Hauptseminar DVP - Projekt

Erstellung einer bootfähigen CD / DVD für die Installation des DVP – Systems

SoSe 2004

Tino Schmidt
Tino.Schmidt@Stud.TU-Ilmenau.de

Inhaltsverzeichnis

1. Einleitung 2
2. Bootvorgang eines PC – Systems 2
2.1. Der LINUX – Bootvorgang 2
2.2. Das Programmpaket ISOLINUX 4
2.3. mkisofs & cdrecord 5
2.4. Der Bootvorgang von CD 5
3. Umsetzung der BootCD 6
4. Benutzung der BootCD 9
5. Probleme / Verbesserungsmöglichkeiten 10
6. Quellen 11

1. Einleitung

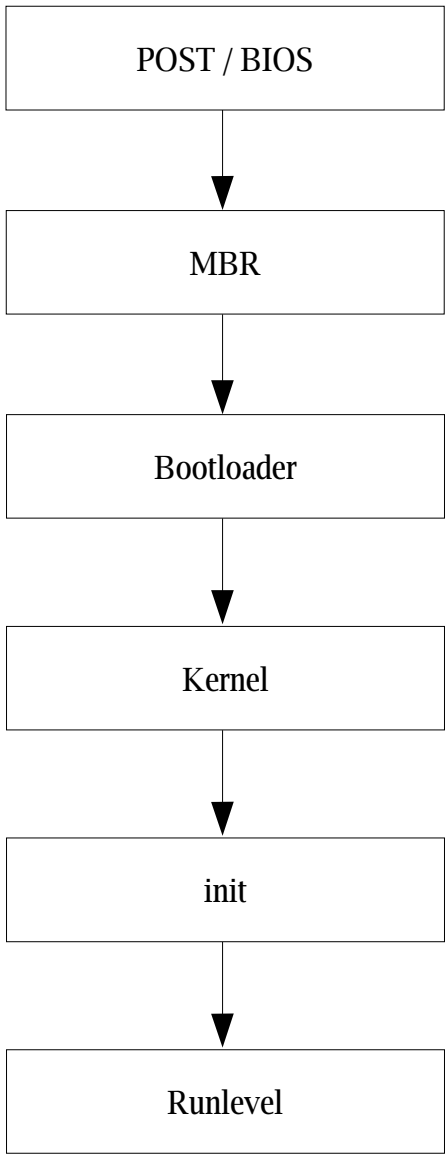
Dieses Hauptseminar beschreibt die Erstellung einer CD, die ein lauffähiges LINUX – System auf einem Rechner installiert, der als digitaler Videorekorder fungieren soll. Eigentliches Ziel hierbei ist es, eine genau für den jeweiligen Benutzer abgestimmte CD zu erstellen. Diese CD soll dann ein LINUX – System eigenständig installieren und einrichten, welches die gesetzten Anforderungen des Benutzers erfüllt.
Zuerst wird allerdings der Bootvorgang eines Computers an sich näher betrachtet, dann werden einige Werkzeuge vorgestellt.
Danach wird die Umsetzung der CD beschrieben und abschließend die eigentliche Erstellung der CD dokumentiert.

2. Booten eines PC – Systems

Um die Funktionsweise der CD zu verstehen, ist es erforderlich einen Überblick über die Vorgänge in einem Rechner zu bekommen, die während des Bootens ablaufen.
Da für dieses Hauptseminar nur LINUX als Betriebssystem relevant ist, wird der Bootvorgang daran detailliert erläutert.
Zuerst wird ein gewöhnlicher Bootvorgang beschrieben, dann wird auf den Bootvorgang von CD eingegangen und die Unterschiede dargestellt.
Weiterer Bestandteil dieses Kapitels sind die Dokumentierung einiger Programme, die für die Erstellung der CD notwendig sind.

2.1 Der LINUX – Bootvorgang

Zunächst soll ein grober Überbick zum Bootvorgang geschaffen werden. Das folgende Bild beschreibt die elementaren Vorgänge des Bootvorgangs, die im folgenden Kapitel genauer beschrieben werden:



Nach dem Einschalten des Rechners wird zunächst der **POST**, der Power On Self Test, ausgeführt. Der POST besteht aus 2 Teilen, zuerst wird die zentrale Hardware des Systems initialisiert (wie z.B. CPU, DMA- und Interrupt – Controller, BIOS – ROM und die ersten 64 KB des RAM). Danach, wenn der erste Schritt erfolgreich war, wird ein weiterer Teil der Hardware initiiert, die für das weitere Booten notwendig ist. Jetzt wird auch nach peripheren ROM – Erweiterungen gesucht, wie z.B. dem Video – ROM. Danach werden eventuelle SCSI – Systeme, die über ein eigenes BIOS verfügen, bearbeitet.

