# Study-Guide: Installation von Linux und Paketmanagement

Auch dieses Thema war in der letzten Version der Level-1 Zertifizierung im zweiten Teil gelegen. Es geht prinzipiell um die Fragen, die sich vor, während und nach der Installation von Linux stellen, sowohl was die Fragen von Partitionsaufteilungen betrifft, als auch Einrichtung von Bootmanagern, Verwaltung der Libraries und Paketverwaltung. Entwerfen einer Festplattenaufteilung Installation eines Bootmanagers Erstellen und Installieren von im Sourcecode vorliegenden Programmen Verwaltung von Shared Libraries Verwendung des Debian Paketmanagements Verwendung des Red Hat Package Managers (RPM)


| | |
|---|---|
| Seite: | -= LinuxLernSystem =- ( http://www.lpi-test.de ) |
| Kurs: | LPIC-1 [101] |
| Buch: | Study-Guide: Installation von Linux und Paketmanagement |
| Gedruckt von: | André Scholz |
| Datum: | Dienstag, 1 November 2005, 10:16 Uhr |

# Inhaltsverzeichnis

# 1.102 - Installation von Linux und Paketmanagement

Auch dieses Thema war in der letzten Version der Level-1 Zertifizierung im zweiten Teil gelegen. Es geht prinzipiell um die Fragen, die sich vor, während und nach der Installation von Linux stellen, sowohl was die Fragen von Partitionsaufteilungen betrifft, als auch Einrichtung von Bootmanagern, Verwaltung der Libraries und Paketverwaltung.

---

- Entwerfen einer Festplattenaufteilung
- Installation eines Bootmanagers
- Erstellen und Installieren von im Sourcecode vorliegenden Programmen
- Verwaltung von Shared Libraries
- Verwendung des Debian Paketmanagements
- Verwendung des Red Hat Package Managers (RPM)

# 1.102.1 - Entwerfen einer Festplattenaufteilung

**Beschreibung:** Prüfungskandidaten sollten in der Lage sein, ein Partitionsschema für ein Linux-System zu entwerfen. Dieses Lernziel beinhaltet das Erzeugen von Dateisystemen und Swap-Bereichen auf separaten Partitionen oder Festplatten und das Maßschneidern des Systems für die beabsichtigte Verwendung des Systems. Ebenfalls enthalten ist das Platzieren von `/boot` auf einer Partition, die den BIOS-Anforderungen für den Systemstart genügt.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- `/ (root)` Dateisystem
- `/var` Dateisystem
- `/home` Dateisystem
- Swap-Bereiche
- Mount-Points
- Partitionen
- Zylinder 1024

---

Linux arbeitet - im Gegensatz zu anderen Systemen - nicht mit Laufwerksbuchstaben, sondern hängt alle zu benutzenden Partitionen in einen Dateibaum ein. Diese Technik des Mountens wird im Abschnitt 1.104 - Gerätedateien, Linux Dateisysteme, Filesystem Hierarchy Standard noch genauer behandelt. Hier geht es zunächst einmal um die grundlegenden Aufteilungen der Partitionen.

Es gibt verschiedene Gründe für die Benutzung von mehreren Partitionen unter Linux. Die wichtigsten sind:

- Mehrere physikalische Festplatten werden benutzt.
- Backups werden leichter planbar, wenn klar ist, daß veränderbare Daten nur auf bestimmten Partitionen vorkommen können.
- Userquotas (bestimmte Einschränkungen, wieviel Platz pro User zur Verfügung steht) beziehen sich auf Partitionen.
- Daten, die sich grundsätzlich nicht verändern (statische Systembereiche) können - wenn sie auf einer eigenen Partition liegen - ReadOnly gemountet werden.
- Bootmanager können manchmal nur auf Partitionen unterhalb des 1024 Zylinders zugreifen.
- Bestimmte Teile des Systems sollen im Netz auch anderen Rechnern zur Verfügung stehen, andere aber nicht.

Andererseits erfordert die Aufteilung eines Linux-Systems in verschiedene Partitionen auch eine wesentlich genauere Planung, weil schon bei der Installation festgelegt werden muß, wieviel Platz auf welchem Bereich benötigt werden wird.

Jede Partition (Dateisystem) wird in ein Verzeichnis eingehängt (gemountet). Der ursprüngliche Inhalt dieses Verzeichnisses ist solange unsichtbar, solange die Partition in dieses Verzeichnis eingehängt ist. Eine Liste der wichtigen Verzeichnisse auf der Wurzelpartition ist im Abschnitt 1.104.8 nachlesbar.

## Das Wurzeldateisystem

Das wichtigste Dateisystem ist das Wurzeldateisystem. Es ist die Partition, die beim Booten als erstes gemountet wird und von der aus alle weiteren Mount-Vorgänge erledigt werden. Jeder Bootmanager muß wissen, welche Partition des Systems die Wurzelpartition ist. Diese Partition enthält Verzeichnisse, die entweder Daten enthalten oder in die im weiteren Verlauf des Bootvorganges andere Partitionen gemountet werden.

Viele dieser Verzeichnisse können auf solche anderen Partitionen ausgelagert werden, andere dürfen unter keinen Umständen außerhalb der Wurzelpartition liegen. Diese Aufteilung ist nicht willkürlich, sondern hat ihre Gründe. Wie so oft, ist es besser diese Gründe zu verstehen, als einfach eine Liste von Verzeichnissen auswendig zu lernen. Daher folgt hier eine Liste all der Verzeichnisse, die niemals ausgelagert werden dürfen, zusammen mit der Begründung warum:

- **/bin**
  Das Verzeichnis `/bin` enthält die grundlegenden Systemprogramme, die zum Starten des Systems benötigt werden. Zum Beispiel das **mount**-Programm. Ohne dieses Programm können keine weiteren Dateisysteme gemountet werden, es ist also zwingend notwendig, daß `/bin` auf der Wurzelpartition liegt, wo es gleich nach dem Laden des Kernels zur Verfügung steht.
- **/dev**
  Das Verzeichnis `/dev` enthält die Gerätedateien, mit denen der Kernel physikalische Geräte (unter anderem auch Partitionen und Platten) ansprechen kann. Diese Gerätedateien werden benötigt, um Partitionen zu mounten. Also muß auch dieses Verzeichnis zwingend auf der Wurzelpartition liegen.
- **/etc**
  Das Verzeichnis `/etc` enthält - neben vielen anderen Informationen - die Information, wohin welche Partition gemountet werden soll. Ohne diese Information wäre es gar nicht möglich, andere Partitionen automatisch zu mounten.
- **/lib**
  Das Verzeichnis `/lib` enthält die grundlegenden Libraries, die von Programmen wie **mount** benötigt werden um zu funktionieren. Außerdem liegt in diesem Verzeichnis das Unterverzeichnis `modules`, das die Kernelmodule

enthält. Es können hier auch Kernelmodule liegen, die für die Erkennung von Dateisystemtypen notwendig sind.

- **/sbin**
  Das Verzeichnis `/sbin` enthält Programme, die während des Startvorganges notwendig sind, bevor gemountet wurde. So sind hier beispielsweise die Programme zur Überprüfung der Konsistenz von Dateisystemen enthalten. Diese Konsistenzüberprüfung muß vor dem Einhängen des jeweiligen Dateisystems stattfinden.

Diese Verzeichnisse müssen also zwingend auf der Wurzelpartition liegen, alle anderen Verzeichnisse können theoretisch auf andere Partitionen ausgelagert werden. Zumindestens das Verzeichnis `/root`, das Heimatverzeichnis des Systemverwalters sollte (muß aber nicht) noch auf dem Wurzelverzeichnis gelegen sein, da es vorkommen kann, daß der Systemverwalter in einem Wartungsmodus mit dem System arbeiten muß, in dem nur das Wurzelverzeichnis eingehängt ist. Wenn `/root` auf der Wurzelpartition liegt, ist so sichergestellt, daß er auch in diesem Wartungsmodus Zugriff auf seine Dateien hat.

## Das /usr- und das /var-Dateisystem

Alle statischen Daten des Betriebssystems, die nicht während des Startvorganges benötigt werden, liegen im `/usr`-Verzeichnis (usr bedeutet nicht User sondern Unix System Resources). Dieses Dateisystem ist in aller Regel auf einer separaten Partition untergebracht, die wesentlich größer als die Wurzelpartition ist. Dieses Verzeichnis kann (und soll) Read-Only gemountet werden, so daß keine Veränderungen darin im laufenden Betrieb möglich sind. Das schafft eine wesentlich höhere Stabilität des Gesamtsystems, weil nur durch ein bewußtes Remounten eine Veränderung am System möglich wird.

