

# Systemsoftware

## Root Filesystem

Prof. Dr. Michael Mächtel

Informatik, HTWG Konstanz

Version vom 26.03.17

# Übersicht

1 RootFS

2 Anwendung

3 Busybox

4 Weitere Tools

# Übersicht

1 RootFS

2 Anwendung

3 Busybox

4 Weitere Tools

# RootFS Möglichkeiten

- Root-Filesystem im Flash (oder USB-Stick)
  - Flash lässt nur **endliche Schreibzyklen** zu!
- ReadOnly Device (z.B. CDROM)
  - Temporäre Verzeichnisse werden als tmpfs gemountet
  - Per Overlay-Filesysteme sind Modifikationen an den Dateien im laufenden Betrieb möglich.
- Linux bietet die Möglichkeit, das RootFS Image auch im RAM zu laden und dort zu mounten.
  - ‘Early-Userland’
  - entwickelt von Distributoren, um mit einem minimalen Linux das System zu konfigurieren
  - Early-Userland langt für Embedded Systems völlig aus

# Vorteile RootFS im RAM

- Konsistentes Filesystem bei jedem Neustart
- Booten ohne Filesystemcheck
- Einfache Systemaktualisierung
- Eventuell Auswahl des Rootfilesystems (Debug/Test)
- Schneller Zugriff
- Auf RAM sind unbegrenzte Schreibzyklen möglich

# Nachteile RootFS im RAM

- Modifikationen am Root-Filesystem gehen mit einem Neustart verloren, wenn diese nicht im Image durchgeführt werden
- Für die Ramdisk wird Hauptspeicher (RAM) benötigt (2–8 Mbyte)

# RootFS Ramdisk

- Auf dem Entwicklungsrechner wird eine Datei der Größe der späteren Ramdisk erzeugt.  
 $dd if=/dev/zero of=<NAME> count=<size>$
- RootFS wird beim Booten in die **Ramdisk** geladen und dort als ‘Early-Userland’ gemountet
  - Ein Teil des Hauptspeichers wird als eine **Ramdisk** reserviert
  - **Ramdisk** wird wie eine herkömmliche Disk gemountet
- Pflege des Rootfilesystems:
  - weniger aufwendig als die Pflege eines Desktopsystems.
  - zu beachten sind Logfiles!
    - sind Logfiles nötig?
    - Größe von Logfiles begrenzen
    - Logfiles löschen

# Ramdisk im Kernel Konfigurieren

- Unter ‘Block Devices’ den Punkt ‘Ram Disk Support’ (CONFIG\_BLK\_DEV\_RAM) auswählen
- Nach dem Booten stehen die Gerätedateien */dev/ram0*, */dev/ram1* und so weiter zur Verfügung
- Diese ‘Geräte’ werden wie herkömmliche Discs behandelt
  - \$ *mke2fs /dev/ram0*
  - \$ *mount /dev/ram0 /mnt*

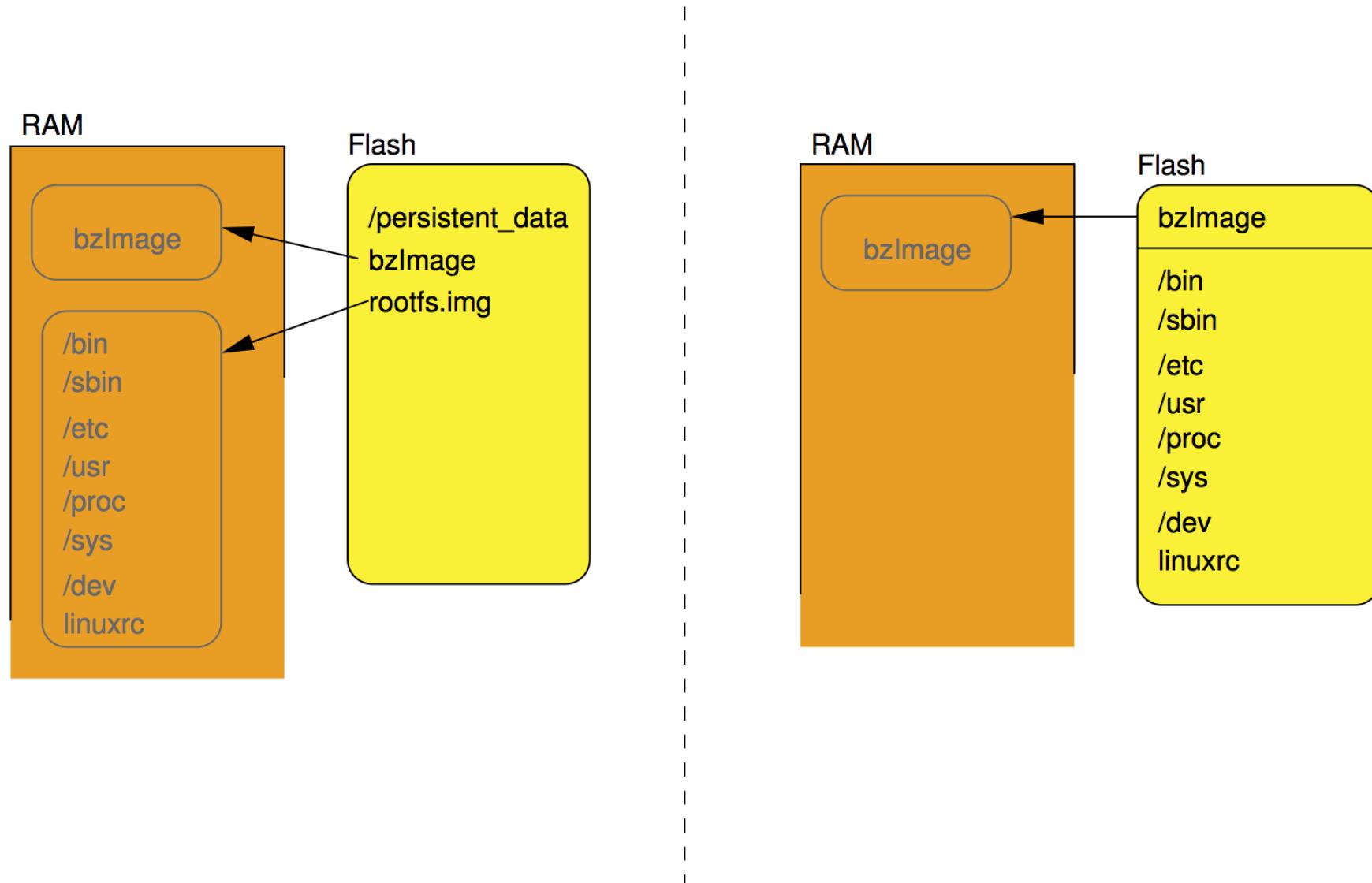
# 'initrd' Support

- Mit 'initrd' Support packt der Kernel das komprimierte Root-FS Image aus und
- kopiert den Inhalt in eine RAM Disk.
- Der Kernel mounted die RAM Disk als Root
  - Kernel Parameter 'initrd='
  - startet das Skript 'linuxrc' auf dem Root Filesystem.