Sind nun alle Tests erfolgreich beendet worden, sucht das BIOS (Basic Input Output System) nach gültigen Bootsequenzen auf den (im BIOS vermerkten) Bootgeräten. Das können z.B. Festplatten, Disketten oder CD-ROMs sein, auf denen eine Bootsequenz in der Reihenfolge gesucht wird, wie sie im BIOS angegeben werden kann.

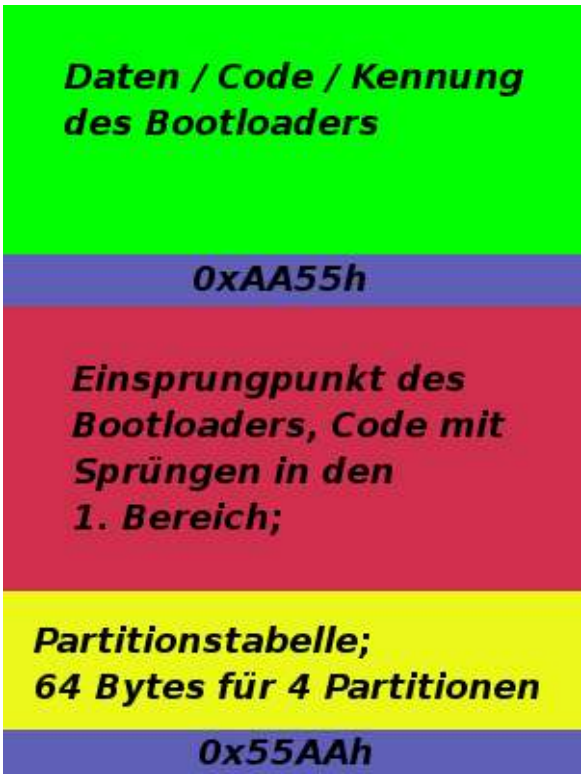


Bild: Aufbau eines MBR

Eine gültige Bootsequenz beginnt stets mit den 2 Bytes „**0xAA55h**“, diese Bootsequenz wird im MBR (Master Boot Record) des ersten Bootmediums gesucht, sie ist aber nicht an eine bestimmte Stelle gebunden! Der MBR enthält neben der Partitionstabelle noch ein kleines Programm, das die Daten der Partitionstabelle verarbeitet und in eine mit „bootable“ markierte Partition einspringt. Das Ende des MBR ist durch den Code „**0x55AAh**“ gekennzeichnet. Enthält der Bootsektor dieser Partition (der 1. Sektor) keine gültige Bootsequenz, wird eine Fehlermeldung ausgegeben, ansonsten wird mit dem MBR des nächsten Bootmediums fortgefahren.
Alternativ kann auch ein Bootloader im MBR liegen, der das weitere Booten des Rechners übernimmt. Dieser passt allerdings meist nicht komplett in den MBR, deswegen sind diese Programme meist 2 – stufig realisiert: Der erste Teil, der im MBR liegt, ist nur dafür da, den eigentlichen Loader auf der Partition zu laden und zu starten. Diese Bootloader haben teilweise vielfältige Möglichkeiten: Sie können Passwortabfragen implementieren, verschiedene Betriebssystem starten oder auch Parameter an das startende Betriebssystem übergeben. In diesem Beispiel soll nur auf LILO (**LI**NUX **Lo**ader) eingegangen werden.

Sobald LILO geladen ist, kann nun ausgewählt werden, welcher Kernel (im Fall von LINUX) oder welches Betriebssystem gestartet werden soll, hier wird wiederum nur der weitere Verlauf beim Starten eines LINUX – Kernels betrachtet. Wurde ein Kernel ausgewählt übergibt LILO die Kontrolle des Rechners an diesen Kernel. Das Laden des Kernels ist etwas diffizil, da LILO keine Dateisysteme kennt, und deshalb nur auf physikalische Adressen der Festplatte zugreifen kann. Diese Adressen müssen LILO bekannt also beim Booten bereits bekannt sein!

Seitenbemerkung: GRUB, ein anderer Bootloader, kann mit Dateisystemen umgehen, deshalb unterscheidet sich der Bootvorgang mit GRUB leicht. Dies soll jedoch nicht Gegenstand der Ausarbeitung sein.

Zunächst bewegt sich die Tätigkeit des Kernels sehr nah an der Hardware. Zuerst werden elementare Parameter aus dem BIOS ermittelt. Dann wird die Speicherverwaltung initialisiert, der Interrupt – Controller und ein eventuell vorhandener Co-Prozessor werden aktiviert. Nun werden alle Kernelteile initialisiert, dazu zählen die virtuelle

