

Systemsoftware

Linux Kernel

Prof. Dr. Michael Mächtel

Informatik, HTWG Konstanz

Version vom 06.03.17

Übersicht

1 Kernel Source

2 Kernel Configuration

3 Kernel Build

4 Kernel Boot

5 Kernel Bootparameter

6 Kernel Device Tree

Übersicht

1 Kernel Source

2 Kernel Configuration

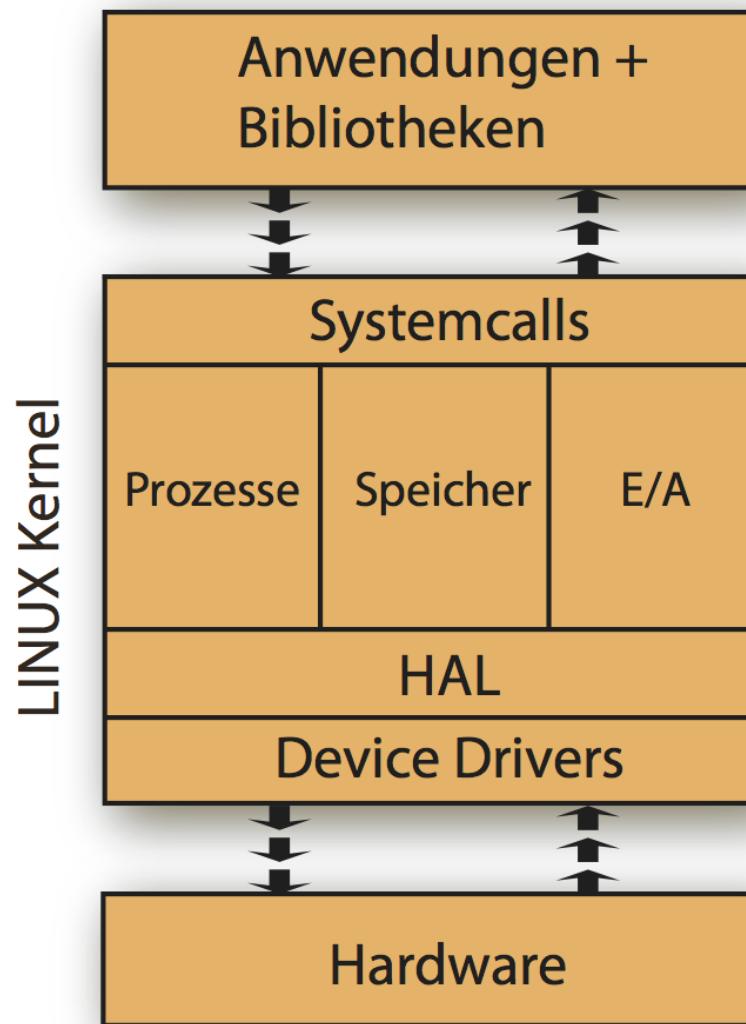
3 Kernel Build

4 Kernel Boot

5 Kernel Bootparameter

6 Kernel Device Tree

Architektur Embedded Linux System



Prozess-/Speicher-Subsystem

- Prozess-Management
 - Verteilung der Ressource CPU.
 - Single-Core-Scheduling.
 - Multi-Core-Scheduling.
- Speicher-Management
 - Adressumsetzung.
 - Speicherschutz.
 - Virtuellen Speicher (Swap-Space).
 - Extended Memory.

E/A-/Treiber-Subsystem

- E/A-Subsystem
 - Einheitliches Programmierinterface (API) für den Zugriff auf Dateien und Geräte.
 - Systemkonforme Integration von Hardware in den Kernel (Treiberinterface).
 - Performanter Zugriff auf E/A (Stichworte Page-Cache, E/A-Scheduling).
 - Filesysteme.
- Gerätetreiber
 - Standardisierter Zugriff auf Hardware.
 - Für jedes Peripheriegerät, das aus der User-Ebene angesprochen werden soll, muss ein Gerätetreiber existieren.
 - Alle Kernel-Komponenten, somit auch die Gerätetreiber, haben Zugriff auf den Kernel-Adressraum.

Beispiele Gerätetreiber

- Character-Devices
- Block-Devices
- Netzwerk
- SCSI
- USB
- Irda
- Cardbus und PCMCIA
- Parallelport
- I2C
- I2O
- ...