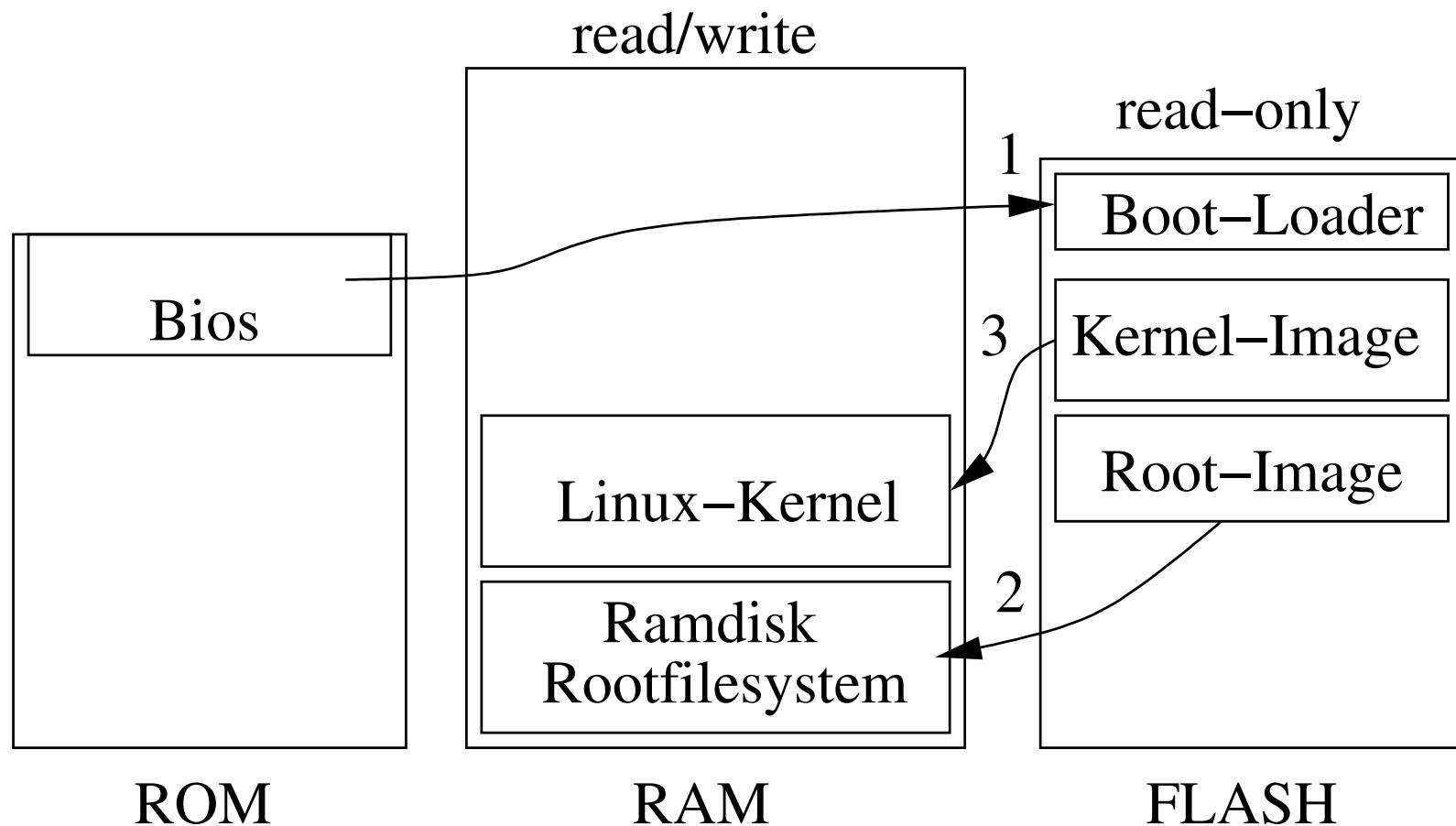
# Kernel Bootablauf Early-Userland

- ① Bios
- ② Bootloader
- ③ Kernel mit RootFS im Early-Userland
- ④ /init (aus RootFS Early-Userland)
  - RootFS Early-Userland enthält evtl. Treiber plus Konfigurationen
- ⑤ Im Fall von Desktop/Server
  - Newmount der Ziel-Root-Partition
  - Starten von **/sbin/init** (per exec!)

# ramfs vs Flash



# Bootablauf mit 2 Images und Ramdisk



# Erklärung Bootablauf

- Beim Booten wird zunächst das Root-Image z.B. vom Flash geladen und im Hauptspeicher abgelegt.
- Danach wird der Betriebssystemkern geladen und gestartet.
- Beim weiteren Betrieb bleibt das Root-Image auf der Flash unangetastet