Entsprechend werden alle Dateien, die im laufenden Betrieb veränderbar sein müssen, in ein anderes Verzeichnis ausgelagert, das die sogenannten variablen Dateien enthält und aus diesem Grund den Namen `/var` trägt. Hier liegen mindestens die folgenden Unterverzeichnisse:

- **/var/lib**
  Ein Verzeichnis für modifizierbare Einstellungen, die traditionell unter `/usr/lib` liegen würden.
- **/var/lock**
  Hier liegen sogenannte Lock-Dateien. Das sind Dateien, die von Programmen angelegt werden, um zu zeigen, daß diese Programme laufen. Damit kann verhindert werden, daß bestimmte Programme mehrfach gestartet werden.
- **/var/log**
  Hier liegen die Logbücher des Systems, die permanent erweitert werden müssen.
- **/var/run**

Ähnlich wie /var/lock liegen hier Dateien, in die Programme ihre PID eintragen, damit diese auch ohne Aufruf von **ps** herausfindbar sind.

- **/var/spool**
- **/var/tmp**
  Temporäre Dateien.

Das Verzeichnis /var wird auch gerne und oft auf eine andere Partition als die Wurzelpartition ausgelagert. Es ist klar, daß in diesem Verzeichnis ständig Dateien angelegt, verändert und vergrößert werden. Die Gefahr, auf der Wurzelpartition keinen Platz mehr zu haben kann also speziell durch die Auslagerung von /var verringert werden.

## Das /home-Dateisystem

Im Verzeichnis /home liegen die Verzeichnisse der Benutzer/innen. Je nach Anspruch bzw. je nach Aufgabe des Rechners wird dieses Verzeichnis viel oder wenig Platz brauchen. Soll der Rechner z.B. nur ein Router ins Internet werden, oder ein alleinstehender Webserver, so benötigen wir für ihn praktisch keinen Speicherplatz für Benutzer, da es auf einem solchen Rechner keine Benutzer gibt. Ein Rechner hingegen, der als Fileserver dient oder eine normale Arbeitsstation ist, wird hier - je nach verfügbarem Plattenplatz - möglichst viel Speicher anbieten.

Im zweiten Fall ist es unbedingt ratsam, das /home Verzeichnis auf eine andere Platte auszulagern. Ein bösartiger Benutzer könnte ansonsten ein Programm schreiben, das die Festplatte füllt, und damit die Wurzelpartition bis obenhin voll machen, was dazu führen würde, daß Linux nicht mehr arbeiten kann. Es gibt dagegen zwar verschiedene Hilfsmechanismen wie Disk-Quota (siehe Abschnitt 1.104.4 - Verwalten von Diskquotas) oder der Reservierung von Plattenplatz auf EXT2-Dateisystemen, trotzdem ist eine Auslagerung auf eine andere Partition in einem solchen Fall immer ratsam.

Ein weiterer Aspekt der Auslagerung sind die Backups. Das /home Verzeichnis sollte grundsätzlich immer in ein Backup aufgenommen werden, weil hier ja die Dateien der User liegen, die einer ständigen Veränderung unterworfen sind.

## Das /tmp-Verzeichnis

Dieses Verzeichnis ist - neben dem /home-Verzeichnis - das einzige auf dem System, in dem Normaluser Schreibrechte besitzen. Also ist es auch hier ratsam, dafür zu sorgen, daß - wie oben schon erwähnt - die User nicht die Systemplatte füllen können. Auch dieses Verzeichnis kann und soll auf eine andere Partition ausgelagert werden, wenn es ein System ist, von dem wir erwarten, daß dort User arbeiten. Auf statischen Servern wie reinen Internet-Routern kann dieses Verzeichnis aber problemlos auch auf der Wurzelplatte verbleiben.

## Das /boot Dateisystem und das Problem der 1024 Zylinder

Auf älteren Systemen gibt es ein Problem mit Festplatten, die mehr als 1024 Zylinder haben. Dieses Problem bezieht sich nicht auf den Zugriff auf die Platte durch Linux, sondern auf den Zugriff durch Bootmanagersoftware.

Wenn ein Bootmanager wie LILO beim Systemstart auf Festplatten zugreift, etwa um den entsprechenden Kernel zu laden, so muß er ohne Betriebssystem auf die Platte zugreifen. Er muß also mit BIOS-Routinen arbeiten, die eben das Problem haben können, nicht mehr als 1024 Zylinder direkt ansprechen zu können. Ist die Partition, die die Dateien des Bootmanagers enthält jetzt auf einer Partition, deren Grenzen überhalb der 1024 Zylinder liegen, dann kann der Bootvorgang nicht funktionieren.

Aus diesem Grund gibt es die Möglichkeit, eine sehr kleine Partition (ungefähr 20 MB) anzulegen, die am Anfang der Platte liegt, also unterhalb der 1024 Zylinder. Diese Partition wird ins Verzeichnis `/boot` gemountet und enthält alle Informationen, die der Bootmanager benötigt (Kerneldateien, Initrd-Images, `System.map, boot.b, chain.b`, ...).

Durch diesen Trick ist der Bootmanager während des Bootvorgangs in der Lage, auf seine Dateien zuzugreifen, ohne die 1024er Grenze zu überschreiten.

Moderne BIOSe und moderne Bootmanager umgehen dieses Problem über die Verwendung der LBA-Methode. Dann ist die Auslagerung nicht mehr nötig.

## Swap-Partitionen

Ein Linux-System kann, wenn der physikalische Arbeitsspeicher zur Neige geht, einen bestimmten Partitionstyp als Speicherersatz für den Notfall benutzen. Diese Technik des Auslagerns (Paging) ist eine der ältesten Eigenschaften von Linux und hat es gerade auf Privatrechnern mit wenig Speicher so beliebt gemacht.

Eine Partition, die als Speichererweiterung dienen soll, wird als Swap-Partition bezeichnet und benötigt zwei grundlegende Vorbereitungstechniken:

- Sie muß beim Anlegen mit **fdisk** bereits den Partitionstyp *Linux Swap* (Typ 82) bekommen.
- Sie bekommt zwar kein "echtes" Dateisystem, muß jedoch mit dem Programm **mkswap** vorbereitet werden.

Während des Bootens wird eine solche Partition mit Hilfe des Kommandos **swapon** aktiviert. Sie wird also nicht wirklich gemountet, bekommt jedoch auch einen Eintrag in `/etc/fstab` in der Art:

```
/dev/hdc2       none            swap    sw      0       0
```

Eine schwierige Frage ist die richtige Größe einer solchen Swap-Partition. Früher gab es die Daumenregel, daß diese Partition doppelt so groß wie der physikalische Arbeitsspeicher sein sollte. Das war in den Zeiten verständlich, in denen Arbeitsspeichergrößen im Bereich 16-32 MB üblich waren. Da konnte es vorkommen, daß ein Programm geladen werden sollte, daß alleine 13 MB erfordert hatte (etwa Netscape). Damit wäre dieses Programm zwar langsam, aber eben doch gelaufen.

Heute sind wesentlich größere Speichergrößen üblich, die diese Daumenregel ad absurdum führen. Wenn ein Rechner 256 MB Ram besitzt, so macht es wenig Sinn, seine Swap-Partition 512 MB groß zu erstellen. Denn eine tatsächliche Arbeit mit einem System, das 512 MB Speicher auslagert ist nicht mehr möglich. Arbeitsspeicher ist einfach sehr viel schneller als Plattenzugriffe...

Eine praxistaugliche Größe der Swap-Partition ist etwas um 64 MB bis 128 MB.

## Zusammenfassung

Es gibt keine wirklich allgemeingültige Aussage, wie die Partitionen unter Linux aufgeteilt werden sollen. Es ist zu sehr abhängig von der geplanten Verwendung des Systems. Ein kleiner Router stellt andere Anforderungen, als ein großer Fileserver, eine Arbeitsstation andere als ein Server. Prinzipiell sollten die oben gemachten Angaben über die verschiedenen auszulagernden Dateisysteme immer auf die jeweilige Verwendung des Rechners bezogen überdacht werden. Ein kleiner Router mit einer Festplatte mit weniger als 1024 Zylindern kann einfach eine Partition bekommen, die alles enthält, ein großer Server sollte eine durchdachte Aufteilung der Partitionen erhalten.

# 1.102.2 - Installation eines Bootmanagers

**Beschreibung:** Prüfungskandidaten sollten in der Lage sein, einen Bootmanager auszuwählen, zu installieren und zu konfigurieren. Dieses Lernziel beinhaltet das Bereitstellen alternativer und Sicherungsbootmöglichkeiten (z.B. mittels Bootdiskette).

