQ`
  The interrupt channel can be shared by several devices
- `SA_SAMPLE_RANDOM`
  Interrupts can be used to contribute to the system entropy pool used by `/dev/random` and `/dev/urandom`. Useful to generate good random numbers. Don't use this if the interrupt behavior of your device is predictable!

When to register the handler

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

Information on installed handlers

/proc/interrupts

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

Total number of interrupts

```
cat /proc/stat | grep intr

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

Total number of interrupts

IRQ1 total

IRQ2 total

IRQ3 ...

Interrupt channel detection (1)

- Some devices announce their IRQ channel in a register
- Some devices have always the same behavior: you end up knowing their IRQ channel
- Manual detection
  - Register your interrupt handler for all possible channels
  - Ask for an interrupt
  - Let the called interrupt handler store the IRQ number in a global variable.
  - Try again if no interrupt was received
  - Unregister unused interrupt handlers.

Interrupt channel detection (2)

Kernel detection utilities

- `mask = probe_irq_on();`
- Activate interrupts on the device
- Deactivate interrupts on the device
- `irq = probe_irq_off(mask);`
  - `>` `0`: unique IRQ number found
  - `=` `0`: no interrupt. Try again!
  - `<` `0`: several interrupts happened. Try again!

The interrupt handler's job

- Acknowledge the interrupt to the device
  (otherwise no more interrupts will be generated)
- Read/write data from/to the device
- Wake up any waiting process waiting for the completion of this read/write operation:

```
wake_up_interruptible(&module_queue);
```

Interrupt handler constraints

- Not run from a user context:
  Can't transfer data to and from user space
- Can't run actions that may sleep:
  Need to allocate memory with `GFP_ATOMIC`
- Can't call `schedule()`
- Have to complete their job quickly enough:
  they shouldn't block interrupts for too long.

Interrupt handler prototype

```
irqreturn_t (*handler) (
  int,                          /* irq number */
  void *dev_id,                 /* Pointer used to keep track of the
                                   corresponding device. Useful
                                   when several devices are
                                   managed by the same module */
  struct pt_regs *regs          /* cpu register snapshot, rarely
                                   needed*/
);
```

Return value:

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

top half and bottom half processing (1)

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

top half and bottom half processing (2)

- Declare the tasklet in the module source file:

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

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

```
tasklet_schedule(&module_do_tasklet);
```


Interrupt management summary

- Find an available interrupt number (if possible)
- Activate interrupts on the peripheral
- Detect the interrupt number used by the peripheral, by polling the several possibilities, if needed.
- Register the interrupt handler with the identified IRQ number.
- Once the interrupt handler is called, acknowledge the interrupt.
- In the interrupt handler, schedule a tasklet taking care of handling data
- In the tasklet, handle data
- In the tasklet, wake up waiting user processes
- Unregister the interrupt handler if the device is closed.

Advanced driver development

mmap

mmap

- Answers requests from the `mmap` glibc function:

```
void * mmap(void *start, size_t length, int prot,
            int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
```

- Makes it possible to provide user programs with direct access to device memory.
- Used by programs like X-Window servers. Much faster than other methods (like writing the corresponding `/dev` device) for user applications needing high bandwidth.

Process VMA (1)

Virtual Memory Areas: contiguous area in process virtual memory with the same permission flags.

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

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

Process VMA (2)

X server example (maps excerpt)

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


Simple mmap

To allow `mmap()` operations, the driver just needs to create memory page tables mapping a physical area

This can be done by the below function (`linux/mm.h`) to be called in a *`driver`*`_mmap` function:

```
int remap_page_range(
  struct vm_area_struct *vma,
  unsigned long from, /* Virtual */
  unsigned long to, /* Physical */
  unsigned long size, pgprot_t prot);
```

The corresponding *`driver`*`_mmap` function is then added to the driver's `file_operations` structure.

Example: `drivers/char/mem.c`

devmem2

http://www.lart.tudelft.nl/lartware/port/devmem2.c

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

- Very useful for early interaction experiments with a device, without having to code and compile a driver.
- Uses `mmap` to `/dev/mem`.
  Need to run `request_mem_region` and setup `/dev/mem` first.
- Examples (`b`: byte, `h`: half, `w`: word)
  `devmem2 0x000c0004 h` (reading)
  `devmem2 0x000c0008 w 0xffffffff` (writing)

Advanced driver development

DMA

DMA situations

Synchronous

- A user process calls the read method of a driver. The driver allocates a DMA buffer and asks the hardware to copy its data. The process is put is sleep mode.
- The hardware copies its data and raises an interrupt at the end.
- The interrupt handler gets the data from the buffer and wakes up the waiting process.

Asynchronous

- The hardware sends an interrupt to announce new data.
- The interrupt handler allocates a DMA buffer and tells the hardware where to transfer data.
- The hardware writes the data and raises a new interrupt.
- The handler releases the new data, and wakes up the needed processes.

Memory constraints

- Need to use contiguous memory in physical space
- Can use any memory allocated by `kmalloc` or `__get_free_pages`
- Can use block I/O and networking buffers...
- Can't use `vmalloc` memory
  (would have to setup DMA on each individual page)
- Using DMA doesn't suppress the need for memory barriers.
  Otherwise, your compiler may scramble read or write order.

Memory synchronization constraints

Memory caching could interfere with DMA

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

Consistent or streaming DMA mappings

- Consistent mappings

  Allocated for the whole time the module is loaded.

- Streaming mappings

  Set up for each transfer.

  Keep DMA registers free on the physical hardware registers. Some optimizations also available.

Generic DMA API reference

- Most subsystems supply their own DMA API
  May be sufficient for most needs
- Documentation/DMA-API.txt
  Linux DMA generic API description, in parallel with the corresponding PCI DMA API.

A useful document to show in this course!

Advanced driver development

New Device Model

Device Model features (1)

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

Device model features (2)

Allows to view the system for several points of view:

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

sysfs

- Userspace representation of the Device Model.
- Configure it with
  `CONFIG_SYSFS=y` (Filesystems -> Pseudo filesystems)
- Mount it with
  `mount -t sysfs /sys /sys`
- Spend time exploring `/sys` on your workstation!

sysfs tools

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

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

The device structure

Declaration

- The base data structure is `struct device`, defined in `include/linux/device.h`
- In real life, you will rather use a structure corresponding to the bus your device is attached to: `struct pci_dev`, `struct usb_device`...

Registration

- Still depending on the device type, specific register and unregister functions are provided

Device attributes

Device attributes can be read/written from/by userspace

```
struct device_attribute {
   struct attribute        attr;
   ssize_t (*show)(struct device * dev, char * buf, size_t count, loff_t off);
   ssize_t (*store)(struct device * dev, const char * buf, size_t count, loff_t off);
};

#define DEVICE_ATTR(name,mode,show,store)
```

Adding / removing from the device directory

```
int device_create_file(struct device *device, struct device_attribute * entry);
void device_remove_file(struct device * dev, struct device_attribute * attr);
```

Example

```
/* Creates a file named "power" with a 0644 (-rw-r--r--) mode */

DEVICE_ATTR(power,0644,show_power,store_power);
device_create_file(dev,&dev_attr_power);
device_remove_file(dev,&dev_attr_power);
```


The device driver structure

Declaration

```
struct device_driver {
    /* Omitted a few internals */
    char            *name;
    struct bus_type         *bus;
    int     (*probe)        (struct device * dev);
    int     (*remove)       (struct device * dev);
    void    (*shutdown)       (struct device * dev);
    int     (*suspend)        (struct device * dev, u32 state, u32 level);
    int     (*resume)       (struct device * dev, u32 level);
};
```

Registration

```
extern int driver_register(struct device_driver * drv);
extern void driver_unregister(struct device_driver * drv);
```

Attributes

Available in a similar way

Device Model references

Very useful and clear documentation in the kernel sources!

- `Documentation/driver-model/`