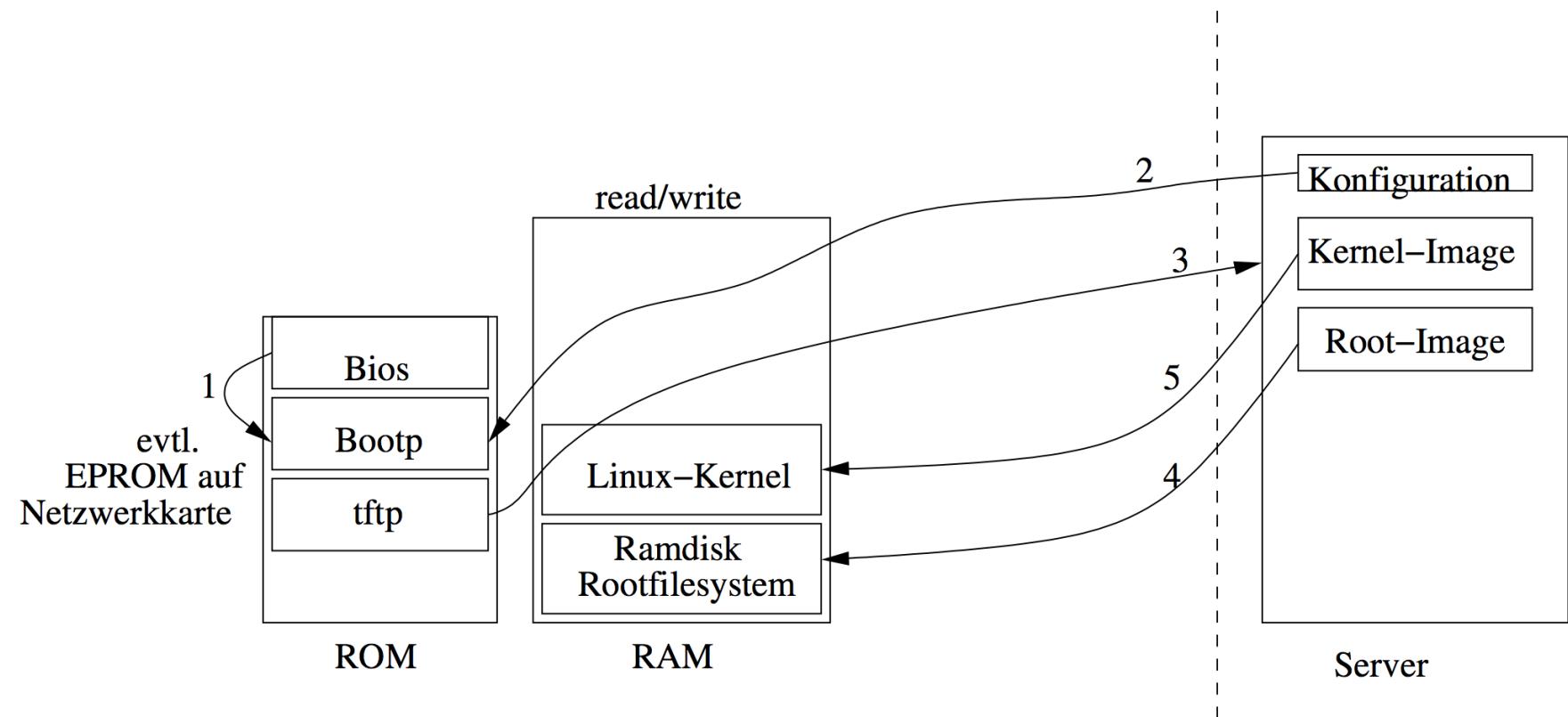
# Vorteile Ramdisk

- Der Zugriff auf die Ramdisk ist schneller als der Zugriff auf den Flashspeicher.
- Beim Flashspeicher ist die Anzahl der Schreibzugriffe limitiert.
- Die Größe der Ramdisk ist frei wählbar.
- Daten in Ramdisk ‘temporär’ -> System muss nicht runtergefahren werden.
- Das Verfahren vereinfacht die Systemaktualisierung.

# Nachteile Ramdisk

- Ramdisk verbraucht Hauptspeicher.
- Die Daten sind doppelt vorhanden:
  - im Image selbst auf z.B. der Flashdisk
  - in der Ramdisk
- Modifikationen, die während des Betriebs im Root-Filesystem vorgenommen werden, sind nach dem Ausschalten des Systems verloren.

# Booten über Netzwerk



# Erklärung Bootablauf über Netzwerk

- **Bootp** selbst wird verwendet, um wesentliche IP-Konfigurationsparameter automatisiert zu verteilen.
- Ein Client macht dazu einen Bootp-Request (Broadcast).
- Der Bootp-Server entscheidet anhand der MAC-Adresse des Clients, ob er den Request beantwortet oder nicht.
- Fühlt er sich für den Client verantwortlich, schickt er ihm die wichtigsten Netzwerkparameter wie IP-Adresse, Netzmase und Broadcastadresse.
- Darüberhinaus kann er dem Client auch die Namen von Systemdateien mitteilen, die sich der Client dann per tftp (eine einfache Variante des ftp-Protokolls) vom Server holt und als System startet.

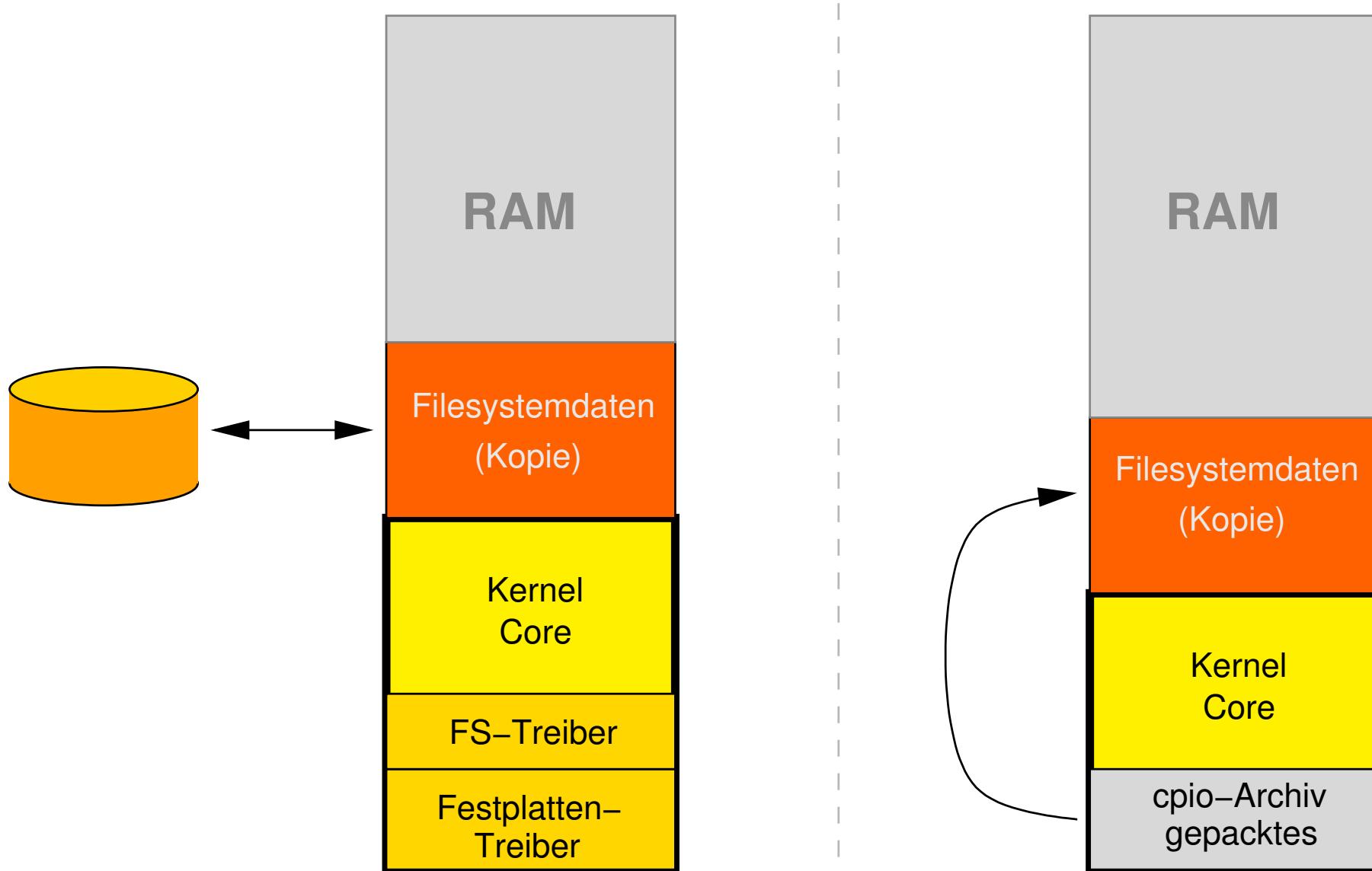
# VFS Abkürzung

- Ein Filesystemtreiber ist bei genauer Betrachtungsweise nichts anderes als eine Mappersoftware, die spezifische Dateisystemeigenschaften auf das intern verwendete 'virtual filesystem' umsetzt.
- Warum also erst umsetzen, wenn die Daten direkt in das VFS eingepflegt werden können?
- Dieser Überlegung folgend packt das Kernel-Build-System einfach die notwendigen Treiber, Gerätedateien und Bootprogramme in ein Archiv.
- Der Bootloader lädt
  - das Archiv als eigenständiges Image zusätzlich zum Kernel Image oder
  - Kernel direkt mit CPIO-Archiv gelinkt.
- Kernel Option hierfür: **initramfs**