Linux Kernel

- Quelle <http://www.kernel.org>
 - wget <linux-3.xx.yy.tar.bz2 File>
 - tar xvzf <linux-3.xx.yy.tar.bz2 File>
 - Kernel Sourcen im linux-3.xx.yy Verzeichnis
- Kernel-Versionsnummern: 3.xx.yy
 - Aktuell: Version 3 (Sprung von 2.6.40 auf 3.0)
 - 3.xx aktuelle Major-Version (z.B. 3.1, 3.2, 3.3 ...)
 - .yy Fehlerbereinigung (BugFix Releases)
- Kernel in C:
 - Grund: Effizienz, wenige Zeilen Assembler für Hardware Initialisierung, Interrupts sowie zeitkritische Funktionen
 - LoC: ca. 15 Millionen LOC
 - SLOC¹ ca. 10 Millionen ‘physical LoC’: ‘non-blank, non-comment lines’

¹<http://www.dwheeler.com/sloccount/>

Kernel Patchen

- Ein Kernel-Patch ist ein Diff-File
 - wird mit Hilfe der Option **diff -urN <orginaldatei> <neuedatei>** erzeugt
- Im Patchfile sind im Regelfall auch die Namen der Verzeichnisse.
- Zum Patchen wechselt man in das Quellverzeichnis
 - und verwendet die Option -p1. Dadurch wird der erste Verzeichniseintrag entfernt.
- Ablauf Kernel patchen:
 - Kernel-Quellen downloaden,
 - Bug-Fixes als Patch downloaden
- Zur Vergleich der Patches **diff** benutzen