Speicherverwaltung, die Interruptroutinen, der Taktgeber und der Scheduler.
Darauf folgend wird das Dateisystem gestartet, Kernelpuffer angelegt, Netzwerkschnittstellen und die Interprozesskommunikation initialisiert. Zu diesem Zeitpunkt gibt es aber nur ein Prozess, er hat die **PID** (process identifier) 0 und wird allgemein als **idle** – Prozess bezeichnet. Dieser initialisiert in einer Mehrprozessorumgebung die zusätzlichen Prozessoren, im Normalfall übernimmt er nur das Starten des Prozesses mit der **PID** 1, den **init** – Prozess. Sobald der **init** – Prozess läuft, sperrt er die Kernelfunktionen für alle anderen Prozesse, da er in den folgenden Schritten nicht unterbrochen werden darf. Was nun der **init** – Prozess genau initialisiert, hängt vom jeweilgen Kernel ab, üblich sind z.B. PCI – Geräte.
Darauf folgend werden die ersten Dämonen (Hintergrundprozesse mit Überwachungstätigkeit) gestartet, z.B. **bdflush** (Cachesynchronisation) und **kswapd** (Verwaltung des Swapspeichers).
Nun wird versucht das **root** – Dateisystem auf dem Bootmedium einzuhängen (man spricht auch im Deutschen oft vom „*mounten*“), dafür muss der entsprechende Dateisystemtyp im Kernel natürlich enthalten sein. Ist dieser Schritt abgeschlossen gibt der **init** – Prozess den Kernel wieder frei.
Abschließend wird das **init** – Programm auf dem Bootmedium gesucht, was wiederum systemspezifische Schritte ausführt (z.B. Initialisierung des Netzwerks, Serverdienste, usw.), kann dieses Programm nicht gefunden werden, wird versucht eine Reparaturkonsole für den Administrator zu öffnen.
Mit dem Laden des **init** – Programms kann nun Einfluss auf das Booten des Systems genommen werden. Fast alle LINUX – Distributionen verwenden das Startverfahren des *UNIX Systems V*, dabei wird jede Komponente, die initialisiert werden soll durch ein eigenes Skript beschrieben, die Reihenfolge der Abarbeitung kann selbstverständlich auch angegeben werden, um diverse Abhängigkeiten zu erfüllen. Wenn aus administrativen Gründen nur ein Teil der Skripte geladen werden soll, besteht die Möglichkeit das System in verschiedenen ***Runlevels*** zu starten, wo eine Teilmenge der Skripte definiert wird, welche dann entsprechend dem ***Runlevel*** geladen wird.
Sobald die Abarbeitung dieser Skripte beendet ist, wird üblicherweise das ***login*** – Programm gestartet, und man kann sich am System anmelden. Der Bootprozess endet im Normalfall mit diesem Vorgang.

2.2 Das Programmpaket ISOLINUX

ISOLINUX ist ein Bootloader für linuxbasierte **i386** – Architekturen, der den **ISO 9660/El Torito** – Standard benutzt. Dieser Standard definiert genau, wie eine Cd aufgebaut sein muss, um davon booten zu können, wo z.B. der ausführbare Code liegen muss usw.
Um mit **ISOLINUX** arbeiten zu können, müssen einige wenige Anforderungen erfüllt sein: Zunächst sollte ein funktionsfähiges Linuxsystem zur Verfügung stehen, mit dem eine Abbilddatei für eine CD erstellt und gebrannt werden kann. Es sollten jeweils aktuelle Versionen von **mkisofs** und **cdrecord** installiert sein. Eine genaue Erklärung dieser Programme und ihrer Anwendung folgt anschließend.
Jetzt sollte ein Verzeichnis angelegt werden, in welches alle benötigten Dateien kopiert werden und aus dem dann das CD – Image erstellt wird. Für **ISOLINUX** ist es wichtig, dass ein Unterverzeichnis „**isolinux**“ existiert.

2.3 mkisofs & cdrecord

Diese beiden Programme sind notwendig, um die bootfähige CD zu brennen: **mkisofs** erzeugt ein ISO-konformes Abbild aus einem festgelegten Verzeichnis. In diesem Verzeichnis müssen alle Daten, so wie sie auf dem root – Verzeichnis der CD sein sollen, abgelegt werden. Mit der Option „**-b**“ wird **mkisofs** mitgeteilt, dass es sich um eine bootfähige CD handelt, desweiteren muss noch die Option „**-no-emul-boot**“ angegeben werden, da sonst eine feste Größe des Bootimages festgelegt wäre. Die Option ist nicht für ältere BIOS – Versionen geeignet, da diese nur von Disketten oder Festplatten booten können und deshalb eine Emulation eines Floppys benötigen, sobald von keinem dieser beiden Standardmedien gebootet werden soll. Die „**-l -R -r**“ - Optionen sind essentiell für die Rock Ridge Extensions, die für erweiterte Dateiattribute und Dateinnamenkonventionen nötig sind. Der Hintergrund für die BootCD liegt darin, dass Softlinks in dem Image zulässig sein müssen, was ohne die Rock Ridge Extensions nicht möglich wäre.