# initramfs Support

- Dateien werden direkt in die kernelinternen Datenstrukturen des VFS umgesetzt.
  - das spart Code, Speicherplatz und Rechenzeit!
  - **initramfs** ist keine **ramdisk**!
- Es wird jeweils genau soviel Hauptspeicher verwendet, wie Daten im RootFS liegen.
- Achtung: keine Überwachung bezüglich der RootFS-Größe
- initramfs kann auch direkt ins Kernel-Image gelinkt (z.B. als cpio Archiv) werden.
  - Achtung: **rechtliche Situation**: eigene App in den Kernel gelinkt -> GPL

# Speicherbelegung ohne/mit initramfs

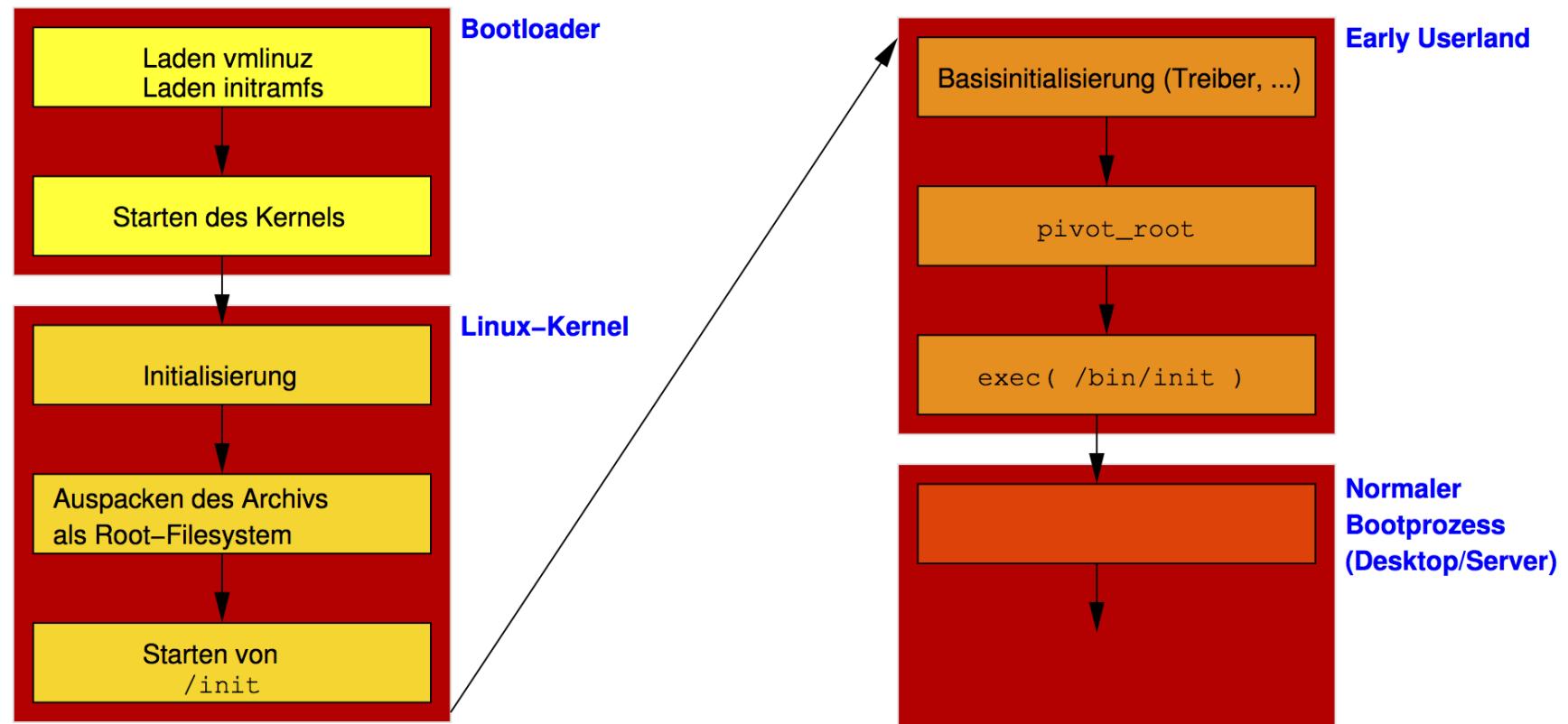


- Das **initramfs** Archiv ist im neuen CPIO-Format (Option: -H newc) abgespeichert.
- Zum vollständigen Auspacken müssen Sie Superuser sein, da CPIO Gerätedateien anlegt und die Besitzrechte anpasst.
- CPIO wertet normalerweise absolute Pfadangaben aus (Option –no-absolute-filenames).
  - Rufen Sie daher CPIO immer mit der Option ‘–no-absolute-filenames’ auf!
  - Ansonsten kann es schon mal passieren, dass Sie sich wesentliche Systemprogramme überschreiben und Ihr System damit unbrauchbar wird.

# Zugriff auf CPIO Archiv

```
$ mkdir /tmp/cpio  
$ cd /tmp/cpio  
$ zcat /usr/src/linux/usr/initramfs_data.cpio.gz | \  
sudo cpio -i -d -H newc -no-absolute-filenames
```