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

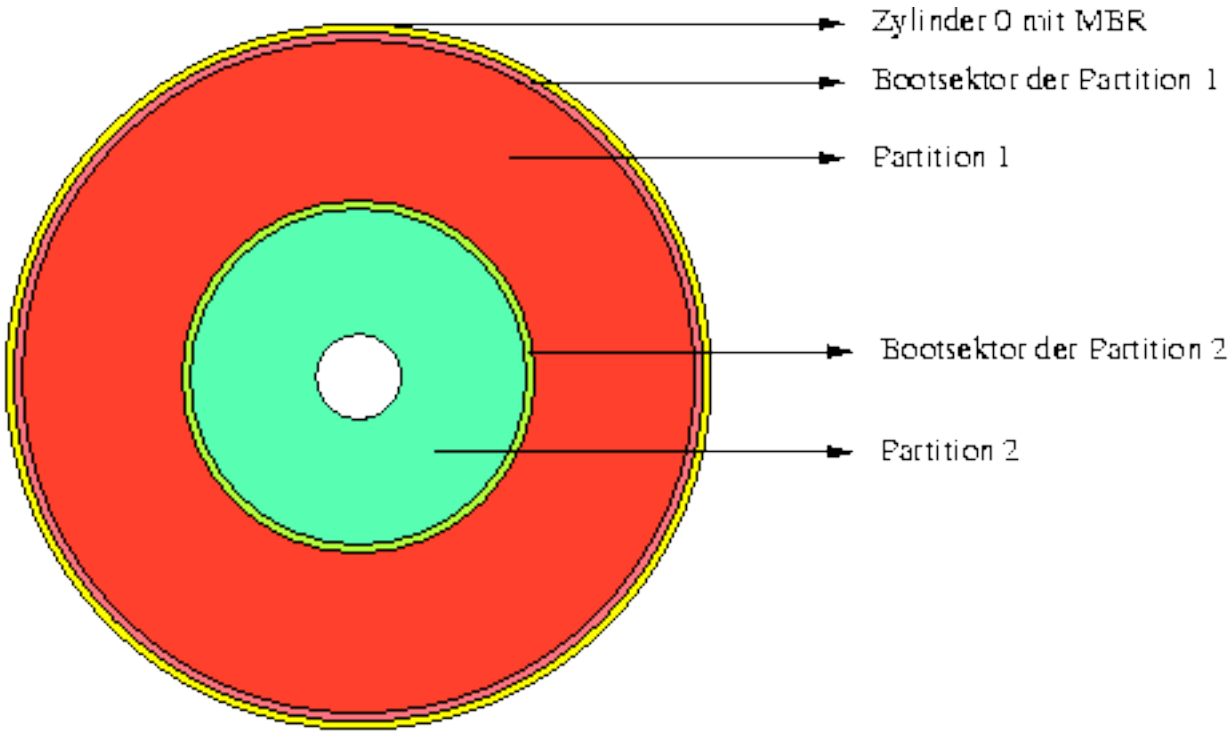
- `/etc/lilo.conf`
- `/boot/grub/grub.conf`
- **lilo**
- **grub-install**
- MBR
- Superblock
- Bootloader der ersten Stufe

---

## Prinzipielles zur Funktion von Bootmanagern

Auf der Hardware-Ebene von industriestandardkompatiblen Computern (und um die geht es hier) ist der Bootvorgang von Betriebssystemen klar vorgegeben. Das BIOS (Basic Input Output System) im ROM jedes Computers sucht auf der Platte, von der es booten soll/will nach dem äußersten Zylinder (Nr.0). Dieser Zylinder ist selbst nicht Teil von Partitionen, sondern er enthält z.B. die Partitionstabelle, die erstmal festlegt, wo denn überhaupt welche Partitionen beginnen und enden. Neben dieser Information befindet sich auf der Spur 0 jedes physikalischen Laufwerks der sogenannte **Master Boot Record** (MBR). Dieser MBR enthält gewöhnlich einfach einen Verweis auf die erste Partition der Platte und der dort abgelegten Bootinformation.

Jede Partition der Platte besitzt nämlich wiederum auf ihrem ersten Zylinder einen lokalen Bootsektor, der Informationen enthalten kann (bei DOS/Win-Systemen) wie das auf dieser Partition befindliche Betriebssystem gebootet werden soll. Diese Information besteht aus einem kleinen Programm, dem sogenannten Bootstrap-Loader.




Wenn auf einem System mehrere Betriebssysteme vorhanden sind, dann kann im Master Boot Record ein kleines Programm installiert werden, das den User entscheiden lässt, von welcher Partition welches System gebootet werden soll. Dieses Programm wird als **Bootmanager** bezeichnet.

Auf dem MBR ist aber nur sehr wenig Platz (weniger als 512 Byte). Das dort abgelegte Programm ist also zwangsläufig sehr klein. Meist wird das Programm nichts anderes tun, als ein zweites Programm zu laden, das das eigentliche Menü des Bootmanagers enthält. Dieses zweite Programm befindet sich zwangsläufig schon auf einer Festplattenpartition. Der Bootmanager muß also - unter Umgehung des Betriebssystems, das ja noch gar nicht geladen ist - Zugriff auf das Dateisystem dieser Partition haben und das gewünschte Programm dort laden. Dieser Vorgang wird als erste Stufe (Stage 1) des Bootloaders bezeichnet.

Das Programm, das das eigentliche Menü enthält, ist also die zweite Stufe (Stage 2) des Ladevorganges. Hier kann der/die BenutzerIn also aussuchen, welches Betriebssystem von welcher Partition gebootet werden soll. Erst nach dieser Auswahl wird das eigentliche System geladen (Stage 3).

Der klassische Bootmanager unter Linux war jahrelang das Programm **lilo** (Linux Loader). Neuerdings existiert eine zweite Möglichkeit, der GRand Unified Bootloader **grub**. Verschiedene Distributionen bedienen sich inzwischen dieser zweiten Methode, daher werden beide im weiteren Verlauf des Kapitels besprochen.

## Konfiguration von lilo

Wenn der Bootmanager **lilo** benutzt werden soll, so werden alle Konfigurationsinformationen in der Datei `/etc/lilo.conf` abgelegt. Das Programm

**lilo** liesst diese Datei und führt die folgenden Schritte aus:

- Aus den Einträgen der Konfigurationsdateien werden Menüeinträge erstellt, die in eine Datei (standardmäßig `/boot/boot.b`) abgelegt werden. Weitere Menüelemente können - je nach Konfiguration - in den Dateien `/boot/boot-menu.b`, `/boot/message` und `/boot/boot-bmp.b` abgelegt werden.
- Die Adressen und Größen der zu ladenden Kernel-Dateien werden in die Datei `/boot/map` geschrieben. Damit ist lilo später in der Lage, auf diese Dateien zuzugreifen, ohne ein Betriebssystem zu benutzen.
- Ein kleines Programm wird in den Master Boot Record geschrieben, das die Adressen der oben genannten Dateien in `/boot` enthält und die Aufgabe hat diese zu laden.

Das bedeutet, daß jedesmal, wenn eine Veränderung an den zu bootenden Kerneldateien gemacht wird, das Programm **lilo** neu aufgerufen werden muß. Selbst wenn alle Dateinamen gleich bleiben, so verändern sich eben doch die Adressen und Größen auf der Platte und die Informationen in `/boot/map` würden nicht mehr stimmen.

Normalerweise genügt der Aufruf von lilo ohne weitere Parameter.

Wird lilo mit der Option `-u` oder `-U` aufgerufen, so löscht es die Einträge auf dem entsprechenden Bootsektor und stellt ihn wieder so her, wie er war, bevor lilo installiert wurde. Damit kann eine Platte also wieder vom Bootmanager befreit werden.

Wenn beim Booten mit lilo ein Fehler auftritt, so kann dieser Fehler anhand des Bootprompts eingekreist werden. Der Bootprompt ist standardmäßig einfach das Wort:

```
LILO boot:
```

Das Wort LILO wird Zeichen für Zeichen auf den Schirm geschrieben. Falls es zu einem Fehler während des Bootmanager-Vorgangs kommt, so kann dieser Fehler anhand der Tatsache eingekreist werden, wieviele Buchstaben schon geschrieben wurden. Die folgenden Schritte werden jeweils durchgeführt, wenn ein Buchstabe geschrieben wurde:

***nichts***
: Kein Teil von Lilo wurde geladen. Entweder ist lilo nicht installiert oder er sitzt auf einem Bootsektor einer Partition, die nicht aktiviert ist.