Hierzu eine kurze Seitenbemerkung zu Softlinks (Verknüpfung): Softlinks sind zwingend notwendig, um z.B. Abwärtskompatibilität von Bibliotheken zu gewährleisten. Normalerweise implementieren spätere Versionen von Bibliotheken auch alle Funktionen/Methoden ihrer Vorgänger, deswegen greifen sie auf den Standardnamen der Bibliothek zu (z.B. **libc.so.6**). Diese Datei ist nun lediglich ein Softlink auf die aktuellste **libc** – Bibliothek, die in dem System vorhanden ist (z.B. **libc-2.3.3**). So werden alle Programme, die den Standardnamen der Bibliothek benutzen automatisch auf die aktuelleste Version dynamisch gebunden (engl.: *linking*). Das hat den Vorteil, dass eventuelle Fehler in älteren Bibliotheken durch einfaches Ersetzen einer neuen Bibliothek und Anpassen des Softlinks, gleich alle Programme davon profitieren. Sollte nun aber ein Programm eine genau bestimmte Version einer Bibliothek benötigen, so muss dieses Programm den Dateinamen direkt referenzieren, das sollte allerdings nur in Ausnahmefällen gemacht werden!

cdrecord ist ein Programm zum Brennen von Images, in unserem Fall, auf eine CD. Die notwendigen Optionen hier variieren von System zu System und sollen hier nicht weiter betrachtet werden.

2.4 Der Bootvorgang von CD

Bei dem Booten von CD gibt es einige Parallelitäten zum Bootvorgang, wie er in Kapitel 2.1 beschrieben ist. Der **POST** und die Suche nach einem gültigen Bootcode sind natürlich identisch. In unserem Fall wird nun auf der CD ein Bootcode gefunden, der ausgeführt wird. Ein Bootloader wird üblicherweise auf CDs nicht verwendet, da es hier einige Probleme zu lösen gäbe, wie z.B. die mangelnde Dateisystemunterstützung vom BIOS und das es keine direkten physikalische Sektoren auf CDs gibt, die adressierbar wären. Anwendungsfälle sind für einen solchen Loader in der Praxis auch weniger relevant.
Nun muss der Kernel, der sich auf der CD befindet, geladen werden. Hierfür ist das Programm **ISOLINUX** notwendig: Es übernimmt diese Aufgabe. Dann lädt **ISOLINUX** das komprimierte **ext2** – Dateisystem für unsere **initrd** (Initial Ramdisk) und legt diese in eine feste Speicheradresse ab.
Kurz zur Erklärung des Begriffs der **initrd**: Die initiale Ramdisk („Laufwerk im Speicher“) ist ein logisches Konstrukt im Speicher, auf dem ein Dateisystem

realisiert ist und das wie ein Laufwerk verwendet werden kann. Bei dem Booten von CD wird eine vorher definierte root – Umgebung zusammengestellt und diese wird dann in die initrd geladen und der Kernel nimmt diese Ramdisk als neue Wurzel des Dateisystems („**/**“).
Jetzt übernimmt der Kernel die Kontrolle über das System und dekomprimiert die, im Speicher abgelegte **initrd**, in eine Ramdisk (normalerweise „**/dev/ram0**“) und mountet diese dann als root – Benutzer. Danach versucht der Kernel die **/linuxrc** – Datei auszuführen, sofern sie in der Ramdisk vorhanden ist. Die Funktionalität des **linuxrc** – Programms kann sehr unterschiedlich sein, wenn es abgeschlossen ist, wird die Kontrolle des Systems wieder dem Kernel übergeben, um das Booten des Systems abzuschließen. Das **linuxrc** – Script unterbricht also den regulären Bootvorgang mit seiner Ausführung!