# Bootablauf mit CPIO Archiv



# pivot\_root()

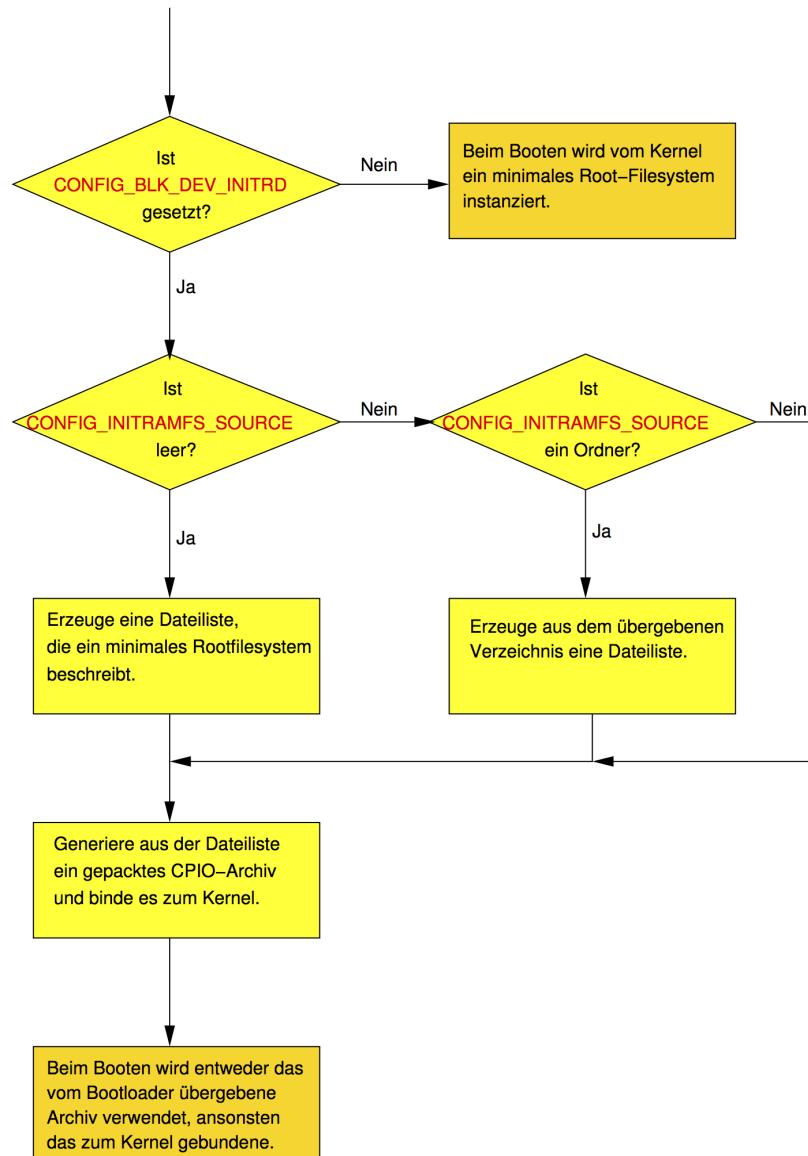
- Systemfunktion: \*pivot\_root(new\_root, put\_old)
  - *pivot\_root()* moves the root file system of the current process to the directory put\_old and makes new\_root the new root file system.
- Beispiel:

```
$ mount /dev/hda1 /new-root
$ cd /new-root
$ pivot_root . old-root
$ exec chroot . sh <dev/console>dev/console 2>&1
$ umount /old-root
```
- chroot startet Programm in neuer 'ROOT' Umgebung.

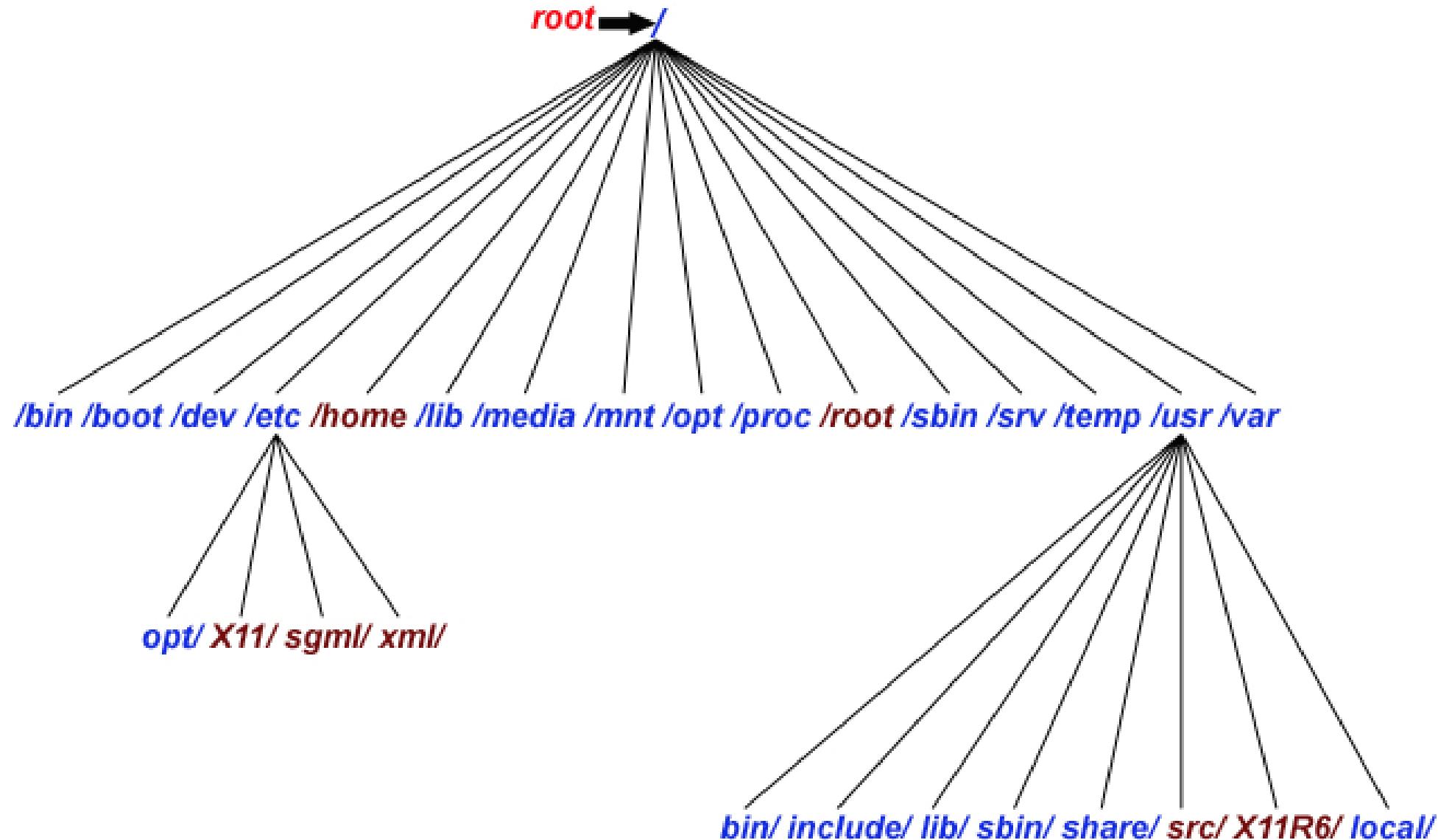
# Kernelbuild RootFS

- Ist CONFIG\_BLK\_DEV\_INITRD im Kernel aktiviert,
  - dann wird über CONFIG\_INITRAMFS\_SOURCE festgelegt,
    - ob das Standard-Minimal-Rootfilesystem CPIO-Archiv erzeugt wird oder
    - ob woher sonst die Daten für das CPIO-Archiv genommen werden.
- Wird die Variable mit einem Wert belegt,
  - so generiert das Kernel-Build-System **nicht** das Linux Standard-Minimal-Rootfilesystem
  - sondern generiert ein spezifisches Archiv.
  - Die Variable wird entweder mit
    - dem Namen eines Verzeichnisses oder
    - mit dem Namen einer Dateiliste (Textdatei) belegt.
- Kernel Own RootFS
  - Quelle: <http://www.landley.net/writing/rootfs-howto.html>