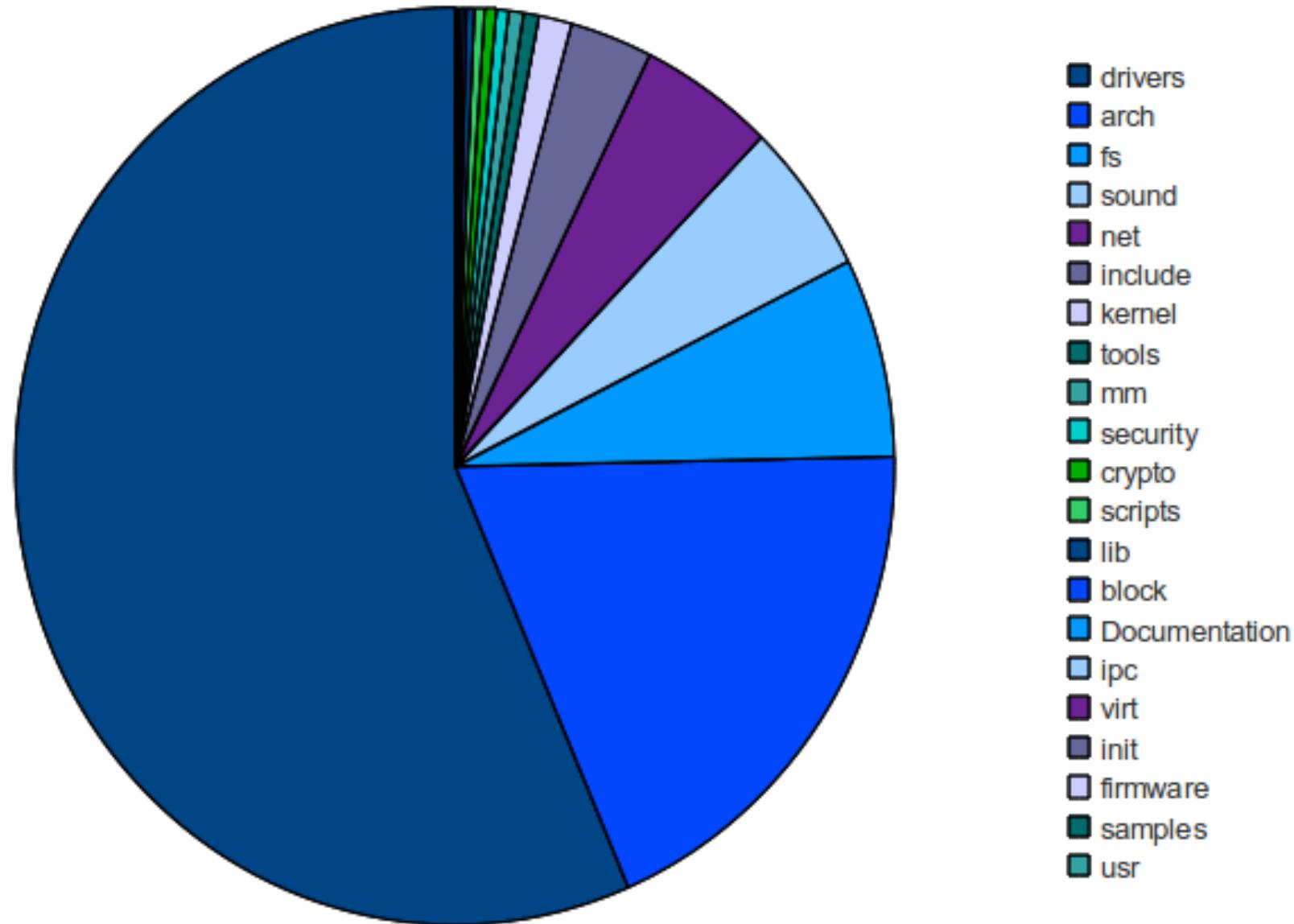
```
$ diff patch1 patch2
```

Programmiersprachen

Tabelle Kernel 3.2: Programmiersprachen

Sprache	Lines of Code (SLOC)
ansic:	5215716 (95.78%)
asm:	216350 (3.97%)
perl:	4215 (0.08%)
yacc:	2637 (0.05%)
sh:	2233 (0.04%)
cpp:	2129 (0.04%)
lex:	1510 (0.03%)
python:	331 (0.01%)
lisp:	218 (0.00%)
awk:	96 (0.00%)
lisp:	218 (0.00%)
sed:	30 (0.00%)

Code Verteilung



2

²Quelle: <http://www.heise.de>

GCC Erweiterungen

- Linux Kernel benutzt einige der GNU C Erweiterungen:
 - Inline C Funktionen,
 - Inline Assembler,
 - Initialisierung von Elementen einer Struktur unabhängig von deren Reihenfolge,
 - Branch Anweisungen *likely()* und *unlikely()*
 - ...

Kernelcode Besonderheiten

- Im Linux Kernel können keine libc Funktionen benutzt werden:
 - Linux Kernel muss ‘standalone’ benutzbar sein
 - kann keinen User-Code benutzen
 - Kernel bietet eigene Funktionen wie *printk()*, *memset()* und *kmalloc()*, die den Standard C Funktionen sehr ähnlich sind.
- keine Nutzung von Gleitkommazahlen im Kernel, da Kernel Gleitkommazahlen emuliert.
- Um im Namensraum Konflikte zu vermeiden, alle Symbole als static definieren, ausser natürlich exportiere Symbole (Stichwort: Namespace Pollution).
- Für weitere Informationen siehe *Documentation/CodingStyle*.

Kernel Dokumentation

- Kernel Source Verzeichnis: *Documentation/*
- Zukünftige geplante Änderungen:
Documentation/feature-removal-schedule.txt
- Zusammenfassung der wichtigsten Änderungen:
<http://kernelnewbies.org/LinuxChanges>
- Linux Kernel Berichte: <http://lwn.net/Kernel/>

Kernel Hardwarearchitekturen

- Die unterstützten Hardware Architekturen finden sich im Verzeichnis *arch/*
- Der Mainstream Kernel benötigt als Minimum eine 32 Bit Architektur
- 32 Bit Architekturen: alpha, arm, avr32, cris, frv, h8300, i386, m32r, m68k, m68knommu, mips, parisc, powerpc, ppc, s390, sh, sparc, um, v850, xtensa, ...
- 64 Bit Architekturen: ia64, mips, powerpc, sh64, sparc64, x86_64, ...
- Weitere Informationen siehe auch *arch/<arch>/README*, oder *Documentation/<arch>/*

Kernel Makefile

- Makefile = BUILD Zentrale des Kernels
 - Das Linux *Makefile* nutzt so gut wie jedes Feature von GNU Make aus!
 - **make help**: Übersicht aller Linux Kernel Makefile ‘Kommandos’
- Anpassungen im Makefile selbst:
 - Compiler Erweiterungen wie z.B. ccache oder distcc
 - EXTRAVERSION Setzen (wenn gewünscht)
- gewünschte Architektur lässt sich über Shell Variable **ARCH=<arch>** setzen
 - **\$ ARCH=<arch> make**
- Alle Kernel Konfigurationsparameter werden in der Datei **.config** gespeichert.