3. Umsetzung der Boot – CD

Für das Erstellen des 1. Prototyps der BootCD wurde auf ein gepacktes Archiv des Quellrechners zurückgegriffen, welches alle Dateien der gesamten Partition enthält. Somit enthält dieses Archiv alle notwendigen Daten.
Auf die CD mussten allerdings noch eine Reihe anderer Daten: Als wichtigstes natürlich das „**isolinux**“ - Unterverzeichnis, welches für das eigentliche Booten zuständig ist. Bei der Arbeit mit **ISOLINUX** sind einige Probleme aufgetaucht: Der ausführbare Code, der in den Bootbereich der CD geschrieben werden sollte musste zuerst mit einem Assembler – Compiler übersetzt werden, es hat einige Zeit gedauert einen Compiler zu finden, der dazu in der Lage war (**NASM**).
Dann musste die initiale Ramdisk kreiert werden, dazu wurden zahlreiche Verzeichnisse und Dateien angelegt, die in einem LINUX – System standardmäßig vorkommen.
Um die BootCD flexibel und anpassbar zu gestalten, war es erforderlich einige zusätzliche Dateien in die Ramdisk aufzunehmen. So mussten bestimmte shared libraries, wie **ld-linux.so.2** (Link auf die eigentliche Bibliothek **ld-2.3.3.so**), **libc.so.6** (Link auf die C-Bibliothek **libc-2.3.3**) mit in die Ramdisk, weil diese Bibliotheken von einer Vielzahl von Programmen benötigt werden. Außerdem wurden dann noch einige weitere Programme kopiert, unter anderem **bash**, **copydata**, **fdisk**, **gzip**, **ldd**, **mke2fs**, **mount** und **tar**. Außer **fdisk** (Partitionierung von Datenmedia) und **mke2fs** (Erstellung eines **ext2/3** Dateisystems) sind alle Programme für die Ramdisk unabdingbar.
Die **bash**, um als Script-Interpreter das Kopierscript (**copydata**) auszuführen, **ldd** für das dynamische Linken der shared libraries, **gzip** & **tar** für das Entpacken des Originalarchivs und **mount** für das Einbinden der CDROM und der Partition, auf der die Daten des Archivs kopiert werden sollen.
fdisk und **mke2fs** stellen nur eine nützliche Ergänzung dar und wurden zu Testzwecken benötigt.
Ein wichtiger Bestandteil der initialen Ramdisk ist auch noch das Programm **busybox**. **busybox** ist ein kompaktes, statisch kompiliertes Programm, was zahlreiche Standard – LINUX – Programme kapselt, wie z.B. die Aston-Shell, cat, chgrp, chown, chroot, echo, grep, dd, df, usw. Das eigentlich Besondere an **busybox** ist allerdings, dass sie mit einer speziellen libc (**dietlibc**) statisch kompiliert wurde, so dass Ihre Dateigröße sehr gering ist.
Als weiterer wichtiger Bestandteil muss natürlich auch ein Kernel auf die CD, der beim Booten gestartet wird. Verwendet wird die Kernelversion **2.4.20** mit einer Konfiguration, die eine größtmögliche Abdeckung aller Hardware bietet. Einige Komponenten können bei Bedarf modular eingebunden werden (z.B.

Netzwerktreiber), um den Kernel selbst nicht unnötig groß werden zu lassen.

```
.
|-- image
|   `-- isolinux
`-- initrd
    |-- bin
    |-- dev
    |-- etc
    |-- lib
    |   `-- modules
    |       `-- 2.4.20
    |           |-- kernel
    |           |   |-- drivers
    |           |   |   |-- block
    |           |   |   |-- char
    |           |   |   |-- i2c
    |           |   |   |-- media
    |           |   |   |   `-- video
    |           |   |   |-- net
    |           |   |   |   `-- e100
    |           |   |   |-- parport
    |           |   |   `-- sound
    |           |   |-- fs
    |           |   |   `-- udf
    |           |   `-- lib
    |           |       `-- zlib_deflate
    |           `-- pcmcia
    |-- mnt
    |-- mnt2
    |-- opt
    |-- proc
    |-- sbin
    |-- tmp
    |-- usr
    |   |-- bin
    |   `-- lib
    `-- var
        |-- adm
        |-- empty
        |-- lock
        |-- log
        |-- run
        |-- ssh_etc
        `-- tmp
```

Bild: Verzeichnisstruktur des Quellrechners

Da jetzt alle nötigen Bestandteile vorhanden sind, müssen sie noch in die vorbereitete Verzeichnisstruktur (s. Bild) kopiert werden, um das Erstellen des CD – Images vorzubereiten.
Dazu wurden 2 Shell-Skripte geschrieben, deren Inhalt hier kurz erläutert werden soll. Das erste Skript ist **erzeuge_16mb_Datei.sh** benannt, die einzelnen Schritte werden anhand des Listings erklärt:

erzeuge_16mb_datei:

```
if [ -f 16mbfs.gz ]; then
  rm -f 16mbfs.gz
fi
```

Diese Anweisung ist lediglich die Abfrage, ob aus einem vorherigen Aufruf des Skripts die Datei „**16mbfs.gz**“ noch im aktuellen Verzeichnis liegt. Ist dies so, wird sie gelöscht. Die Option „**-f**“ in der if – Bedingung überprüft, ob die angegebene Datei vorhanden ist und gibt in diesem Fall **true** zurück. „**-f**“ bei dem **rm** – Befehl bewirkt, dass keine Ausgabe erzeugt wird und keine Abfrage an den Benutzer gestellt wird.

```
dd if=/dev/zero of=16mbfs bs=1k count=16384
losetup /dev/loop0 16mbfs
mke2fs -m 0 /dev/loop0
[ ! -d /mnt2 ] && mkdir /mnt2
mount /dev/loop0 /mnt2
```