**L*Fehlernummer***
: Der erste Teil des Bootloaders wurde lokalisiert und geladen, aber er kann den zweiten Teil nicht laden. Die zweistellige Fehlernummer gibt genauere Hinweise auf den Grund an. In der Regel deutet dieser Fehler auf ein Problem

mit der Plattenoberfläche oder falschen Plattenparametern im BIOS hin.

**LI**

Der erste Teil des Bootloaders hat den zweiten Teil geladen, kann ihn aber nicht ausführen. Das kann daran liegen, daß es entweder eine Inkompatibilität mit der Plattengeometrie gibt, oder die Datei /boot/boot.b wurde von der Stelle bewegt, an der sie lag, als lilo installiert (aufgerufen) wurde.

**LIL**

Der zweite Teil des Bootloaders wurde gestartet, kann aber die Beschreibungstabelle des map-files nicht laden. Typischerweise ein Medien-Fehler (Oberflächenbeschädigung) oder unpassende Plattengeometrie.

**LIL?**

Der zweite Teil des Bootloaders wurde an eine falsche Adresse geladen. Auch hier ist die wahrscheinlichste Ursache, daß die Datei /boot/boot.b verändert oder bewegt wurde.

**LIL-**

Die Beschreibngstabelle ist beschädigt. Entweder ein Oberflächenfehler, oder die Datei /boot/map wurde verändert oder bewegt.

**LILO**

Alle Teile von lilo wurden ordnungsgemäß geladen.

In jedem Fall ist bei einem Fehler immer der erste Versuch, lilo nochmal aufzurufen, nachdem das System über eine andere Technik (Bootdiskette, BootCD) gebootet wurde. War es der erste Versuch, der gescheitert ist, also hat lilo noch nie vorher funktioniert, so sollte davor aber nochmal die Datei /etc/lilo.conf überprüft werden.

## Die Datei /etc/lilo.conf

Die Datei /etc/lilo.conf enthält die Angaben, die lilo braucht um den Bootloader in einen Bootsektor zu schreiben. Sie besteht im Wesentlichen aus zwei Teilen. Der erste Teil beschreibt globale Einstellungen, der zweite beschreibt für jedes zu bootende System separate Einstellungen.

Ein typischer erster Teil einer lilo.conf-Datei könnte folgendermaßen aussehen:

```
append="reboot=warm"
boot=/dev/hda
lba32
message=/boot/message
prompt
timeout=300
```

Die Zeile `append=...` ermöglicht es, jedem zu bootenden Kernel bestimmte Parameter mitzugeben. Hier ist es der Parameter `reboot=warm`, der dazu führt, daß beim Reboot nicht ein Kaltstart (mit Speicherprüfung) sondern ein Warmstart ausgeführt wird.

Die Zeile `boot=...` gibt den Ort an, auf den lilo den Bootloader schreiben soll. Wird hier eine ganze physikalische Platte angegeben, also eine Gerätedatei ohne Partitionsnummer wie /dev/hda, so wird der Master-Boot-Record dieser Platte beschrieben. Stünde hier die Angabe einer Partition, also etwa /dev/hda2, so würde der Bootsektor der Partition beschrieben.

Die Angabe von `lba32` versetzt lilo in die Lage, auch mit Platten mit mehr als 1024 Zylindern umzugehen, wenn im BIOS der LBA-Modus eingestellt wurde.

Die Zeile `message=...` gibt an, aus welcher Datei die Bildschirmmeldung des Bootloaders stammen soll. Ohne diese Zeile wird keine Bildschirmmeldung benutzt. Der Inhalt der angegebenen Datei wird mit auf den Bootsektor geschrieben, wenn lilo aufgerufen wird. Sollten also Veränderungen vorgenommen werden sollen, so reicht es nicht, diese in die angegebene Datei zu schreiben. Es muß das Programm lilo nochmal aufgerufen werden, damit diese Veränderungen auch tatsächlich auf den Bootsektor kommen.

Die Angabe `prompt` veranlasst lilo, auf eine Benutzereingabe zu warten. Das ist insbesondere dann nötig, wenn mehr als ein Betriebssystem bootbar sein soll.

Die Zeile `timeout=...` gibt schließlich die Zehntelsekunden an, die auf Benutzereingaben gewartet werden soll, bevor das voreingestellte System gebootet wird. Die Angabe 300 steht also für 30 Sekunden.

Die Handbuchseite über lilo.conf(5) zeigt alle möglichen globalen Parameter auf.

Als zweiter Teil der Datei `/etc/lilo.conf` stehen die Einstellungen für die zu bootenden Systeme. Jede dieser Angaben beginnt mit einem `image=...` für Linux-Systeme oder einem `other=...` für alles andere und enthält dann noch mindestens die Angabe, auf welcher Partition sich die Wurzel des zu bootenden Systems befindet und welches label sie hat, mit dem lilo dann mitgeteilt werden kann, daß diese Partition gebootet werden soll. Ein Beispiel:

```
image = /boot/zImage
  root = /dev/hda2
  initrd = /boot/initrd
  label = linux

image = /boot/vmlinuz
  root = /dev/hda2
  label = linuxalt

other = /dev/hda1
  label = windows
```

alias = win

Der erste Block ist der voreingestellte. Dieses System wird also gebootet, wenn das Timeout erreicht ist oder LILO einfach mit der Enter-Taste beantwortet wird.

Dieser erste Block beschreibt also, daß der Kernel, der in /boot/zImage abgelegt wurde, gebootet werden soll. Das Wurzeldateisystem ist /dev/hda2. Die Zeile `initrd = /boot/initrd` bezeichnet die zu ladende initiale Ramdisk.

Der zweite Block beschreibt eine andere Kerneldatei (/boot/vmlinuz), die aber auch die Partition /dev/hda2 als Wurzelverzeichnis nützt. Um diesen Kernel zu booten muß auf dem Bootprompt das Wort "linuxalt" eingegeben werden.

Der dritte Block bezieht sich auf ein Nicht-Linux System (other). Statt der Kerneldatei wird hier einfach die Angabe der Partition gemacht, auf der dieses andere System liegt. Lilo kann ja davon ausgehen, daß jedes andere System auf dem Bootsektor seiner Partition seinen Bootstrap-Loader selbst abgespeichert hat. Das einzige, was wir noch an Information brauchen ist das Label, also das Wort, was wir auf dem Bootprompt angeben müssen, um dieses System zu laden. In unserem Beispiel ist es "windows". Weil das sehr lang ist, kann dazu noch ein Alias vergeben werden, in unserem Fall das Wort "win".

## Konfiguration von grub

Im Gegensatz zu **lilo** schreibt **grub** kein statisches Menü in eine der Dateien unter `/boot`. **grub** schreibt einfach nur einen Eintrag in den Master Boot Record der seinerseits wieder ein Programm unter `/boot/grub` läd. Dieses Programm ließt dann die Konfigurationsdatei `/boot/grub/grub.conf` und erstellt aus der darin gefundenen Information das Bootmenü.

Um mit **grub** zu arbeiten genügt es also einmal den Befehl

/sbin/grub-install /dev/hda

einzugeben, wobei die angegebene Gerätedatei `/dev/hda` den Master Boot Record der ersten IDE-Platte meint. Soll von einer SCSI-Platte gebootet werden (in einem System ganz ohne IDE-Platten), so muß diese Angabe entsprechend modifiziert werden. Diese Angabe bezieht sich **nicht** auf die Partitionen der zu bootenden Systeme sondern auf den Ort, wohin der Bootmanager geschrieben werden soll. Das BIOS der PCs sucht den Bootmanager immer zuerst auf dem Master Boot Record der ersten IDE-Platte, also ist `/dev/hda` in der Regel richtig.

Durch diese Architektur muß **grub** jetzt nicht mehr jedesmal aufgerufen werden, wenn sich an seiner Konfiguration etwas ändert. Da die zweite Stufe des

Bootmanagers selbst die Konfigurationsdatei liesst, entfällt diese Notwendigkeit. Das ist natürlich auch der Grund, warum die Konfigurationsdatei von **grub** nicht im Verzeichnis `/etc` liegt, sondern auf dem `/boot` Dateisystem.

## Die Datei /boot/grub/grub.conf

Diese Datei enthält die eigentliche Information über die zur Verfügung stehenden Bootmöglichkeiten. Damit geht **grub** einen anderen Weg als **lilo**. Diese Datei kann jederzeit auch im Nachhinein verändert werden und **grub** muß deshalb nicht erneut aufgerufen werden. Die Datei hat ein ähnliches Format wie die Konfigurationsdatei von **lilo**, aber sie unterscheidet sich im Detail erheblich.