Übersicht

- 1 Kernel Source
- 2 Kernel Configuration
- 3 Kernel Build
- 4 Kernel Boot
- 5 Kernel Bootparameter
- 6 Kernel Device Tree

Hilfe zum Makefile

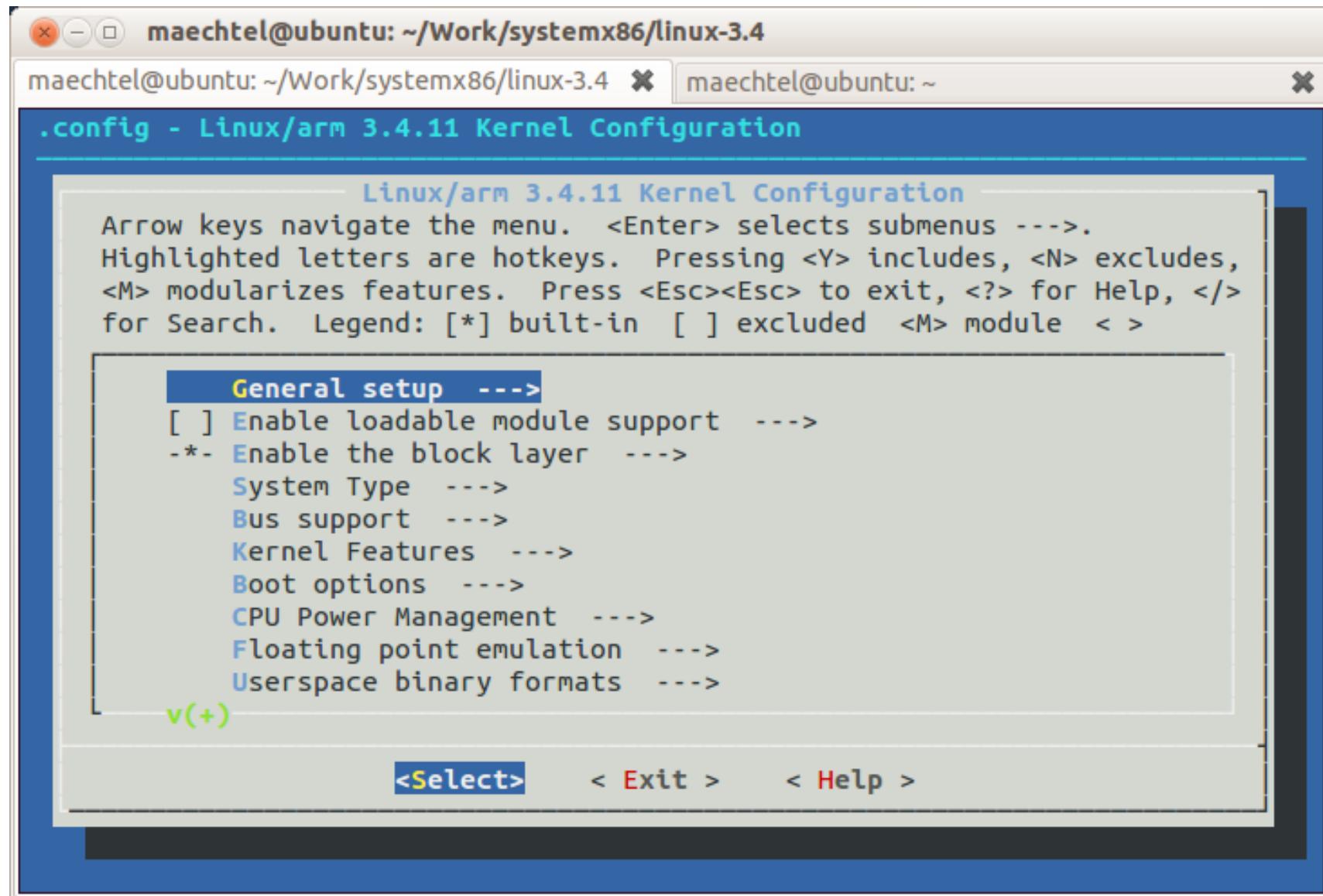
- **make help**

- Listet alle make Optionen auf.
- Hilfreich, um einen Überblick über die Optionen zu bekommen und auch um neue Optionen (bei neuen Kernel-Releases) zu 'entdecken'.