Das **dd** – Kommando liest vom Device **if=** , hier in unserem Fall das **zero** – Device (welches nur „leere“ Bytes erzeugt) und schreibt in das **of=** Device, hier eine normale Datei. Für die Blockgröße wurde **1KB** gewählt und es sollen **16384** Blöcke gelesen / geschrieben werden. Somit kommen wir auf eine leere, 16MB große Datei **16mbfs**.
Der **losetup** – Befehl initialisiert das **loopback** – Device **loop0** mit der Datei 16mbfs, welche eben erzeugt wurde. Somit steht uns jetzt eine Art „virtuelle“ Festplatte zur Verfügung, mit der wir arbeiten können. Dafür allerdings muss die **loopback** – Unterstützung im Kernel aktiviert sein!
Jetzt wird auf unserer „virtuellen“ Festplatte (**/dev/loop0**) mit **mke2fs** ein Dateisystem erzeugt, hier ein **ext2** – Filesystem. Die Option „**-m 0**“ sagt aus, dass kein Speicherplatz für den Superuser (root) reserviert ist. Standardmäßig ist dieser Wert auf 5 gesetzt, was bedeutet, dass 5% des Gesamtspeicherplatzes reserviert wären. Da das in unserer Anwendung

aber überflüssig ist, wurde die Option benutzt.
Jetzt wird geprüft ob das Verzeichnis „**/mnt2**“ vorhanden ist, wenn nicht, wird es angelegt.
Als letzter Schritt in diesem Abschnitt wird nun das Device gemountet, auf den Mountpoint „**/mnt2**“, den wir eben angelegt haben.

```
cd initrd
tar cf - . | (cd /mnt2 && tar xf - )
cd ..
```

Jetzt wechseln wir in das Verzeichnis, wo die Verzeichnisstruktur für unsere initiale Ramdisk liegt.
Der tar – Befehl hier erfordert etwas mehr Erklärung, da hier eigentlich 3 Kommandos simultan ablaufen. Der erste Teil des Befehls („**tar cvf - .**“) fasst einfach alle Dateien, Links und Verzeichnisse des aktuellen Verzeichnisses (unsere spätere initiale Ramdisk!) zusammen. Diese Daten werden jetzt nicht in eine Datei geschrieben sondern mittels des **pipe** – Befehls („**|**“) an die nächsten Befehle weitergeleitet. Die runden Klammern bedeuten jetzt, dass die Befehle darin in einer **Subshell** verarbeitet werden. Ein passender Vergleich aus der Programmierwelt wäre ein Funktionsaufruf. In der **Subshell** wird jetzt in das Verzeichnis „**/mnt2**“ gewechselt, welches wie vorhin gemountet haben und unserer „virtuellen“ Festplatte entspricht. Die Daten von dem ersten **tar** – Befehl werden jetzt hier, mittels des 2. **tar** – Befehls, extrahiert („**x**“ - Option).
Zusammenfassend wird einfach der komplette Inhalt vom aktuellen Verzeichnis auf unsere Festplatte kopiert, ein einfaches **copy** – Kommando wäre allerdings nicht möglich, da es da Probleme mit den Device – Nodes gibt, die im **dev** – Verzeichnis der initialen Ramdisk liegen. Ein weiteres Problem ist die Änderung der Rechteverteilung der Dateien bei dem **cp** – Befehl.

Der letzte Befehl (**cd ..**) wechselt zum Vaterverzeichnis.

```
umount /mnt2
losetup -d /dev/loop0
```

Jetzt wird unsere „virtuelle“ Festplatte wieder ausgehangen und das **loopback** – Device aufgelöst. Das heißt jedoch nicht, dass unsere Daten verloren sind! Diese wurden ja parallel zu den Operationen mit dem **loopback** – Device respektive der „virtuellen“ gemounteten Festplatte in der Datei **16mbfs** gemacht! Dort sind also alle Daten momentan gespeichert.

```
echo "Komprimiere Ramdisk, bitte warten"
gzip -c -9 16mbfs >image/isolinux/initrd.gz
rm -f 16mbfs
```

Der erste Befehl gibt einfach in der Konsole den String aus, der nur signalisieren soll, dass der Rechner noch arbeitet, da das Komprimieren längere Zeit in Anspruch nehmen kann und eventuell keine Anzeichen für Aktivität zu verzeichnen sind.
Der **gzip** – Befehl komprimiert nun die **16mbfs** – Datei und legt diese im **image/isolinux** – Verzeichnis (keine absolute Pfadangabe!, relativ zum aktuellen Pfad !) unter dem Namen **initrd.gz** ab. Der Name dieser Datei ist nicht frei wählbar, wie im vorigen Kapitel bei der Programmbeschreibung von **ISOLINUX** bereits angedeutet wurde.

```
ls -l image/isolinux/initrd.gz
```

Dieser Befehl gibt lediglich Daten über die initale Ramdisk-Datei aus, die soeben erzeugt wurde.

```
.
|-- image.tar.gz
`-- isolinux
    |-- initrd.gz
    |-- isolinux.bin
    |-- isolinux.cfg
    `-- vmlinuz
```