# Übersicht RAM Filesysteme



# Linux File System Hierachie Standard



# Standard Verzeichnisse

- */sbin*
  - Kommandos für die Systemverwaltung
- */sys*
  - Virtuelles Dateisystem
  - Informationen über Geräte und Treiber
- */tmp*
  - Temporäre Dateien, werden typische mit jedem Reboot gelöscht
  - oft als tmpfs Dateisystem realisiert
- */usr*
  - Anwendungsprogramme
  - */usr/src*: Linux Kernel Sourcen
- */var*
  - Logdateien, Spooldateien
  - temporäre Dateien (*/var/tmp*), PID Dateien

# Minimale RootFS Hierarchie

- Ablage der zum Betrieb notwendigen Daten
  - Systemprogramme
  - Konfiguration
  - Applikationsprogramme
- (Minimaler) Dateibaum
  - /, /usr, /etc, /lib, /bin
  - Temporäre Verzeichnisse: /tmp, /var
  - /dev

# Layout Minimales RootFS

- Static (z.B im FLASH):
  - */usr*
  - */lib*
  - */bin*
  - */dev*
  - */etc*
- Dynamic (z.B. im RAM):
  - */var*
  - */tmp*

# Gerätedatei

- Verzeichnis */dev* benötigt anfangs drei Gerätedateien:
  - */dev/console*
  - */dev/null*
  - */dev/zero*
- udevd
  - Automatisches Erzeugen der Gerätedateien
  - Verzeichnis */dev* wird automatisch durch den udevd befüllt (beim Laden der Treiber)
  - Alternativ (ohne udevd) werden die Dateien von Hand angelegt
  - Konfigurationen zu udev liegen im Verzeichnis */etc/udev.d/*

# init Prozess Allgemein

- Zentraler Angelpunkt der Initialisierung eines Embedded Linux Systems ist das Programm **init**.
- Je nach Konfiguration des RamFS wird per default gesucht:
  - nach **/init** bei initramfs Konfiguration,
  - nach **/linuxrc** bei initrd Konfiguration,
  - nach **/sbin/init** bei Standard Block-Geräten
- **init** ist dafür verantwortlich, dass die für den Betrieb des Embedded System notwendigen Programme in einer geeigneten Reihenfolge gestartet werden
- Damit **init** diese Aufgabe flexibel wahrnehmen kann, ruft es verschiedene Skripte auf.
  - typischerweise im Verzeichnis: **/etc/init.d**.

# init Konfiguration Allgemein

- Init kann ein Skript sein, meist ist es aber ein normales Programm, welches per Konfiguration den weiteren Bootprozess durchführt.
- Dreh- und Angelpunkt von **init** ist die Datei /etc/inittab. In dieser Datei ist konfiguriert:
  - welche Rechenprozesse beim Booten gestartet werden sollen,
  - welcher Rechenprozess im Fall einer Störung der Spannungsversorgung aktiv werden soll,
  - Wie das System auf Ereignisse, z.B. die Tastenkombination CTRL-ALT-DEL reagiert,
  - in welchem so genannten Runlevel das System hochfahren soll.
- Runlevel:
  - Runlevel 1 ist der so genannte Single-User-Mode.
  - Runlevel 2 ist der normale Multi-User-Mode

# init Modi Allgemein

- *respawn*: Terminierte der Prozess, wird er wieder gestartet
- *wait*: Prozess wird gestartet, init wartet auf das Prozessende
- *boot*: Prozess wird während des Bootvorgangs gestartet (nicht in einem spezifischen Runlevel)
- Weitere spezielle Ereignisse:
  - *powerwait/powerfail*: init startet prozess und ‘wartet’/‘wartet nicht’ auf das Ende des Prozesses
  - *powerfailnow*: Low Battery Event
  - *ctrlaltdel/kbrequest*: spezielle Events vom Keyboard

## Standard Linux *inittab*

```
id:1:initdefault:
rc::bootwait:/etc/rc
1:1:respawn:/etc/getty 9600 tty1
2:1:respawn:/etc/getty 9600 tty2
3:1:respawn:/etc/getty 9600 tty3
4:1:respawn:/etc/getty 9600 tty4
```

# Weitere Dokumentation

[fs/ramfs/\\*](#) Quellcode zu ramfs

[init/initramfs.c](#) Quellcode zum Auspacken des CPIO-Archivs

[scripts/gen\\_initramfs.sh](#) Skript zur Erzeugung der Archiv-Dateiliste

[usr/gen\\_cpio.c](#) Programm zum Erzeugen eines CPIO-Archivs aus  
einer Dateiliste

[Documentation/early-userspace](#) Erläuterungen zum  
Kernel-Verzeichnis 'usr/'

[Documentation/initrd.txt](#) Erläuterungen zur 'alten' Ramdisk  
(1999/2000)

[Documentation/filesystems/ramfs-rootfs-initramfs.txt](#)  
Erläuterungen zum ramfs

[Documentation/filesystems/tmpfs](#) Erklärung zu tmpfs und dessen  
Mount-Parameter

# Übersicht

1 RootFS

2 Anwendung

3 Busybox

4 Weitere Tools

# Anwendung Allgemein

- Es steht (meist) nur eine eingeschränkte Umgebung zur Verfügung.
- Eventuell kein Framework für DLLs
  - Programme statisch linken.
  - Multicall-Binaries einsetzen
- Erzeugt mit Hilfe von Cross-Entwicklungswerkzeugen.
  - Funktionsbestimmende Applikationen
  - Paketmanagement (ipkg, Beispiel Linksys, ...)

# Beispiel Webserver

- Ein *HTTP-Server* beispielsweise ist für viele eingebettete Systeme beinahe obligatorisch.
- Aus zwei Gründen wird auf einem eingebetteten System selten ein Standardserver eingesetzt:
  - ① Ein Standardserver benötigt zu viele Ressourcen, die auf dem eingebetteten System nicht zur Verfügung stehen.
  - ② Der HTTP-Server sollte harmonisch den Zugriff auf Objekte des Gerätes ermöglichen.
- Aber auch für eingebettete Systeme kann auf vorgefertigte HTTP-Server zurückgegriffen werden.
  - Spezielle Server sind für den Einsatz im eingebetteten System optimiert
  - Erweiterungsmöglichkeiten sind gegeben.