```
binding.txt  class.txt      driver.txt
overview.txt  porting.txt    bus.txt
device.txt   interface.txt platform.txt
```

- `Documentation/filesystems/sysfs.txt`

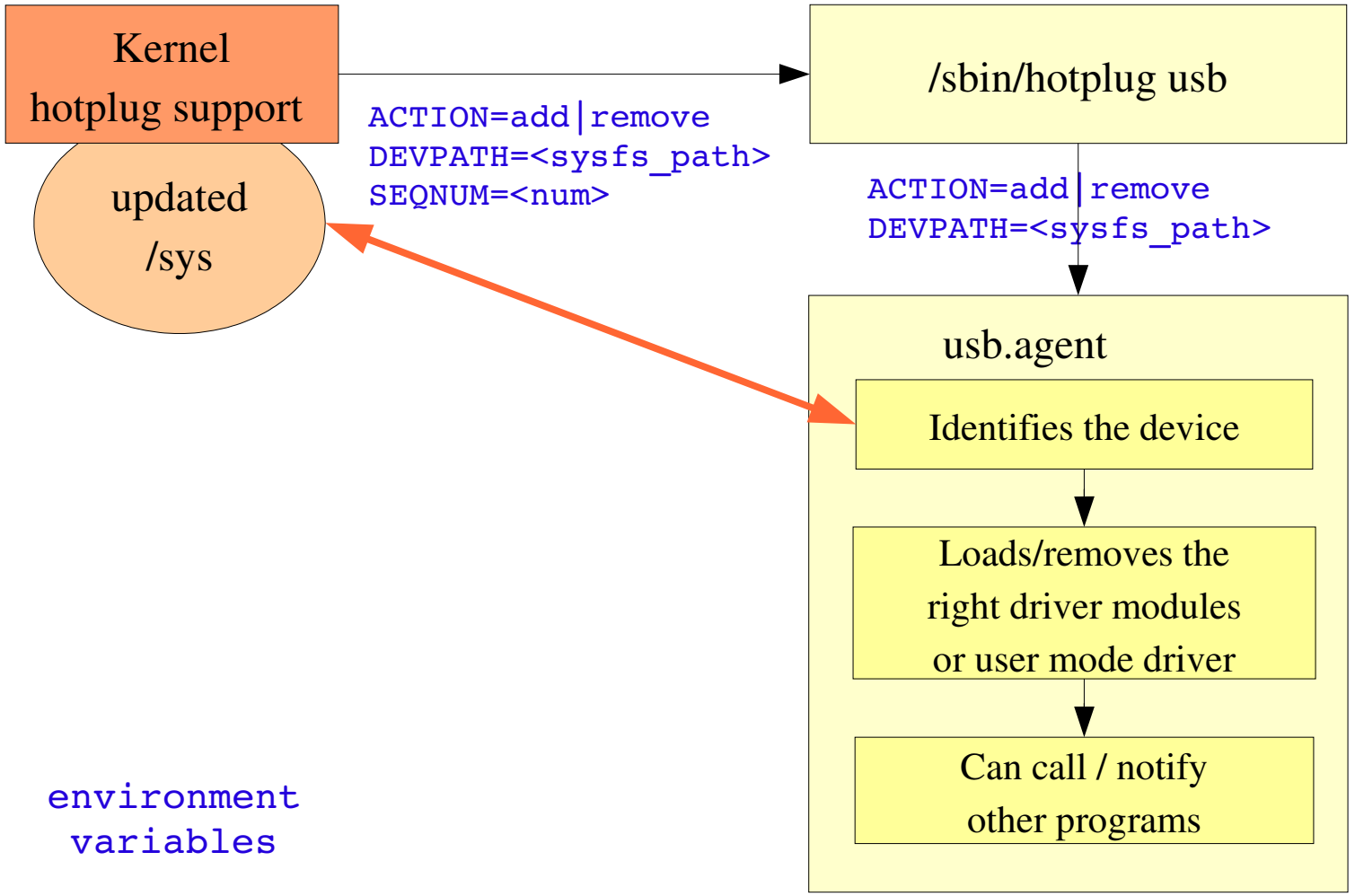
Advanced driver development

hotplug

hotplug overview

- Introduced in Linux 2.4. Pioneered by USB.
- Kernel mechanism to notify user space programs that a device has been inserted or removed.
- User space scripts then take care of identifying the hardware and inserting/removing the right driver modules.
- Linux 2.6: much easier device identification thanks to sysfs
- Makes it possible to load external firmware
- Makes it possible to have user-mode only driver (e.g. libsane)
- Kernel configuration:
  `CONFIG_HOTPLUG=y` ("General setup" section)

hotplug flow example



hotplug files

`/lib/modules/*/modules.*map`
depmod output

`/proc/sys/kernel/hotplug`
specifies hotplug program path

`/sbin/hotplug`
hotplug program (default path name)

`/etc/hotplug/*`
hotplug files

`/etc/hotplug/NAME*`
subsystem-specific files, for agents

`/etc/hotplug/NAME/DRIVER`
driver setup scripts, invoked by agents

`/etc/hotplug/usb/DRIVER.usermap`
depmod data for user-mode drivers

`/etc/hotplug/NAME.agent`
hotplug subsystem-specific agents

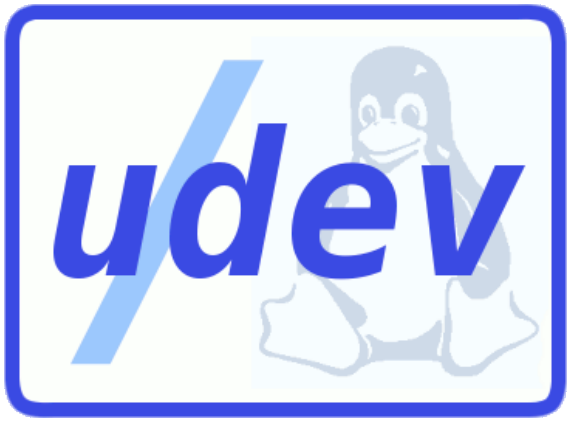
Firmware hotplugging

- Kernel configuration: needs to be set in `CONFIG_FW_LOADER` (`Device Drivers -> Generic Driver Options -> hotplug firmware loading support`)
- Need `/sys` to be mounted
- Location of firmware files: check `/etc/hotplug/firmware.agent`

See `Documentation/firmware_class` for nice overview

hotplug references

- Project page and documentation
  http://linux-hotplug.sourceforge.net/
- Mailing list:
  http://lists.sourceforge.net/lists/listinfo/linux-hotplug-devel



Advanced driver development

udev: user-space device file management

/dev issues and limitations

- On Red Hat 9, 18000 entries in `/dev`!
  All entries for all possible devices need to be created at system installation.
- Need for 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 doesn't know what devices are present in the system.
- Userspace can't tell which `/dev` entry is which device

devfs solutions and limitations

- Only shows present devices
- But uses different names as in `/dev`, causing issues in scripts.
- But no flexibility in device names, unlike with `/dev/`, e.g. the 1st IDE disk device has to be called either `/dev/hda` or `/dev/ide/hd/c0b0t0u0`.
- But doesn't allow dynamic major and minor number allocation.
- But requires to store the device naming policy in kernel memory. Can't be swapped out!

udev features

Takes advantage of both hotplug and sysfs

- Entirely in user space
- Automatically creates device entries (by default in `/udev`)
- Called by `/sbin/hotplug`, uses information from sysfs.
- Major and minor device numbers found in sysfs
- Requires no change to the driver code
- Small size

udev toolset (1)

Major components

- `udevsend` (8KB in FC 3)
  Takes care of handling the `/sbin/hotplug` events, and sending them to `udevd`
- `udevd` (12KB in FC 3)
  Takes care of reordering hotplug events, before calling `udev` instances for each of them.
- `udev` (68KB in FC 3)
  Takes care of creating or removing device entries, entry naming, and then executing programs in `/etc/dev.d/`

udev toolset (2)

Other utilities

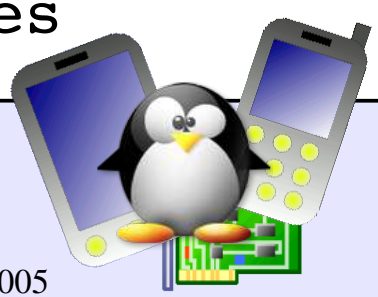
- `udevinfo` (48KB in FC 3)
  Lets users query the udev database
- `udevstart` (functionality brought by `udev`)
  Populates the initial device directory from valid devices found in the sysfs device tree.
- `udevtest <sysfs_device_path>` (64KB in FC 3)
  Simulates a `udev` run to test the configured rules

udev configuration file

`/etc/udev/udev.conf`

Easy to edit and configure

- Device directory (`/udev`)
- udev database file (`/dev/.udev.tdb`)
- udev rules (`/etc/udev/rules.d/`)
  udev permissions (`/etc/udev/permissions.d/`)
- default mode (`0600`), default owner (`root`) and group (`root`), when not found in udev's permissions.
- Enable logging (`yes`)
  Debug messages available in `/var/log/messages`



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

udev can also create device links

udev rules file example