Bild: Verzeichnisstruktur des ./image – Verzeichnisses

Zusammenfassung: Wir haben nun ein Verzeichnis „image“, wovon das Abbild für die CD erstellt werden soll. In diesem „**image**“ - Verzeichnis sind ein „**isolinux**“ - Unterverzeichnis, in diesem wiederum einige Dateien, die **ISOLINUX** selbst benötigt, und die **initrd.gz** – Datei, in der unsere initiale Ramdisk komprimiert vorliegt. Wenn nun noch weitere Dateien auf die CD sollen, müssen sie jetzt in dieses „**image**“ - Verzeichnis kopiert werden, bevor das nächste Skript gestartet wird! In unserem Fall muss nun genau das mit dem Archiv des Quellrechners (image.tar.gz) gemacht werden, bevor das 2. Skript gestartet wird.
Nun das Listing des 2. Skripts:

erzeuge_iso_datei:

```
mkisofs -o bootcd.iso -b isolinux/isolinux.bin -c
isolinux/boot.cat -no-emul-boot -boot-load-size 4
-boot-info-table -l -R -r image
```

Dieser Befehlsblock besteht nur aus einer Anweisung, die Optionen für den **mkisofs** – Befehl wurden bereits in Kapitel 2.2 erläutert. Wie der Name schon andeutet, wird durch dieses Skript die iso – Datei erzeugt, welche dann gebrannt werden kann.
Dieser Schritt kann einige Zeit brauchen, da das Archiv des Quellrechners während der Testphase über 700 MB groß war.

Ist dieser Schritt abgeschlossen, muss die **bootcd.iso** – Datei lediglich noch gebrannt werden. Dafür liegt auch ein Skript vor, dass aber für den jeweiligen Einsatzrechner angepasst werden muss.

brenne_cd:

```
cdrecord -v dev=ATAPI:0,0,0 blank=fast &&
cdrecord -v dev=ATAPI:0,0,0 bootcd.iso
```

Die Anweisungsfolge besteht wieder aus 2 Befehlen, der 1. Befehl löscht auf dem ATAPI – Device 0,0,0 den eingelegten RW – Rohling, der 2. Befehl brennt die iso – Datei. Die **&&** sagen aus, dass der 2. Befehl nur ausgeführt wird, wenn der erste erfolgreich beendet wurde (ohne Fehler), das muss eventuell angepasst werden, genauso wie die **dev=** - Option, um den entsprechenden Brenner zu benutzen.
Soll z.B. auf einen normalen CD – Rohling geschrieben werden, wäre der erste Befehl hinfällig und würde zu einem Fehler führen, sodass der 2. Befehlsteil nicht mehr abgearbeitet wird.

4. Benutzung der BootCD

Nachdem der Zielrechner für das Booten von CD eingerichtet wurde und sich die gebrannte CD im Laufwerk befindet, kann der Rechner nun gebootet werden.
Nun startet **ISOLINUX** von der CD und bootet automatisch den Kernel „**vmlinuz**“. Das Verhalten von ISOLINUX beim Starten kann durch eine Konfigurationsdatei angepasst werden, das soll aber nicht Gegenstand dieser Dokumentation werden.
Nachdem der Kernel gestartet ist, muss man sich am System anmelden, als Benutzername ist „**root**“ zu wählen und das Passwort ist „**bootcd**“. Dann kann einfach das kleines Skript „**/starte**“ ausgeführt werden, welches dann das eigentliche Kopierskript „**copydata**“ startet.

copydata:

```
#!./bash
mount /dev/hda1 /mnt2
mount /dev/hdc /mnt
cd /mnt2
tar -xvvzf /mnt/image.tar.gz
```

Dieses Skript liegt im **/usr/bin** – Unterverzeichnis der initialen Ramdisk, da sich dort auch die ganzen anderen statisch kompilierten Tools befinden, die nicht zum eigentlichen Bootvorgang benötigt werden.
Die erste Zeile gibt lediglich an, dass als Skriptinterpreter das Programm „**bash**“ im aktuellen Pfad (**/usr/bin**) benutzt

werden soll.
Der nächste Befehl mountet nun die Festplatte, auf die das Archiv kopiert werden soll an den Mountpoint **/mnt2**, der sich auf unserer initialen Ramdisk befindet. Man darf sich jetzt nicht verwirren lassen: Auch wenn der Mountpoint sich auf der Ramdisk befindet, wird auf die angegebene Partition (hier **hda1**) kopiert, der Mountpoint ist lediglich eine Art Verknüpfung.
Der nächste Befehl mountet nun das Medium im CD-ROM Laufwerk (die BootCD), auf der sich das Quellarchiv befindet. Dieser Schritt ist notwendig, weil die BootCD nicht gemountet wird, auch wenn von ihr gebootet wurde.
Dieses Skript muss unter Umständen angepasst werden, was die Festplattenpartition und das CD-ROM – Laufwerk betrifft, da die Devices auch an anderer Stelle sich befinden können.
Der nächste Befehl (**cd /mnt2**) wechselt nun einfach in das Verzeichnis, in das die Festplattenpartition gemountet wurde.
Der letzte Befehl entpackt nun das Quellarchiv direkt von der CD in das aktuelle Verzeichnis (Festplattenpartition). Die Option „**vv**“ sorgt hierbei für eine ausführliche Ausgabe auf der Konsole, sie kann bei Bedarf weggelassen werden.