Ein Beispiel für eine `grub.conf` Datei erfordert es, daß wir uns ein imaginäres Linux-System aufbauen. Nehmen wir folgendes Partitionsaufteilung:

| | |
|---|---|
| /dev/hda1 | /boot-Partition von Linux |
| /dev/hda2 | Windows98-Partition |
| /dev/hda3 | Wurzelpartition von Linux |

Eine einfache Konfigurationsdatei von **grub** könnte dann folgendermaßen aussehen:

```
default=0
timeout=10
splashimage=(hd0,0)/grub/splash.xpm.gz

title Linux (2.4.18)
    root (hd0,0)
    kernel /bzImage-2.4.18 ro root=/dev/hda3
    initrd /initrd-2.4.18.img

title Linuxalt (2.2.14)
    root (hd0,0)
    kernel /bzImage-2.2.14 ro root=/dev/hda3
    initrd /initrd-2.2.14.img

title Windows 98
    map (hd0,0) (hd0,1)
    map (hd0,1) (hd0,0)
    rootnoverify (hd0,1)
    chainloader +1
```

Die ersten drei Zeilen sind - ähnlich wie bei `lilo.conf` die globalen Einstellungen. `default=0` bedeutet, daß der erste Eintrag gewählt wird, wenn der Timeout überschritten wurde. Die zweite Zeile definiert diesen Timeout Wert in Sekunden. Die dritte Zeile definiert das verwendete Hintergrundbild. Hier finden wir bereits die erste

Anwendung der grub-spezifischen Adressierung, die gleich näher erläutert wird.

Die folgenden Einträge definieren jeweils die zu bootenden Systeme. Die einzelnen Zeilen bedeuten jeweils folgendes:

**title** *Titelzeile*
: Diese Zeile leitet einen Abschnitt eines zu bootenden Systems ein. Der unter *Titelzeile* gemachte Eintrag erscheint später im Menü.

**root** hd*n,m*
: Dieser Eintrag ist **nicht** das Wurzelverzeichnis des Linuxsystems. Er bezeichnet die Partition, auf der die Kerneldateien und die **grub**-Konfigurationsdateien liegen. In unserem Beispiel ist das `/dev/hda1`, also die `/boot`-Partition von Linux. Die angegebenen Zahlen bezeichnen folgendes:
  - *n* - Nummer der Platte (0 ist die erste Platte, 1 die zweite, ...)
  - *m* - Nummer der Partition auf der Platte (0 ist die erste Partition, 1 die zweite, ...)

  Unser Beispiel bezeichnet also mit `(hd0,0)` die erste Partition der ersten Platte also im Linux-Jagon `/dev/hda1`.

  Alle weiteren Angaben beziehen sich auf diese Platte.

**kernel** *Kerneldatei Parameter*
: Mit dieser Angabe wird die Kerneldatei angegeben und mit Aufrufparametern versehen. Der Pfad der Kerneldatei bezieht sich hier **nicht** auf die Wurzel des Linux-Systems sondern auf die Wurzel des Systems, das wir im letzten Punkt (root) angegeben hatten. In unserem Beispiel liegt die Kerneldatei also unter `/boot/bzImage-2.4.18`.

  Die eigentliche Wurzel des zu bootenden Linux-Systems wird als Kernelparameter angegeben (hier also `/dev/hda3`.

**initrd** *Ramdiskimage*
: Hier wird der Name der Ramdiskdatei angegeben, die als initiale Ramdisk geladen werden soll. Benutzt der Kernel keine initiale Ramdisk, so wird diese Zeile komplett weggelassen. Auch hier bezieht sich der Pfad auf die Wurzel des unter `root` angegebenen Devices.

Der zweite Eintrag definiert einen zweite Kernel auf der selben Partition. Die Angaben entsprechen denen des ersten Eintrags.

Im dritten Eintrag wird eine Windows-Partition definiert, die Win98 booten soll. Dieser Eintrag enthält einige Besonderheiten, die hier noch kurz dargestellt werden sollen.

**map** (*Partition1*) (*Partition2*)

Windows98 bootet normalerweise nur von der ersten Partition der ersten Festplatte. Durch den Befehl `map (hd0,0) (hd0,1)` erklären wir die Partition `hd0,0` zur zweiten Partition (`hd0,1`) und mit der nächsten Zeile wird entsprechend die zweite Partition zur ersten. Aus der Sicht des Windows-Systems ist also die zweite Partition jetzt die erste und umgekehrt. Damit ist für Windows die Welt wieder in Ordnung und es glaubt, von der ersten Partition zu booten.

**rootnoverify** (*Partition*)

Diese Angabe entspricht der Angabe `root` bei Linux-Einträgen, nur das diesesmal die entsprechende Partition nicht gemountet wird. (Das ist unter Windows nicht nötig, da ja nicht eine bestimmte Datei geladen werden soll, sondern der Bootsektor der Partition)

**chainloader +1**

Diese Angabe ist für Windows erforderlich. Windows wird über einen **chainloader** gebootet, der auf einem betimmten Sektor der Windows-Partition liegt. Die Angabe +1 bezeichnet hier die Sektornummer 1, also den Bootsektor der Windows-Partition (die unter `rootnoverify` angegeben wurde).

Es existieren natürlich noch viele weitere Möglichkeiten für die Konfigurationsdatei, aber die hier genannten sollten einen ausreichenden Überblick über den Einsatz von **grub** als Bootmanager geben.

## Sicherheitsbootmöglichkeiten

Wenn aus irgendeinem Grund der Master Boot Record zerstört wurde, dann wird weder **lilo** noch **grub** ein Bootmenü anbieten. In einem solchen Fall ist es notwendig, über eine alternative Möglichkeit booten zu können um dann mit den entsprechenden Befehlen **lilo** (`lilo`) oder **grub** (`grub-install`) erneut in den MBR zu installieren.

Praktisch jedes Installationsmedium von Linux (Boot-CDs, Bootdisketten) bietet die Möglichkeit, ein bereits installiertes Linux-System zu booten. Dazu muß nur eine Angabe bekannt sein, nämlich die Partition, die für dieses zu bootende System die Wurzelpartition ist. Das kann einfach dadurch geschehen, daß beim Booten des Installationsmediums der Kernelparameter

root=*Partition*

angegeben wird. Die Partition wird hier in einer für Linux verständlichen Form angegeben, also beispielsweise `/dev/hda3`. Damit wird dann der auf dem Installationsmedium befindliche Kernel gebootet, seine Wurzelpartition ist aber dann die angegebene Partition. Das heißt, wir können dann entsprechend auf dem System arbeiten, um den richtigen Bootmanager wieder zu installieren.

# 1.102.3 - Erstellen und Installieren von im Sourcecode vorliegenden Programmen

**Beschreibung:** Prüfungskandidaten sollten in der Lage sein, ein ausführbares Programm aus dem Quellcode zu erstellen und zu installieren. Dieses Lernziel beinhaltet die Fähigkeit, ein Source-Paket aus einem Archiv zu extrahieren. Kandidaten sollten in der Lage sein, einfache Anpassungen im Makefile vorzunehmen, wie z.B. Pfade zu ändern oder zusätzliche Verzeichnisse einzubinden.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **gunzip**
- **gzip**
- **bzip2**
- **tar**
- **configure**
- **make**

---

## Grundsätzlicher Vorgang

Ein Programm wird von seinem Programmierer (oder seinen Programmierern) in vielen einzelnen Quellcode-Dateien geschrieben. Diese Dateien enthalten den Quellcode in der Programmiersprache, die vom Programmierer benutzt wurde, also beispielsweise C.

Ein Compiler muß jetzt jede dieser einzelnen Dateien in sogenannte Objekt-Code Dateien compilieren. Diese Objekt-Dateien enthalten dann den Maschinencode, der vom Compiler erzeugt wurde, sind aber noch nicht selbst lauffähig. Für jede einzelne Quellcode-Datei wird eine entsprechende Objektdatei erzeugt. In der Regel tragen diese Dateien die Namensendung `.o`.

Erst im nächsten Schritt werden die verschiedenen erzeugten Objektdateien zusammen mit eventuellen weiteren Libraries zu einer ausführbaren Datei verbunden. Dieser Vorgang wird Linken (engl. to link - verbinden) genannt. Das Ergebnis dieses Vorgangs ist die ausführbare Datei, die dann verwendet werden kann.

Der Compilier- und Linkvorgang besteht also aus vielen Einzelschritten. Damit nicht bei jedem Compiliervorgang alle diese Einzelschritte angegeben werden müssen, gibt es das Programm **make**. Dieses Programm erhält seine Informationen welche

Dateien mit welchem Compiler wie übersetzt werden müssen und welche Objektdateien dann wie zu welcher ausführbaren Datei zusammengefügt werden sollen aus einer Datei mit Namen **Makefile**.