```
# if /sbin/scsi_id returns "OEM 0815" device will be called disk1
BUS="scsi", PROGRAM="/sbin/scsi_id", RESULT="OEM 0815", NAME="disk1"

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

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

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

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

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

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


udev sample permissions

```
#name:user:group:mode
input/*:root:root:644
ttyUSB1:0:8:0660
video*:root:video:0660
dsp1:::0666
```


/etc/dev.d/

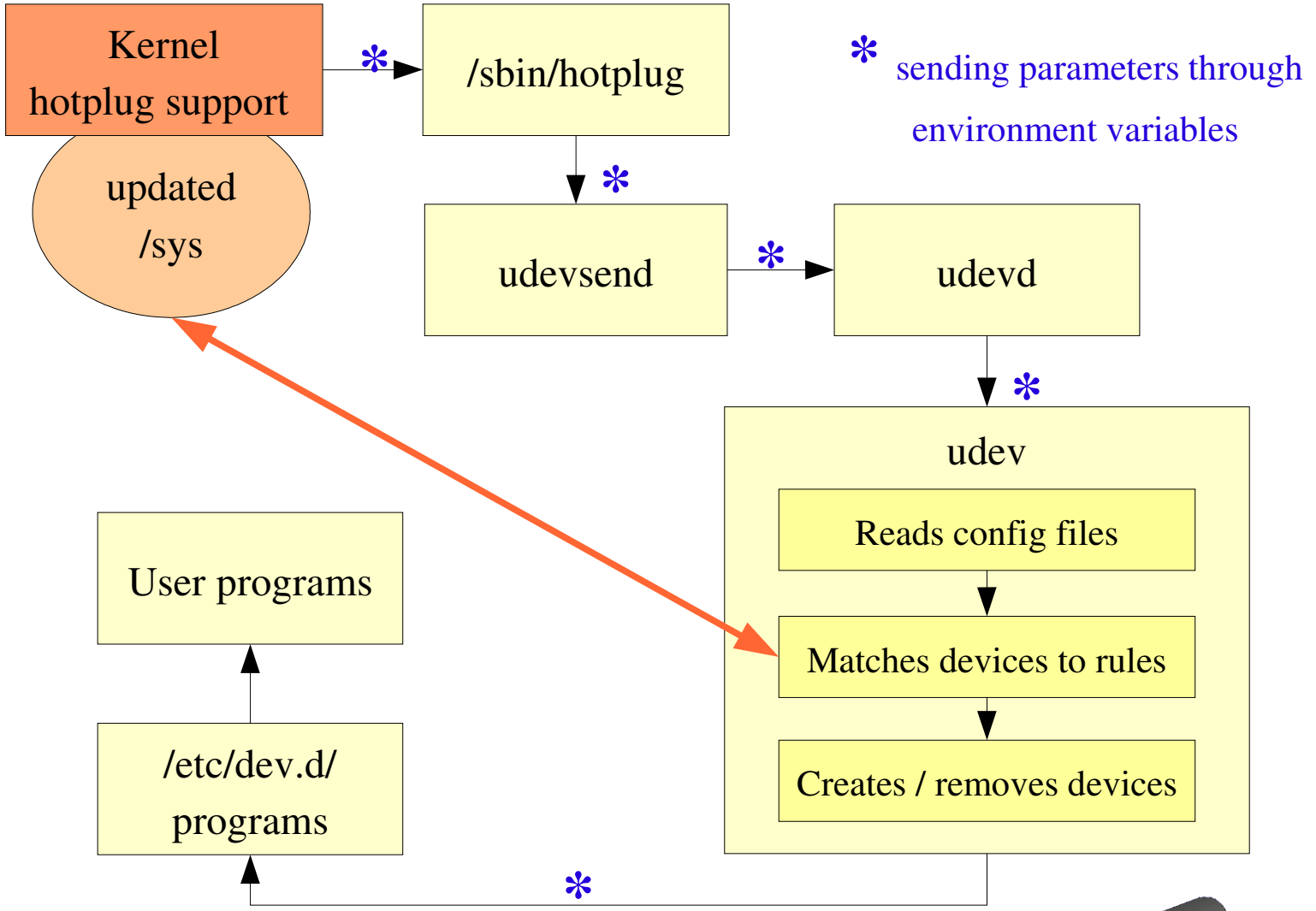
After device nodes are created, removed or renamed, udev can call programs found in the below search order:

- `/etc/dev.d/$(DEVNAME)/*.dev`
- `/etc/dev.d/$(SUBSYSTEM)/*.dev`
- `/etc/dev.d/default/*.dev`

The programs in each directory are sorted in lexical order.

This is useful to notify user applications of device changes.

udev summary



udev links

- Sources
  http://kernel.org/pub/linux/utils/kernel/hotplug/
- BitKeeper source tree
  bk://kernel.bkbits.net/gregkh/udev/
- Mailing list:
  linux-hotplug-devel@lists.sourceforge.net
- Greg Kroah-Hartman, udev presentation
  http://www.kroah.com/linux/talks/oscon_2004_udev/
- Greg Kroah-Hartman, udev whitepaper
  http://www.kroah.com/linux/talks/ols_2003_udev_paper/Reprint-Kroah-Hartman-OLS2003.pdf

Advice and resources

System security

- In production: disable loadable kernel modules if you can.
- Carefully check data from input devices (if interpreted by the driver) and from user programs (buffer overflows)
- Check kernel sources signature
- Beware of uninitialized memory
- Compile modules by yourself (beware of binary modules)

Advice and resources

Using Ethernet over USB

Ethernet over USB (1)

If your device doesn't have Ethernet connectivity, but has a USB device controller

- You can use Ethernet over USB through the `g_ether` USB device ("gadget") driver (`CONFIG_USB_GADGET`)
- Of course, you need a working USB device driver. Generally available as more and more embedded processors (well supported by Linux) have a built-in USB device controller
- Plug-in both ends of the USB cable

Ethernet over USB (2)

- On the PC host, you need to have the `usbnet` module (`CONFIG_USB_USBNET`)
- Plug-in both ends of the USB cable. Configure both ends as regular networking devices. Example:
  - On the target device
    ```
    modprobe g_ether
    ifconfig usb0 192.168.0.202
    route add 192.168.0.200 dev usb0
    ```
  - On the PC
    ```
    modprobe usbnet
    ifconfig usb0 192.168.0.200
    route add 192.168.0.202 dev usb0
    ```
- Works great on iPAQ PDAs!

Advice and resources

Root filesystem on the host through NFS

Usefulness of rootfs on NFS

Once you have setup networking (Ethernet or USB-Ethernet), you can mount a filesystem on the PC through NFS and use it as the new root filesystem. This is very convenient for system development:

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

Example NFS setup

On the PC

- Add the below line to your `/etc/exports` file:
  ```
  /home/rootfs 192.168.0.202/32(rw,insecure,sync,no_wdelay,no_root_squash)
  ```
- Start or restart your NFS server (Fedora Core 2 example)
  ```
  /etc/init.d/nfs restart
  ```

On the target

- ```
  mkdir /mnt/rootfs; mkdir /mnt/initrd
  modprobe nfs
  mount -o nolock,hard,intr -t nfs 192.168.0.200:$rootfs \
  /mnt/rootfs
  ```



  ```

Using pivot_root

Once the NFS share is mounted, you can use it as the new root filesystem:

- Example (continued)