Kernel Konfiguration

- Verschiedene Konfigurationsinterfaces:
 - **make config**: Fragt jede Kernel Einstellung (>1000) nacheinander ab. Extrem lang!
 - **make menuconfig**: Text Interface zur Konfiguration.
 - **make xconfig**: QT Interface
 - **make gconfig**: GTK Interface

make menuconfig



Kernel .config

- *.config* Datei repräsentiert Ihre Kernel Konfiguration!
 - Sichern!
 - z.B. `cp .config CONFIG_BACKUP_<text>`.
- Falls aktiviert, kann die Konfiguration auch aus dem gerade laufenden Kernel ausgelesen werden: `zcat /proc/config.gz`
 - siehe *General Setup -> Kernel .config support*
- **make oldconfig:**
 - Nützlich um die *.config* Datei einer früheren Kernel-Version upzudaten.
 - Gibt Warnung aus, für nicht mehr enthaltene Optionen.
 - Fragt nach Werten für neue Optionen
 - Wenn die Datei *.config* von Hand editiert wird, ist es ratsam, danach `make oldconfig` auszuführen!

Optionen der Kernel Konfiguration

- **make allnoconfig**
 - Minimale Kernel Config. Idealer Startpunkt für Embedded Systems!
 - Setzt nur die unbedingt benötigten Optionen zum Kompilieren des Kernels auf y. Alles andere auf n.
 - Erzeugt keinen Bootfähigen Kernel!
- **make V=1**
 - Um die komplette Commandline von gcc, ld usw. und deren Ausgabe zu sehen
- **make clean**
 - Löschen der durch Make generierten Obj-Files
- **make mrproper** oder **make distclean**
Löschen aller durch make generierten Dateien, sowie der *.config* Datei. ACHTUNG!

Minimale Kernel Konfiguration

- **make allnoconfig** bereitet initialisiert eine Embedded Kernel Konfiguration
- Weitere wichtige Parameter:
 - CPU Typ
 - Ausführbarer Dateityp (ELF)
- Wo liegt das RootFS
 - evtl. Bussysteme, um auf gewähltes Device zuzugreifen (z.B. PCI ...)
 - Devicetyp: IDE, SCSI, INITRD, ...
 - Filesystem: RAMFS, INITRAMFS
- Wie vorgehen:
 - Erstellen Sie sich ein Blockschaltbild über Ihr Embedded Device, auf welchem Linux gebootet werden soll
 - Gilt auch für eine Emulation: Welche Devices unterstützt der Emulator, welche Konfiguration ist dafür im Kernel nötig

Kernel Module

- Der Linux-Kernel ist modular aufgebaut.
 - Die Funktionalität des Linux Kernels kann über Module sehr einfach erweitert werden.
 - Treiber werden ebenfalls meist als Module realisiert.
 - Zum Laden/Entladen der Module wird Infrastruktur benötigt:
 - **insmod**
 - **rmmod**
 - ...
- Funktionalität kann auch statisch im Kernel gebunden werden:
 - alle Treiber werden direkt in den Kernel eincompiliert
 - Vorteil: Keine Infrastruktur für das Laden der Module im RootFS nötig

Übersicht

1 Kernel Source

2 Kernel Configuration

3 Kernel Build

4 Kernel Boot

5 Kernel Bootparameter

6 Kernel Device Tree

Kernel Build

- Zum Kompilieren: **make**, bzw. **ARCH=<arch> make**
- Wird eine Multiprozessormaschine eingesetzt mit n Prozessoren, so kann man make anweisen, mehrere Dateien parallel zu übersetzen:
 - **make -j <n>**
 - Wann immer möglich, wird make nun Dateien echt parallel übersetzen.
 - make -j 2 oder make -j 3 auf einer Einprozessormaschine hilft nicht sehr viel weiter (mögliche Zeitersparnis: 10%).

ccache

- Quelle: <http://ccache.samba.org/>
- Für den Kernel Build sehr nützlich wenn sich die .config Datei sehr oft geändert.
 - Und genau das passiert, wenn an der Konfiguration des Kernel ‘experimentiert’ wird.
- Änderungen im Makefile:

```
... CC = ccache $(CROSS_COMPILE)gcc
HOSTCC = ccache gcc ....
```
- Benchmarks:
 - 63%: Fedora Core 3 config file (viele Module!)
 - 82%: mit einem Embedded Linux config file (wenig Module!)

distcc

- Quelle <http://distcc.samba.org/>
- Aufsetzen des **distccd** auf allen Servern
- Wichtig: gleichen gcc Version auf allen Rechnern
- Auf dem Rechner, der die Kernel-Quellen übersetzt, müssen im Kernel Makefile folgende Anpassungen vorgenommen werden:

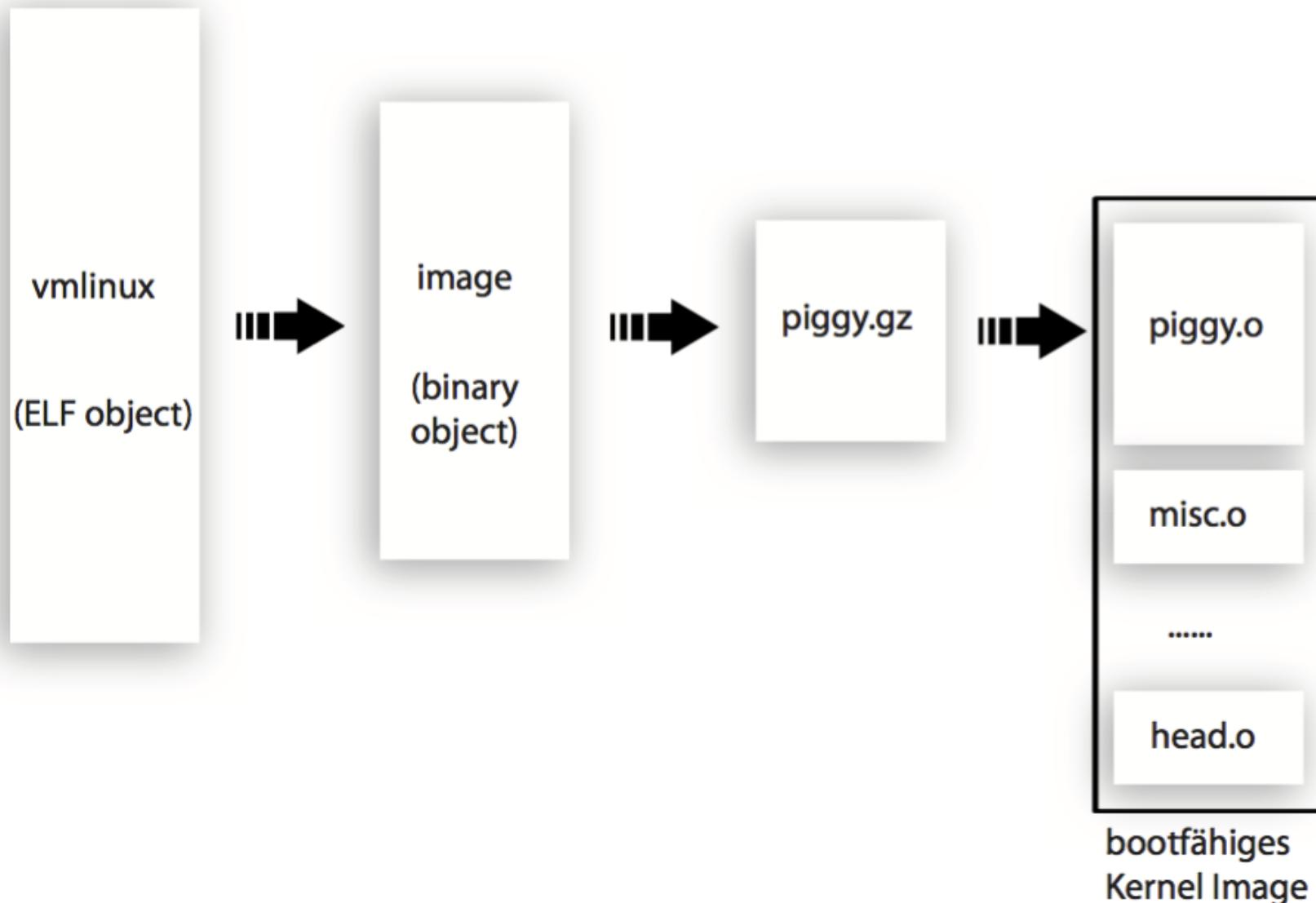
CC = \$(CROSS_COMPILE) distcc

HOSTCC = distcc

....

- Kombination von **distcc** und **ccache** möglich!

Kernel Build Prozess



Kernel Output

System.map Symboltabelle für die Zuordnung Objekt -> Adresse
(wichtig beim Debuggen)

vmlinux komplettes ELF Binary ('kernel proper'), wird
unabhängig von der Zielarchitektur erstellt.

vmlinux.bin (Image im compressed Ordner) Kernel Binary ohne
Symbole, Bemerkungen und Kommentare ('stripped')