Ein Programmierer kann also neben seinem Programm ein dazu passendes Makefile erstellen, so daß alle Schritte, die für die Compilierung notwendig sind, vom Programm **make** ausgeführt werden können. Dazu erstellt er im Makefile bestimmte Regeln, die dann die entsprechenden Befehle enthalten, die notwendig sind um eine Regel auszuführen.

Der Name einer Regel wird dann dem Programm **make** als Parameter mitgegeben. Das Programm sucht dann das Makefile nach der entsprechenden Regel ab und führt die dort gefundenen Befehle aus.

**make** arbeitet außerdem mit den Zeitmarken der Dateien. Bei einem zweiten Compiliervorgang wird für jede Objektdatei überprüft, ob die Zeitmarke der dazu passenden Quellcode-Datei älter als die der Objektdatei ist. Nur wenn die Zeitmarke der Quellcodedatei jünger als die der Objektdatei ist, wird die Quellcodedatei erneut compiliert.

Mit Hilfe dieser Technik ist es auch für einen Nicht-Programmierer einfach möglich, komplexe Programme, die im Quellcode vorliegen, selbst zu übersetzen und zu installieren. Im Linux-Bereich haben sich dazu bestimmte Standards herausgebildet, die im weiteren Verlauf besprochen werden sollen.

## Format eines Quellcode-Paketes

Quellcode-Pakete werden im Linux-Bereich immer als Tar-Archive ausgeliefert. In der Regel sind diese Archive komprimiert, damit sie schneller übertragen werden können. Ein unkomprimiertes tar-Paket trägt die Endung `.tar`, ein mit **gzip** komprimiertes Paket hat entweder die Endung `.tar.gz` oder `.tgz`. Ist das Archiv mit **bzip2** komprimiert worden, so lautet die Endung `.tar.bz2`.

Das tar-Programm von Linux ist in der Lage mit Archiven direkt umzugehen, die mit gzip komprimiert wurden, Archive, die mit **bzip2** komprimiert wurden, müssen vorher entpackt werden.

Die Befehle zum Entpacken eines Tar-Archivs lauten also

```
tar -xzvf Archivname.tar.gz
```

für ein mit **gzip** gepacktes Archiv und

```
bzcat Archivname.tar.bz2 | tar -xvf -
```

für ein mit **bzip2** komprimiertes Archiv. Dabei ist zu beachten, daß zwischen dem Parameter f und dem folgenden Bindestrich ein Leerzeichen steht. Eine genauere Beschreibung des **tar**-Programms findet sich in der Vorbereitung auf die LPI102 Prüfung im Abschnitt 1.111.5 - Aufrechterhaltung einer effektiven Datensicherungsstrategie.

Moderne Versionen von **tar** können auch `.bz2`-Archive direkt entpacken. Dazu muß statt dem `z` ein `j` angegeben werden.

Als Ergebnis dieses Vorganges wird jetzt ein Verzeichnis erstellt, das den Quellcode des zu compilierenden Programmes enthält.

## Erstellen des Makefiles

Ein sehr einfaches Programm enthält jetzt bereits das passende Makefile, das für die Compilierung notwendig ist. Nachdem aber heute die meisten Programme von der Existenz bestimmter Libraries abhängig sind, ist in den meisten Fällen zunächst eine genaue Prüfung der in diesem System vorliegenden Besonderheiten nötig. Dazu dient das GNU-Autoconf Paket, das einen entsprechenden Test des Systems ermöglicht und aus den entsprechenden Ergebnissen dieses Tests das Makefile erstellt.

Wenn das Verzeichnis bereits eine Datei enthält, die den Namen `Makefile` trägt, so ist dieser Schritt nicht mehr nötig. Fehlt diese Datei jedoch aber es existieren die Dateie `Makefile.in` und `configure` und die Datei `configure` ist ausführbar, dann muß das Makefile erst erstellt werden.

Dazu wird einfach das Script **configure** in diesem Verzeichnis aufgerufen. Der Befehl erfordert die Eingabe eines Pfades (`./`) um zu verhindern, daß ein anderes Programm diesen Namens benutzt wird. Er lautet also einfach

```
./configure
```

Jetzt wird das System nach allen möglichen Dingen durchsucht, die notwendig sind, um das Programm zu compilieren. Fehlen entsprechende Elemente, so wird darauf hingewiesen und kein Makefile erstellt. Sind aber alle Voraussetzungen erfüllt, so endet das **configure**-Script mit der Meldung

```
Creating Makefile
```

und erstellt unser Makefile. Jetzt erst kann die wirkliche Compilierarbeit beginnen.

## Manuelles Veärndern des Makefiles

In manchen Fällen sind noch manuelle Veränderungen am Makefile notwendig. Diese Veränderungen beziehen sich in der Regel auf Pfadangaben zu bestimmten Programmen oder Angaben, wohin das Programm nach erfolgter Compilierung installiert werden soll. Diese Variablen können zumeist in den ersten paar Zeilen des Makefiles verändert werden.

Variablen werden in Makefiles ähnlich definiert, wie in Shellscripts. Eine einfache Definition sieht beispielsweise folgendermaßen aus:

```
installprefix=/usr/local
```

Der Zugriff auf Variableninhalte erfolgt aber über eine Konstruktion mit Dollarzeichen und geschweiften Klammern, also etwa `${Variablenname}`. Es könnte also folgendermaßen weitergehen:

```
installprefix=/usr/local
installdir=${installprefix}/foo
mandir=${installprefix}/man
confdir=${installdir}/etc
```

Das würde zu folgenden realen Verzeichnisnamen führen:

- installprefix=`/usr/local`
- installdir=`/usr/local/foo`
- mandir=`/usr/local/man`
- confdir=`/usr/local/foo/etc`

Wenn wir jetzt daran Veränderungen vornehmen wollen, so können wir das durch entsprechende Anpassungen an unser System tun. Um beispielsweise dafür zu sorgen, daß das Programm seine Konfigurationsdateien nicht unter `/usr/local/foo/etc` sucht, sondern unter `/etc/foo`, so verändern wir die entsprechende Zeile in

```
confdir=/etc/foo
```

Manchmal finden sich in solchen Variablen auch ganze Listen von Verzeichnissen, die z.B. nach Include-Dateien durchsucht werden sollen. Auch solche Listen können natürlich entsprechend angepasst werden, um beispielsweise mehr Verzeichnisse aufzunehmen. In der Regel werden Listenelemente durch Leerzeichen voneinander getrennt.

## Aufruf von make

Wenn alle Veränderungen am Makefile abgeschlossen wurden, so kann das

Programm jetzt compiliert werden. Dazu wird einfach der Befehl

```
make
```

eingegeben. Er compiliert jetzt das Programm nach der Regel, die im Makefile `all` genannt wurde. Der Aufruf ist also einfach eine Abkürzung für

```
make all
```

Ist dieser Befehl fehlerfrei abgearbeitet worden, dann ist das Programm jetzt compiliert. Das heißt, es liegen jetzt für jede Quellcodedatei eine Objektdatei vor und es existiert das fertige ausführbare Programm. Damit das Programm auch noch in die vorgesehenen Verzeichnisse kopiert (installiert) wird, haben die meisten Makefiles noch eine Regel, die `install` heisst und entsprechend mit

```
make install
```

aufgerufen wird. Dadurch werden die notwendigen Dateien in die vorgesehenen Zielverzeichnisse kopiert. Um alle Objektdateien wieder zu löschen kann der Befehl

```
make clean
```

benutzt werden, um sicherzustellen, daß bei einem zweiten Compiliervorgang alles erneut übersetzt wird.

## Zusammenfassung

Nachdem ein Quellcode-Paket ausgepackt wurde müssen praktisch immer die drei einfachen Befehle

```
./configure
make
make install
```

in dem Verzeichnis ausgeführt werden, in dem der Quellcode liegt. In der Regel ist mindestens für `make install` das root-Recht erforderlich, denn dieser Befehl installiert das Programm ja an Orte im System, an denen niemand außer root Schreibrecht hat.