# Remote Zugriff

Neben dem HTTP-Server sind somit noch folgende Netz-Dienste interessant:

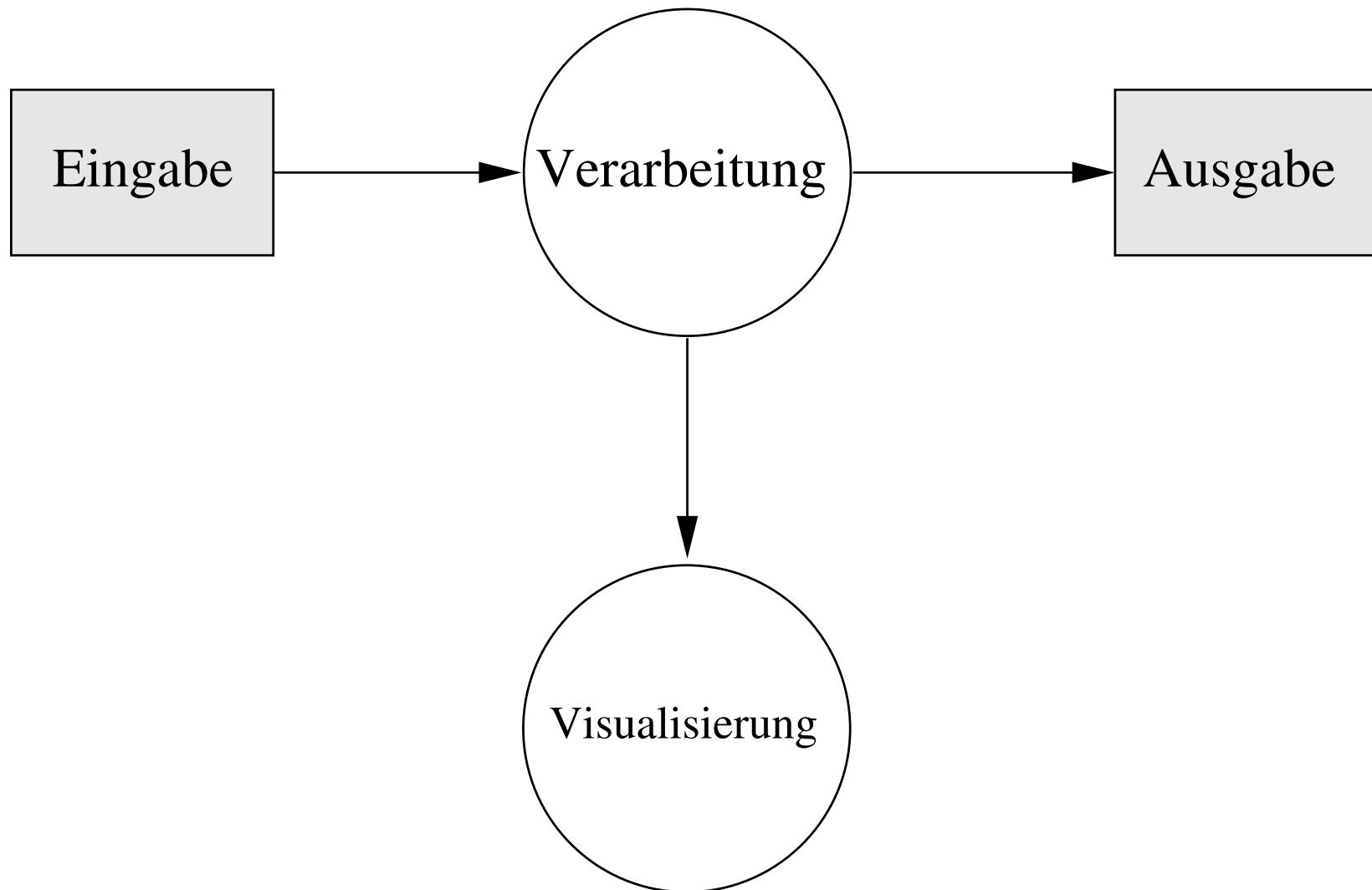
**FTP** FTP kann genutzt werden, um auf das Gerät Konfigurationsdateien und neue Systemversionen (Images) zu laden.

**SSH** Zum gleichen Zweck kann auch SSH eingesetzt werden. SSH ermöglicht ebenfalls die Übertragung von Daten, jedoch gesichert und verschlüsselt.

# Aspekte der Vernetzung

- ① Einzelne Teile einer Applikation lassen sich abtrennen und auf andere Rechner verlagern.
- ② Die Applikation ist per se schon als Komponente eines verteilten Systems geplant.
- ③ Durch die Vernetzung haben mehr Leute oder auch einfach andere Geräte Zugriff auf das Gerät. Entsprechend notwendig ist ein *sicherheitsgerichtetes Programmieren*.

# Grobstruktur einer Anwendung



# Aufteilung

**Einem Serverteil** Dieser Teil muß auf dem eigentlichen Gerät laufen und ist für die Eingabe und die Ausgabe verantwortlich

**Einem Clientteil** Nicht gerätekritische Applikationsteile können auch auf einen Clientrechner verlagert werden.

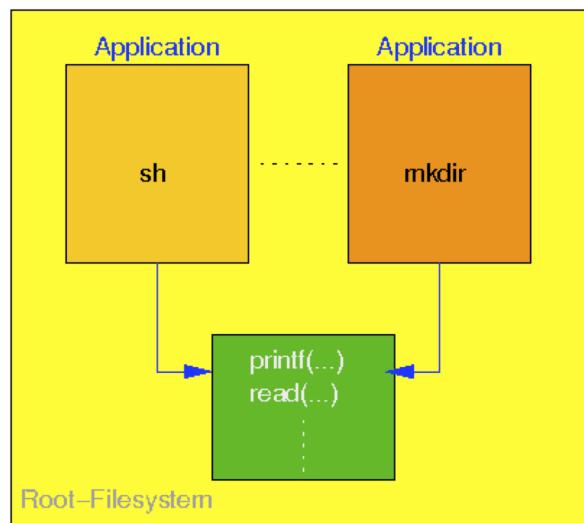
**Einer Visualisierung** Als besonderer nicht gerätekritischer Applikationsanteil kann die Visualisierung verlagert werden.

# Standardanwendungen

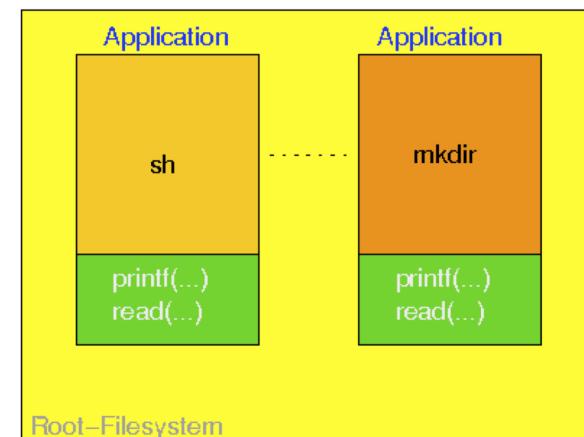
- Applikationen gibt es zwei Arten:
  - die eigentliche Applikation
  - System-Applikationen, die für den Betrieb, die Wartung und die Konfiguration notwendig sind.
- 'Festverdrahtung' der Applikationen
- Statische Bindung der Programme
- Multibinaries, z.B. busybox

# Anwendungs-Binaries

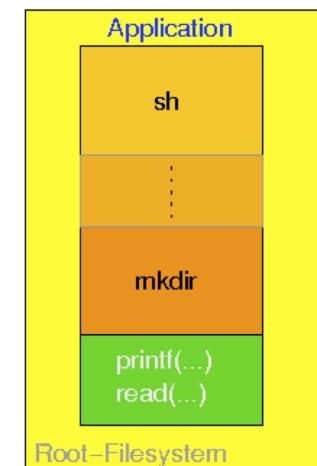
a) Shared Libs.