Target Objects head.o, misc.o Startup-Code und Kernel
Entpack-Code

piggy.gz/piggy.o 'Image' komprimiert, bzw. als linkbares Objekt (.o)

zImage/bzImage komprimiertes Kernel Image für Bootlader

Übersicht

1 Kernel Source

2 Kernel Configuration

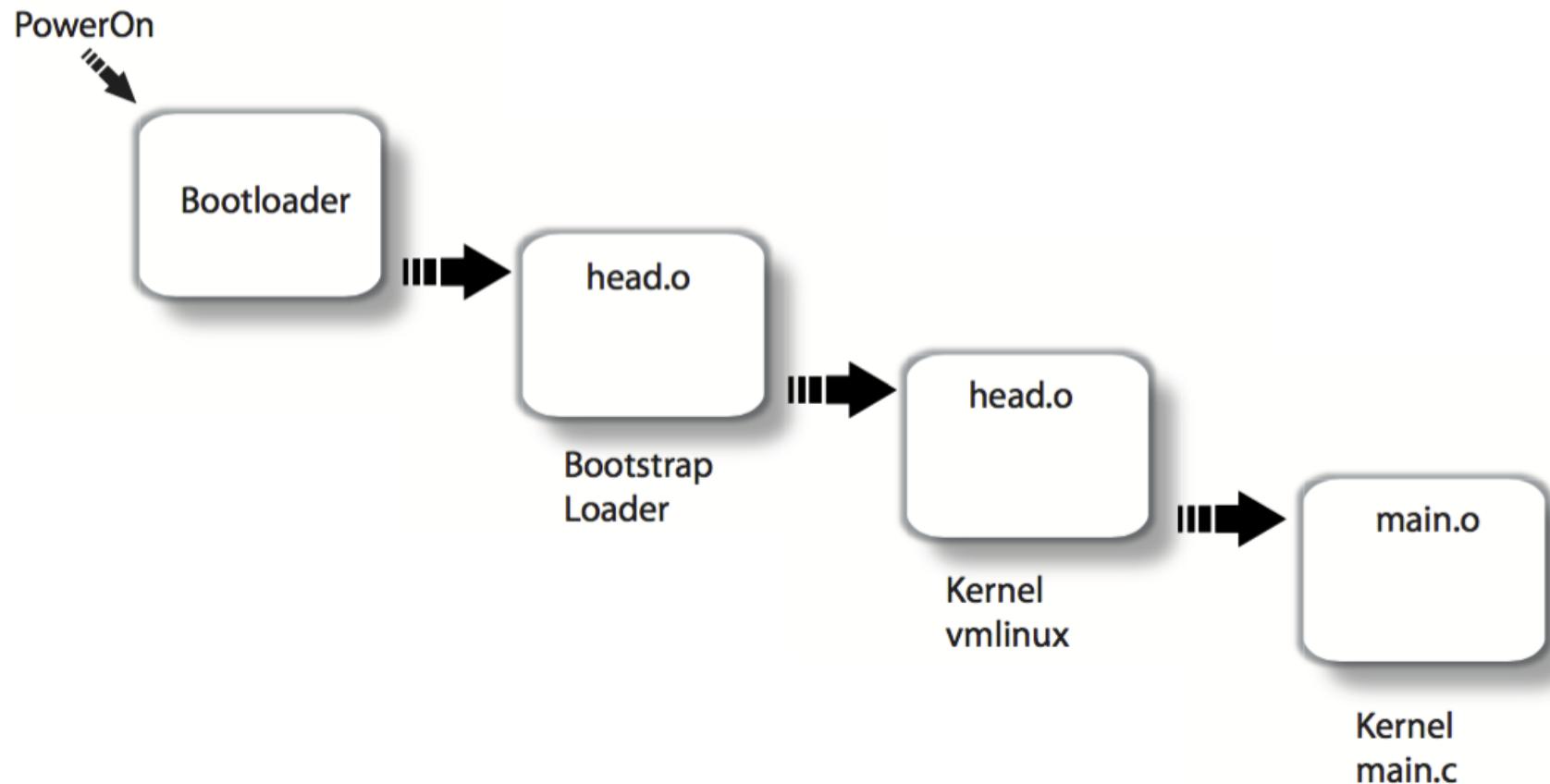
3 Kernel Build

4 Kernel Boot

5 Kernel Bootparameter

6 Kernel Device Tree

Kernel Initialisierung



vmlinux: head.o

- Überprüfung, ob es sich um den richtigen Prozessor/Architektur handelt
- Aufsetzen der Page Tables für die Speicherverwaltung
- Initialisieren und Aktivierung der MMU
- Aufsetzen eines einfachen Error Handlings
- Sprung zur *start_kernel()* Funktion des Kernels: main.c