```
umount /proc
cd /mnt/rootfs
pivot_root . mnt/initrd
exec chroot . /linuxrc <dev/console >dev/console 2>&1
```

- Same `pivot_root` usage for a local storage. Used in all GNU/Linux computers with an initrd.

Advice and resources

Choosing filesystem types

Block device or MTD filesystems

Block devices

- Floppy or hard disks (SCSI, IDE)
- Compact Flash (seen as a regular IDE drive)
- RAM disks
- Loopback devices

Memory Technology Devices (MTD)

- Flash, ROM or RAM chips
- MTD emulation on block devices

See `Documentation/filesystems/` for details

Traditional block filesystems

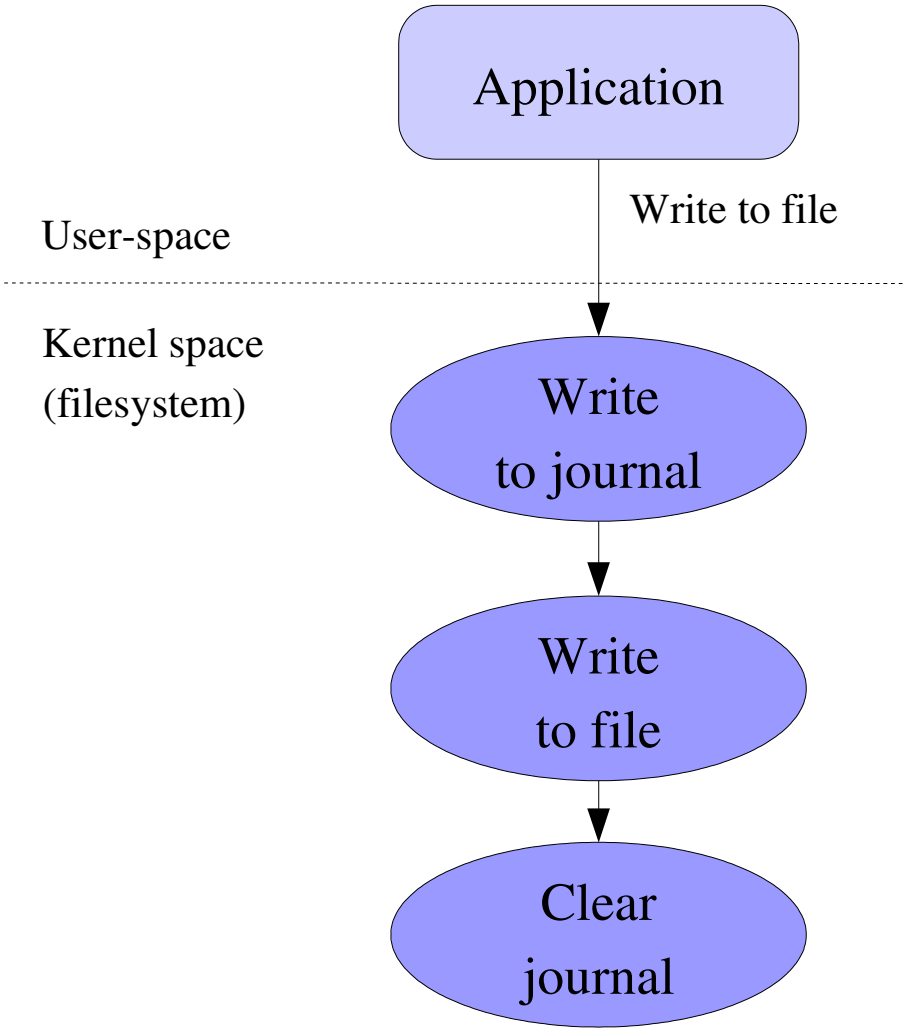
Traditional filesystems:

Hard to recover from crashes. Can be left in a corrupted state after a system crash or sudden power-off.

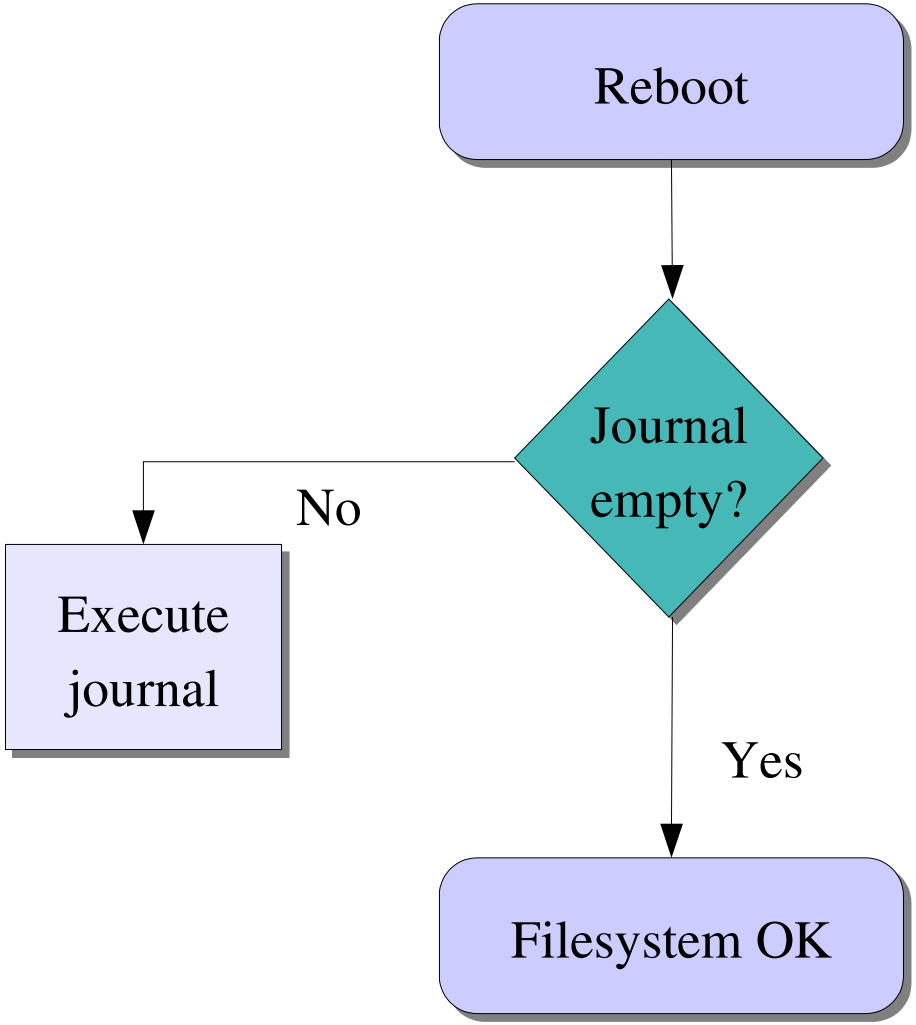
- ext2: traditional Linux filesystem
  (repair it with `fsck.ext2`)
- vfat: traditional Windows filesystem
  (repair it with `fsck.vfat` on GNU/Linux or `Scandisk` on Windows)

Journaled filesystems

- Designed to stay in a correct state even after system crashes or a sudden power-off
- All writes are first described in the journal before being committed to files



Filesystem recovery after crashes



- Thanks to the journal, the filesystem is never left in a corrupted state
- Recently saved data may be lost though

Journaled block filesystems

Journaled filesystems

- ext3: ext2 with journal extension
- reiserFS: most innovative
- Others: JFS (IBM), xfs (SGI)
- NTFS: well supported by Linux in read-mode

Read-only block filesystems

- ISO 9660: used for cdroms
- UDF: used in some cdroms and DVDs
- CramFS: simple, small, compressed filesystems designed for ROM based embedded systems
  (Size < 256 MB, files < 16 MB)

The MTD subsystem

Linux filesystem interface

MTD "User" modules

jffs2

Char device

Block device

Read-only block device

Flash Translation Layers

Caution: patented algorithms!

FTL

NFTL

INFTL

MTD Chip drivers

CFI flash

RAM chips

NAND flash