Wenn der Entpack- & Kopiervorgang abgeschlossen ist, der Rechner neu gestartet werden, dazu kann das Skript „**/beende**“ verwendet werden.
Dann kann der Rechner neu gebootet werden, um das neu installierte System zu testen.

5. Probleme / Verbesserungs-möglichkeiten

Die BootCD war ursprünglich dazu gedacht gewesen auf einem neuen Rechner installiert zu werden, dazu ist die vorliegende BootCD noch nicht fähig.
Das Problem hierbei ist, dass im Bootsektor des Bootmediums geschrieben werden müsste, da dies aber sehr heikel ist, muss hier mit größter Vorsicht gearbeitet werden. Selbst die BootCDs von bekannten Distributionen lösen diesen Weg meist nicht direkt, sondern überlassen die Einrichtung des Bootmanagers dem Benutzer, wenn auch mit Unterstützung von einigen Tools. Für die klar definierten Anforderungen des Video – Systems ist eine spezielle Lösung aber möglich.
Wenn der Rechner bereits vorher mit LINUX installiert war (auf der selben Partition, wie im **copydata** – Skript angegeben !), dann ist das leider immer noch keine Garantie, dass das System danach bootet. Hierbei müssen aber 2 Fälle unterschieden werden:

1. Das System benutzt **GRUB** als Bootloader, dies ist der unkomplizierte Fall, hierbei sollten keine weiteren Änderungen nötig sein.
2. Es wird **LILO** als Bootloader verwendet. Da **LILO** beim Booten auf **raw** – Adressen der Festplatte zugreift und beim Kopieren nicht sichergestellt werden kann, dass der Kernel, bzw. die für LILO wichtigen **map** – Dateien, auf dieselben physischen Adressen kopiert werden (die Wahrscheinlichkeit ist sehr gering), muss „**lilo**“ gestartet werden um die Adressen zu aktualisieren. Dafür steht ein eigenständigen Skript „**inst_lilo**“ bereit. Dieses Skript muss zwischen den beiden Hauptskripten „**/starte**“ und „**/beende**“ ausgeführt werden !

inst_lilo:

```
#!/sbin/ash
echo „Aktualisiere LILO“
echo „“
chroot /mnt2 lilo
```

Das Problem der Aktualisierung der

LILO – Dateien besteht nun darin, dass sich alle LILO – Dateien auf der Festplattenpartition befinden, die in „**/mnt2**“ gemountet sind, unsere root – Umgebung aber die initiale Ramdisk ist. Würden wir einfach „**lilo**“ ausführen, würde nun der Befehlsinterpreter „**lilo**“ auf unserer Ramdisk suchen. Wenn wir nun ein „**lilo**“ auf unserer Ramdisk hätten, würden aber die falschen Dateien aktualisiert werden, da wir uns in der falschen Umgebung befinden! Dafür kann aber nun der Befehl „**chroot**“ verwendet werden, der den Befehl „**lilo**“ startet, aber als root – Umgebung „**/mnt2**“ verwendet, wo die Festplatte gemountet ist.

Eine Verbesserung des BootCD – Projektes wäre ein Skript/Programm auf dem Quellrechner, welches eine selektive Auswahl der Daten realisiert, die gebrannnt werden sollen, um die CD nicht unnötig aufzublähen. Desweiteren hätte dies den Vorteil, CDs mit modularer bzw. eingeschränkter Funktionalität herzustellen.

Eine weitere Verbesserung würde es bedeuten auf **busybox** zu verzichten. Ob dies, in einem überschaubaren Zeitraum, zu realisieren geht, kann momentan noch nicht abgesehen werden. Der Nutzen davon muss auch erst noch abgewägt werden.

Diese Verbesserungen / Änderungen werden wahrscheinlich in einer Studienjahresarbeit nächstes Semester umgesetzt.

6. Quellen

- http://syslinux.zytor.com/
- http://www.geocities.com/potato.geo/bootlinuxcd.html
- http://www.linvdr.org/projects/linvdr/
- http://www.lemoncube.com/index.php?op=show&id=202
- http://www.linux-magazin.de/Artikel/ausgabe/2001/07/bootimages/bootimages.html
- http://www.bablokb.de/bblcd/
- http://www.kernel.org/pub/dist/superrescue/
- http://rescuecd.sourceforge.net/
- http://www.toms.net/rb/home.html
- man – Pages