Aufgabe der `start_kernel()`

- Architektur Setup (`setup_arch()`)
- Verarbeitung der Kernel Command Line
- Initialisierung der Subsysteme
- **init** Thread starten
 - **Init** vom RootFS !
- `start_kernel()` wird zum idle Thread

Linux benötigt RootFS

- Standardbetriebssysteme und insbesondere auch Linux benötigen zum Betrieb ein **Filesystem**, auf dem diverse Programme und Konfigurationsdaten abgelegt werden (RootFS).
- Das System besteht logisch aus zwei Images.
 - Das erste Image ist der Kernel
 - das zweite Image das RootFS

Kernel- /RootFS-Image

- Kernel-Image:
 - Einbinden der notwendigen Treiber
 - Compilieren für den entsprechenden Prozessor
 - Herausnehmen aller nicht benötigten Komponenten
 - Module: ja oder nein?
- RootFS-Image:
 - Systemprogramme
 - Konfigurationssoftware
 - Applikationen

Übersicht

1 Kernel Source

2 Kernel Configuration

3 Kernel Build

4 Kernel Boot

5 Kernel Bootparameter

6 Kernel Device Tree

Initialisierung

- Beim Booten des Linux Kernels wird u.a. die Kernel Commandline ausgewertet
- Treiber/Subsysteme benutzen Makros für das Aufsetzen der `setup()` Funktionen: ‘`__setup("console=", console_setup);`’
- Dadurch wird in einem speziellen Code-Segment (`.init.setup`) eine Liste der Setup Funktionen erstellt.
- Die Funktion zum Parsen der Command Line sucht für jeden Parameter die entsprechende Funktion aus der Liste
- Weitere Funktionen die nur zur Initialisierung benötigt werden, befinden sich ebenfalls in diesem Code Segment
- Nach dem Booten wird dieses Code Segment wieder freigegeben: Bootausgabe: ‘Freeing init memory: 112 K’

Wichtige Bootparameter

- **root**= Angabe des Devices, auf welchem sich das Root Filesystem befindet
- **ro/rw** : Mounted das Root als read-only/read-write Gerät
- **console**= Angabe der Geräte, welche der Linux Kernel als Console benutzen soll.
- **mem**= Wieviel Speicher der Kernel benutzen soll
- **initrd**= Initial RAM Disk Image (startet dort /linuxrc)
- **init**= Angabe eines alternativen Binaries statt '/sbin/init' (Block-Device Default)
- **rdinit**= Angabe eines alternativen Binaries statt '/init' (initramfs Default)
- **ip**= Angaben zur Art und Weise der Netzkarten Konfiguration (siehe nfsroot.txt)

Übersicht

1 Kernel Source

2 Kernel Configuration

3 Kernel Build

4 Kernel Boot

5 Kernel Bootparameter

6 Kernel Device Tree

Problemstellung

- Jede ARM-Plattform bringt eigene Treiber und Module ein:
 - bläht den Kernel-Quellcode zusehends auf.
 - ARM-SoCs Adressenlagen und zugehörige Interrupts unterscheiden sich, oft auch die Ansteuerlogik.
 - Eine in Hard- und Firmware realisierte Konfigurationsschnittstelle wie PCI hat sich in der ARM-Szene nicht etabliert.
 - Das auf PCs gültige Linux-Motto “Compile once, run everywhere” ist auf dem Weg zu ARM verloren gegangen.

Lösung

- Device Trees:
 - die jeweilige Hardwarekonfiguration wird weder fest in den Kernel einkompiliert noch als ewig langen Bootparameter übergeben,
 - sondern wird vor dem Startup als im Hauptspeicher abgelegte Datenstruktur bereit gestellt.
- Ablauf:
 - Eine Board-Firmware oder der Bootloader kopieren die Device Trees in den Hauptspeicher
 - Kernel wie Bootloader müssen Device Trees unterstützen.

Device Tree Blob

- Um mit Hilfe des Device Tree Compilers (DTC) den Device Tree Blob zu erzeugen, wird nach dem Generieren des Kernels im Kernelquellen-Root »make dtbs« aufgerufen.
- ARM Soc's
 - Der Device Tree Blob befindet sich nach erfolgreichem Übersetzen im Verzeichnis »arch/arm/boot/dts/«
 - Die Datei, also zum Beispiel »bcm2835-rpi-b.dtb«, kopieren Raspberry-Pi-Besitzer auf ihre SD-Karte in das Verzeichnis »/boot«.
 - Die Datei »config.txt« in /boot muss noch für die Verwendung der Device Trees angepasst werden.

Ablauf