DiskOnChip flash

ROM chips

Block device

Virtual memory

Virtual devices appearing as MTD devices

Memory devices hardware

MTD filesystems

- jffs2: Journaling Flash File System v2
  - Designed to write flash sectors in an homogeneous way. Flash bits can only be rewritten a relatively small number of times (often < 100 000).
  - Compressed to fit as many data as possible on flash chips. Also compensates for slower access time to those chips.
  - Power down reliable: can restart without any intervention
  - Best choice for your internal flash chips
  - Can of course be mounted as a read-only filesystem

Using block filesystems over MTD

- Can use Flash Translation Layer modules implementing a virtual block device on top of MTD. Can use a regular block filesystem on top of this virtual device then.
- FTL: Flash Translation Layer for NOR flash chips
  Caution: because of patents on algorithms, can only be used on PCMCIA hardware in the US! Better use JFFS2.
- NTFL: NAND Flash Translation Layer.
  Caution: because of M-Systems algorithm patents, can only be implemented on licensed Disc On Chip devices.

Filesystem choices for your flash devices

MTD devices: use JFFS2 (read-write or read-only)

Compact Flash or other removable storage

- Can't use JFFS2 because CF storage is a block device.
  MTD Block device emulation could be used though, but JFFS2 writing scheme could interfered with on-chip flash management (manufacturer independent).
- **Never use block device journaled fs on flash chips!** Keeping the journal would write the same sectors over and over again and quickly damage them.
- Can use ext2 or vfat (caution: patents), with mount options:
  - `noatime`: doesn't write access time information in file inodes
  - `sync`: to perform writes immediately (avoid power down fs failure)

Mounting a jffs2 image

Useful to create or edit jffs2 images on your GNU / Linux PC!

- Mounting an MTD device as a loop device is a bit complex task. Here's an example for jffs2:

```
modprobe loop
modprobe mtdblock
losetup /dev/loop0 <file>.jffs2
modprobe blkmtd erasesz=256 device=/dev/loop0
mknod /dev/mtdblock0 b 31 0    (if not done yet)
mkdir /mnt/jffs2               (example mount point, if not done yet)
mount -t jffs2 /dev/mtdblock0 /mnt/initrd/
```

- It's very likely that your standard kernel misses one of these modules. Check the corresponding `.c` file in the kernel sources and look in the corresponding Makefile which option you need to recompile your kernel with.

Advice and resources

Getting help and contributions

Solving issues

- If you face an issue, and it doesn't look specific to your work but rather to the tools you are using, it is very likely that someone else already faced it.
- Search the Internet for similar error reports
  - On web sites or mailing list archives (using a good search engine)
  - On newsgroups: http://groups.google.com/
- You have great chances of finding a solution or workaround, or at least an explanations 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 on mailing lists
  - 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
  - Refrain from contacting the Linux Kernel mailing list, unless you're an experienced developer and need advice

Getting contributions

Applies if your project can interest other people:

developing a driver or filesystem, porting Linux on a new device available on the market...

External contributors can help you a lot by

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

Encouraging contributions

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

Advice and resources

Bug report and patch submission to Linux developers

Reporting Linux bugs

- First make sure you're using the latest version
- Make sure you investigate the issue as much as you can: see `Documentation/BUG-HUNTING`
- Make sure the bug has not been reported yet. Check the Official Linux kernel bug database (http://bugzilla.kernel.org/) in particular.
- 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.
- Or file a new bug in http://bugzilla.kernel.org/

How to create Linux patches

- Download the **latest** kernel sources
- Make a copy of these sources:
```
rsync -a linux-2.6.9-rc2/ linux-2.6.9-rc2-patch/
```
- Apply your changes to the copied sources, and test them.
- Create a patch file:
```
diff -Nru linux-2.6.9-rc2/ \
linux-2.6.9-rc2-patch/ > patchfile
```
  - Always compare the whole source structures (suitable for `patch -p1`)
  - Patch file name: should recall the addressed issue

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

How to submit patches or drivers

- Don't merge patches addressing different issues
- You should identify and contact the official maintainer for the files to patch.
- See `Documentation/SubmittingPatches` for details. For trivial patches, you can copy the Trivial Patch Monkey.
- Special subsystems:
  - ARM platform: it's best to submit your ARM patches to Russel King's patch system: http://www.arm.linux.org.uk/developer/patches/

Advice and resources

References

Information sites (1)

Linux Weekly News

http://lwn.net/

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

Information sites (2)

KernelTrap

http://kerneltrap.org/

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

Useful reading

- Linux Device Drivers, 3rd edition, Feb 2005
  Jonathan Corbet, Alessandro Rubiny, Greg Kroah-Hartman, O'Reilly
  http://www.oreilly.com/catalog/linuxdrive3/
  The 2nd edition (2001) is available on-line on a free documentation license:
  http://www.xml.com/ldd/chapter/book/index.html
  A must-have book for Linux device driver writers!

- Linux Kernel Development, Second Edition, Jan 2005
  Robert Love, Novell Press
  http://rlove.org/kernel_book/
  A very synthetic and pleasant way to learn about the kernel!

- Building Embedded Linux Systems, April 2003
  Karim Yaghmour, O'Reilly
  http://www.oreilly.com/catalog/belinuxsys/
  Not very fresh, but doesn't depend too much on kernel versions

References

- 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/

  Glossaries, articles, presentations, HOWTOs, recommended reading, useful tools for people getting familiar with Linux kernel or driver development.

ARM resources

Processor docs

- ARM manuals: http://www.arm.com/documentation/
- Full ARM technical publications cdrom
  (free-as-free-beer order)
  http://www.arm.com/documentation/cd_request.html

ARM Linux Project

- Home page:
  http://www.arm.linux.org.uk/
- Developer documentation:
  http://www.arm.linux.org.uk/developer/
- arm-linux-kernel mailing list:
  http://lists.arm.linux.org.uk/mailman/listinfo/linux-arm-kernel
- FAQ:
  http://www.arm.linux.org.uk/armlinux/mlfaq.php
- How to post kernel fixes:
  http://www.arm.uk.linux.org/developer/patches/

Advice and resources

Last advice

Use the Source, Luke!

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

Thanks to LucasArts

Related documents

This document belongs to the 750 page materials of an embedded GNU / Linux training from Free Electrons, available under the GNU Free Documentation License.

- Introduction to Unix and GNU / Linux
  http://free-electrons.com/training/intro_unix_linux
- Embedded Linux kernel and driver development
  http://free-electrons.com/training/drivers
- Development tools for embedded Linux systems
  http://free-electrons.com/training/devtools
- Audio in embedded Linux systems
  http://free-electrons.com/training/audio
- Multimedia in embedded Linux systems
  http://free-electrons.com/training/multimedia
- Java in embedded Linux systems
  http://free-electrons.com/articles/java
- What's new in Linux 2.6?
  http://free-electrons.com/articles/linux26
- Introduction to uClinux
  http://free-electrons.com/articles/uclinux
- Linux real-time extensions
  http://free-electrons.com/articles/realtime

Training labs

Training labs are also available from the same location:

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

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

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

Training and consulting services

This training or presentation is funded by Free Electrons customers sending their people to our training or consulting sessions.

If you are interested in attending training sessions performed by the author of these documents, you are invited to ask your organization to order such sessions.

See http:/free-electrons/training for more details.

If you just support this work, do not hesitate to speak about it to your friends, colleagues and local Free Software community.