b) Statisch gebundene Bibliothek



c) Multi–Call Binary



# Bibliothek glibc

- Standard-Bibliothek
- Viele Funktionen
- Portabel
  - designed für Performance und
  - Unterstützung versch. (umfangreicher) Standards
- Hoher Ressourcenbedarf
  - NICHT auf Codegröße optimiert: Für Embedded Systeme sehr groß

# Bibliothek uclibc

- Download unter <http://www.uclibc.org>
- Auch für MMU-lose Linux-Systeme geeignet (uclinux)
- Umfang ca. 300kByte (für i386)
  - Optimiert auf Codegröße, ca. 4 Mal kleiner als Standard glibc
- unterstützt viele verschiedene Hardwareplattformen
- unterstützt die meisten glibc Funktionen,
  - aber nicht alle Standards wie unter glibc
  - Fokus liegt auf: C89, C99
- Lizenz: LGPL

# Bibliothek dietlibc

- <http://www.fefe.de/dietlibc>
- Sehr schlanke Bibliothek
- Lücken im utf- und Locale-Support
- GNU- und BSD-Erweiterungen (falls gewünscht)
- Umfang ca. 150 kByte (i386)
- Optimiert für statisches Linken
- Dynamisches Linken nur für x86 und ARM
- Einfache Entwicklung über Skript-Aufrufe

# Bibliothek klibc

- Download unter <ftp://ftp.kernel.org/pub/linux/libs/klibc>
- GIT-Archiv:  
<http://git.kernel.org/?p=libs/klibc/klibc.git;a=summary>
- für 32- und 64-Bit Systeme
- Einsatz beim „early userland“ (initramfs)
- Lauffähig auf vielen unterschiedlichen Plattformen
  - für einfache initramfs Binaries
    - klcc Compiler
    - geeignet für einfache Shell Skripte
      - Shell muss mit klibc erzeugt sein
  - **Nicht** thread-safe

# C++ (Standard Template Library)

- uclibc++
  - <http://www.gnu.org/software/libc/>
  - C++ Erweiterung der uclibc
- Embedded STL
  - [http://www.exactcode.de/site/open\\_source/embeddedstl/](http://www.exactcode.de/site/open_source/embeddedstl/)
  - Sehr frühes Entwicklungsstadium

# Grafische Oberflächen

- Hildon (Maemo)
- GTK+
- ...
- Android

# Systemebene

- Basisprogramme und Shell-Kommandos
  - Notwendig: ls, cp, mkdir, ln, rm, rmdir, chmod, chown
  - auf notwendige Funktionalität reduziert
- Remote-Login (Fernwartung)
- Editor
- Webserver
  - boa
  - lighthttp
- ftp
- syslog
  - Dateisystem darf nicht überlaufen!
  - logrotate
- Firmware-Update

# Übersicht

1 RootFS

2 Anwendung

3 Busybox

4 Weitere Tools

# Ziel

- Standard Programme wie z.B. **ls** für Embedded Anwendung oversized (Code + Features)
- Verschiedene ‘tiny’ Versionen von Standardanwendungen vorhanden
- **busybox** Projekt fasst viele Standard-Programme zusammen:
  - in einem Multibinary
  - über eine Konfiguration lässt sich die benötigten Funktionalität von **busybox** anpassen

# Multibinary

- Multicall-Binaries verwenden folgende Strategie:
  - Das Betriebssystem übergibt den Programmnamen als Argument '0'.
  - Über Hard- oder Symlinks kann nun ein Programm mehrere Namen erhalten
  - Je nachdem, mit welchem Namen das Programm aufgerufen wird, verhält es sich anders.
    - Programm 'ls', verhält es sich wie 'ls',
    - heißt es 'cat', verhält es sich wie 'cat'.
  - Im Basis-System gibt es somit nur 1 Binary (!) und viele Links, je nach gewünschten Funktionen

# Übersicht

- **busybox** Binary enthält die meisten UNIX Kommandos, selbst einen Web Server.
- Größe des Binaries ist < 500 KB (statisch gelinkt mit uClibc) oder < 1 MB (statisch gelinkt mit glibc).
- Features (z.B. Welche Kommandos ..) ist einfach zu konfigurieren.
- Bietet sich an, wenn:
  - initrd's mit komplexen Skripten gebildet werden müssen.
  - Für kleine Embedded Systems mit minimalen RAM/ROM Footprint.

# Beispiel busybox dynamisch gelinked

```
.  
|- bin  
  |- busybox  
  '- sh -> busybox  
  
|- dev  
  '- console  
  
|- etc  
  '- init.d  
  '- rcS  
  
'- lib  
  |- ld-2.3.2.so  
  |- ld-linux.so.2 -> ld-2.3.2.so  
  |- libc-2.3.2.so  
  |- libc.so.6 -> libc-2.3.2.so
```

# Konfiguration und Installation

- Aktuelle Version von <http://busybox.net>
- Konfiguration: \*\* make menuconfig \*\*
- **Kompilierung: make**
- Installation /Syncronisation Root-FS:
  - **make install**
  - Das rootfs Image des Embedded Systems mounten (z.B. */mnt/rootfs*)
  - **rsync -a \_install//mnt/rootfs/** (die /Zeichen am Ende sind unbedingt notwendig!)

# inittab der Busybox

```
#This is run first script :sysinit:/etc/init.d/rcS
    #Start an askfirst shell on the console
:askfirst:-/bin/sh

#Stuff to do when restarting the init process
:restart:/sbin/init

#Stuff to do before rebooting
:ctrlaltdel:/sbin/reboot

:shutdown:/bin/umount -a -r
```

# logread

- Ist Bestandteil von Busybox.
- Zeigt die Meldungen von syslogd an (using its circular buffer).
- Ersetzt */var/log/messages*. Reduziert Speicherplatz und Komplexität, da die Datei nicht 'rotiert' werden muss.
- Mit **logread -f** kann der Output gesichert werden.
- Andere Möglichkeit an die Log Ausgaben des Kernels zu kommen: das Kommando **dmesg**.

# Übersicht

1 RootFS

2 Anwendung

3 Busybox

4 Weitere Tools

# tinylogin

- The worlds smallest login/passwd/getty/etc
- Bietet sich an, wenn busybox nicht verwendet wird, da auch busybox diese Dienste anbietet.
- Sammlung von Unix Tools für:
  - Ein-/Ausloggen
  - Authentifizierung
  - Passwörter ändern
  - Handling von Usern und Gruppen. Unterstützt 'shadow password', um die Systemsicherheit zu erhöhen.

# Remote Login

- telnetd: Bestandteil von busybox
- ssh/sshd:
  - <http://matt.ucc.asn.au/dropbear/dropbear.html>
  - Sehr kleiner Speicher footprint für einen ssh
  - Bietet alle gängigen ssh/sshd
  - Nützlich um:
    - Eine Remote Konsole auf dem Target zu
    - Dateien zwischen Host und Target zu
    - rsync zwischen Host und Target auszuführen.