# 1.102.4 - Verwaltung von Shared Libraries

**Beschreibung:** Prüfungskandidaten sollten in der Lage sein, Shared Libraries, die von ausführbaren Programmen benötigt werden, zu bestimmen und nötigenfalls zu installieren. Sie sollten ebenfalls fähig sein, anzugeben wo sich die Systembibliotheken befinden.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **ldd**
- **ldconfig**
- `/etc/ld.so.conf`
- `LD_LIBRARY_PATH`

---

## Die Technik der Shared Libraries

Shared Libraries sind Funktionsbibliotheken, die von mehreren Programmen benutzt werden können und die so nur einmal in den Speicher geladen werden müssen, anstatt für jedes Programm extra Platz zu verschwenden. Normalerweise würde ein Programm beim Linkvorgang (dem Vorgang, bei dem alle Objektdateien und die Systemlibraries zur ausführbaren Datei zusammengefügt werden) mit allen Libraries, aus denen das Programm Funktionen benötigt, statisch gelinkt. Das würde aber bedeuten, daß Programme, die gleichzeitig laufen, diese Funktionsbibliotheken - jedes Programm für sich - in den Speicher laden müsste. Die shared library Technik verhindert das. Ein Programm, das geladen wird, überprüft, ob die notwendigen Libraries schon im Arbeitsspeicher liegen. Tun sie das, so läd sich das Programm in den Arbeitsspeicher und benutzt einfach die schon geladenen Libraries. Liegen die benötigten Libraries noch nicht im Arbeitsspeicher, so werden sie zunächst geladen und erst dann läd sich das Programm.

So wird jede benötigte Library nur einmal in den Speicher geladen, was gerade in einem System, das viele Programme gleichzeitig geladen hält eine enorme Speichereinsparung bewirkt.

Wenn ein Programm geladen wird, dann muß es eine Instanz geben, die den geschilderten Vorgang startet. Denn das Programm selbst kann das nicht tun, es wird ja erst geladen, wenn die nötigen Libraries auch schon geladen sind. Diese Instanz ist sozusagen der Programmlader oder der sogenannte *dynamische Linker und Lader*. Der Name dieses Programms ist **ld.so**.

**ld.so** läd die shared libraries, die ein Programm benötigt, bereitet dann das

Programm entsprechend vor und läd es selbst. Die shared libraries, die das zu ladende Programm benötigt, werden in der folgenden Reihenfolge gesucht:

- Alle Pfade, die in der Umgebungsvariable `LD_LIBRARY_PATH` eingetragen sind, werden nach den Libraries durchsucht. `LD_LIBRARY_PATH` ist eine Shellvariable, die eine durch Doppelpunkte getrennte Liste von Verzeichnissen enthält.
- Die Datei `/etc/ld.so.cache` enthält eine binäre Liste aller Librarie-Kandidaten, die schon vorher in den genannten Verzeichnissen gefunden wurden. (siehe weiter unten bei **ldconfig**.)
- Die Verzeichnisse `/usr/lib` und `/lib` werden durchsucht.

Ein Linux-Programm wird also von einem speziellen Programmlader (**ld.so**) geladen, der die notwendigen Libraries gleich mitläd. Aus diesem Grund ist es nötig, daß jedes Programm, das installiert wird, auch die entsprechenden Libraries mitinstalliert bzw. überprüft, ob sie schon installiert sind. Die Verwaltung dieser Libraries obliegt der Systemverwaltung.

## Welches Programm braucht welche Library?

Um herauszufinden, welche Shared Libraries ein bestimmtes Programm benutzt, gibt es das kleine Programm ldd. Dieses Programm gibt eine Liste aller Libraries zurück, die das Programm benötigt, daß als Parameter **ldd** mitgegeben wurde. Um z.B. herauszubekommen, welche Libraries das Programm **ls** benötigt, schreiben wir

```
ldd /bin/ls
```

Das zu untersuchende Programm muß mit vollem Pfad angegeben werden. Die Ausgabe wäre für unser Beispiel dann etwas in der Art:

```
librt.so.1 => /lib/librt.so.1 (0x40022000)
libc.so.6 => /lib/libc.so.6 (0x40033000)
libpthread.so.0 => /lib/libpthread.so.0 (0x4014e000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Daraus geht hervor, daß das Programm **ls** vom Linker/Loader `/lib/ld-linux.so.2` gestartet werden will und die Libraries `librt.so.1, libc.so.6` und `libpthread.so.0` benötigt. In der rechten Spalte sind die Fundorte der Libraries im Dateisystem zu lesen.

Wenn ein Programm neu installiert wird, so kann mit Hilfe dieses Befehls herausgefunden werden, ob alle Libraries für das Programm existieren oder ob einige nachinstalliert werden müssen.

## Libraries installieren und aktualisieren

Wenn neue Libraries installiert werden sollen, dann stellt sich die Frage, wohin. Zunächst einmal bieten sich die Verzeichnisse `/lib, /usr/lib` und `/usr/local/lib` an. Sollen aber neue Verzeichnisse angelegt werden, die Libraries enthalten, dann muß das der internen Verwaltung der Libraries mitgeteilt werden.

Die interne Verwaltung der Libraries wird durch den Aufruf des Programms **ldconfig** vollzogen. Dieses Programm wartet den Library Cache und erstellt automatisch die notwendigen symbolischen Links auf Libraries. Der Librarie-Cache liegt in der Datei `/etc/ld.so.cache` und enthält eine binär codierte Liste aller dem System bekannten Libraries.

Damit **ldconfig** diese Datei erstellen kann und auch neu hinzugekommene Libraries dort aufgenommen werden, muß **ldconfig** wissen, welche Verzeichnisse nach Libraries durchsucht werden sollen. **ldconfig** durchsucht zunächst die beiden Verzeichnisse `/usr/lib` und `/lib`, danach alle Verzeichnisse, die in der Datei `/etc/ld.so.conf` aufgelistet sind.

Wenn also ein neues Programm **foo** installiert wird, das seine Shared Libraries im Verzeichnis `/usr/local/foo/lib` ablegt, so müssen wir nur dieses Verzeichnis in die Datei `/etc/ld.so.conf` aufnehmen. Nach der Installation neuer Libraries und/oder der Neuaufnahme von Pfaden in der Datei `/etc/ld.so.conf` muß das Programm **ldconfig** ausgeführt werden, bevor die neuen Libraries verwendet werden können. Erst nach dem Aufruf von **ldconfig** stehen sie ja in der Datei `/etc/ld.so.cache` und sind somit dem Linker/Loader bekannt.

# 1.102.5 - Verwendung des Debian Paketmanagements

**Beschreibung:** Prüfungskandidaten sollten in der Lage sein, mit dem Debian Paketmanagement umzugehen. Dieses Lernziel beinhaltet das Benutzen von Kommandozeilen- und interaktiver Werkzeuge zum Installieren, Updaten oder Deinstallieren von Paketen sowie das Auffinden von Paketen, die spezifische Dateien oder Software enthalten (installierte bzw. nicht installierte Pakete). Ebenfalls enthalten ist das Abfragen von Informationen wie Version, Inhalt, Abhängigkeiten, Paketintegrität und Installationsstatus (ob installiert oder nicht) von Paketen.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **unpack**
- **configure**
- `/etc/dpkg/dpkg.cfg`
- `/var/lib/dpkg/*`
- `/etc/apt/apt.conf`
- `/etc/apt/sources.list`
- **dpkg**
- **dselect**
- **dpkg-reconfigure**
- **apt-get**
- **alien**

---

Es gibt im Linux Bereich zwei große Modelle einer Paketverwaltung für zu installierende Programmpakete. Nachdem sich das alte `tgz`-Format für die Verwendung bei binären (vorcompilierten) Paketen als unzulänglich erwiesen hatte, weil es keinerlei Mechanismen zur Deinstallation aufwies, begannen zwei verschiedene Distributionen, eigene Paketverwaltungen aufzubauen. Debian und RedHat.

Inzwischen haben nahezu alle Distributionen das RedHat-Paketmanagement übernommen, nur Debian beharrt auf sein eigenes Modell. Und das nicht zu Unrecht, ist es doch gerade die Paketverwaltung von Debian, die schlichtweg als genial bezeichnet werden muß.

Als distributionsübergreifende Zertifizierung fordert LPI das Wissen um beide Formate. Beide Formate werden auch tatsächlich abgefragt und das nicht zu knapp. In diesem Kapitel wird die Debian-Verwaltung besprochen, im nächsten kommt dann

die RedHat-Technik zur Sprache.

## Das grundlegende Prinzip der Debian Paketverwaltung

Debian Pakete enden mit der Endung `.deb`. Sie haben ein einheitliches Namensschema, das sich folgendermaßen zusammensetzt:

*Programmname_Versionsnummer_Architektur*`.deb`

Wobei *Programmname* der Name des Programmes (oder des Paketes) ist, unter dem es später dann auch angesprochen werden kann, *Versionsnummer* ist die Version des Programms (oder Paketes) und *Architektur* bezeichnet die Hardwarearchitektur, für die dieses Paket gedacht ist, also etwa `i386` für die PC-Architektur.

Ein Debian Paket besteht aus einem `ar`-Archiv, das wiederum zwei komprimierte `tar`-Archive und eine Versionsdatei beinhaltet. Das erste `tar`-Archiv (`data.tar.gz`) enthält die zu installierenden Dateien, das zweite (`control.tar.gz`) enthält die Metainformationen über das Paket, die Scripts, die zum Installieren und Deinstallieren benötigt werden sowie eine Prüfsumme.

Jedes Debian-Paket enthält vier Scripte, je eines das vor und nach der Installation bzw. vor und nach der Deinstallation abgearbeitet wird.

Wird ein Debian-Paket installiert, so werden die Informationen über dieses installierte Paket in mehrere Dateien im Verzeichnis `/var/lib/dpkg` abgelegt, so daß jederzeit eine Überprüfung der bereits installierten Pakete und Dateien stattfinden kann. Die Informationen werden folgendermaßen aufgeteilt:

- In der Datei `/var/lib/dpkg/available` werden die Paketinformationen aller zur Verfügung stehender (installierbarer) Pakete abgelegt.
- In der Datei `/var/lib/dpkg/status` werden die Informationen über den Status der Installation abgelegt, so daß unterschieden werden kann zwischen korrekt oder nur teilweise installierten Paketen.
- Im Verzeichnis `/var/lib/dpkg/info` liegen zu jedem installierten Paket die vier Scripte (`*.preinst, *.postinst, *.prerm, *.postrm`), eine Liste aller enthaltener Dateien (`*.list`), die md5-Prüfsummendatei (`*.md5sums`) und evt. noch andere Informationen wie die zur Verfügung gestellten Libraries (`*.shlibs`).

Mit Hilfe dieser Informationen sind sowohl Installation, als auch Deinstallation von Paketen sehr sicher und umfassend möglich. Man kann das Verzeichnis `/var/lib/dpkg` also als Systemdatenbank der installierten Pakete betrachten. In der Regel sind keine manuellen Zugriffe auf dieses Verzeichnis notwendig, alle Zugriffe werden mit den Hilfsprogrammen erledigt, die im Folgenden näher besprochen werden.

## Das dpkg-Programm

Das Programm **dpkg** ist sozusagen das Low-Level Paketverwaltungsprogramm. Mit ihm ist es möglich, direkt bestimmte Pakete zu installieren oder zu deinstallieren. Noch eine Stufe weiter unten liegt das Programm **dpkg-deb**, das die reine Archivverwaltung (Packen/Entpacken) erledigt, aber niemals manuell aufgerufen wird.

Das **dpkg** Programm muß nur dann manuell aufgerufen werden, wenn direkt vorliegende `.deb`-Pakete installiert werden sollen. Später werden wir das Programm **apt-get** kennenlernen, das sich auch um den Bezug der Pakete kümmert, dann aber selbstständig **dpkg** aufruft, um Pakete zu installieren bzw. zu deinstallieren. Ist jedoch ein Paket manuell z.B. aus dem Internet besorgt worden, dann benötigen wir **dpkg**.

### Pakete installieren mit dpkg

Um ein Paket mit **dpkg** zu installieren wird folgender Befehl ausgeführt:

dpkg [*Optionen*] -i | --install *Paketdatei*.deb

(-i | --install bedeutet entweder -i oder --install). Wenn als Option `-R` oder `--recursive` angegeben wurde, so steht statt eines Paketnamens ein Verzeichnisname. Alle Debianpakete des genannten Verzeichnisses werden dann installiert.

Die Installation besteht aus folgenden Einzelschritten:

1. Die Kontrolldateien des neuen Paketes werden entpackt.
2. Wenn eine ältere Version des selben Paketes bereits installiert war, wird das PreRemove-Script der älteren Version ausgeführt.
3. Das PreInst-Script des neuen Paketes wird ausgeführt.
4. Die Dateien des neuen Paketes werden entpackt und gleichzeitig werden die Dateien einer eventuell existierenden älteren Version gesichert, so daß im Falle eines Fehlers die Installation des neuen Paketes rückgängig gemacht werden kann.
5. Wenn eine ältere Version existierte, dann wird das PostRemove-Script der älteren Version ausgeführt.
6. Das neue Paket wird durch das Abarbeiten des PostInstall-Scripts konfiguriert.

### Pakete deinstallieren mit dpkg

Wenn ein installiertes Paket entfernt werden soll. dann werden zwei Möglichkeiten angeboten. Entweder werden zwar alle Dateien entfernt, aber die

Konfigurationsdateien des Paketes bleiben installiert, oder es werden wirklich alle Dateien, auch die Konfigurationsdateien entfernt.

Zum normalen Deinstallieren (ohne Entfernung der Konfigurationsdateien) wird folgender Befehl benutzt:

dpkg [*Optionen*] -r | --remove *Paketname*

Um wirklich alles zu entfernen benutzt man

dpkg [*Optionen*] -P | --purge *Paketname*

*Paketname* ist hier der Name des Paketes, nicht der Name der Paketdatei. Wurde die Datei `foo_1.0.23_i386.deb` installiert, so ist der Paketname einfach nur `foo`.

Das Entfernen der Pakete geht in folgenden Schritten vor sich:

1. Das PreRemove-Script des Paketes wird abgearbeitet.
2. Die zu entfernenden Dateien werden gelöscht.
3. Das PostRemove-Script wird abgearbeitet.

Informationen über installierte Pakete abfragen

Um Informationen über ein bestimmtes installiertes Paket zu erhalten, wird der Befehl

dpkg -p|--print-avail *Paketname*

ausgeführt. **dpkg** liesst daraufhin die Informationen aus der Datei `/var/lib/dpkg/available` und gibt sie aus.

Auflisten der installierten Pakete

Um eine Liste aller oder bestimmter installierter Pakete zu bekommen, wird der Befehl

dpkg -l | --list [*Paketnamensmuster*]

ausgeführt. Wird kein Namensmuster angegeben, so werden alle installierten Paketnamen ausgegeben, ansonsten nur die Pakete, deren Namen auf das angegebene Muster passen. Als Muster werden die üblichen Shell-Wildcards benutzt. Zu beachten ist, daß die Muster in Anführungszeichen gesetzt werden sollten, um die Shell daran zu hindern, sie zu interpretieren.

Auflisten der Dateien installierter Pakete

Um alle Dateien eines bestimmten Paketes aufzulisten, existiert der Befehl

dpkg -L | --listfiles *Paketname*

Die Dateien, die von Installationsscripts angelegt wurden, werden hier nicht angezeigt.

Abfrage des Installationsstatus

Debian-Pakete, die installiert werden, bekommen einen bestimmten Status, der angibt, wie weit die Installation fortgeschritten ist. Wenn eine Installation aus welchen Gründen auch immer, nicht vollständig abgeschlossen wurde, so kann das später aus dem Status entnommen werden. Folgende Stati sind gültig:

**installed**
: Das Paket ist vollständig installiert

**half-installed**
: Die Installation wurde nicht vollständig ausgeführt

**not-installed**
: Das Paket ist nicht installiert

**unpacked**
: Das Paket ist zwar ausgepackt, aber nicht konfiguriert

**half-configured**
: Das Paket ist ausgepackt und die Konfiguration wurde begonnen, aber nicht korrekt abgeschlossen

**config-files**
: Nur die Konfigurationsdateien des Paketes existieren auf dem System (typischerweise nach einer Deinstallation mit -R)

Um den Status eines Paketes abzufragen wird der Befehl

dpkg -s | --status *Paketname*

ausgeführt. Die Informationen werden der Datei `/var/lib/dpkg/status` entnommen.

Aus welchem Paket stammt eine bestimmte Datei?

Mit dem Befehl

dpkg -S | --search *Dateinamesmuster*

werden die installierten Dateien aus `var/lib/dpkg/info` durchsucht und alle Paketnamen zurückgegeben, die entsprechende Dateien enthalten. Zur

Musterbildung stehen alle Shell-Wildcards zur Verfügung.

Optionen für dpkg

Vor jedem der besprochenen Befehle können verschiedene Optionen gesetzt werden, die den Ablauf des Befehls modifizieren können. Diese Optionen beziehen sich z.B. auf den Umgang mit Abhängigkeiten, oder ab wievielen Fehlern sich **dpkg** weigert, weiterzuarbeiten. Diese Optionen können entweder direkt an der Kommandozeile eingegeben werden, oder in die Datei `/etc/dpkg/dpkg.cfg` eingetragen werden. In letzterem Fall, werden die Optionen für jeden **dpkg** Befehl angewandt. Wichtige Optionen sind:

**--abort-after=***Zahl*
: Gibt an, nach wievielen Fehlern **dpkg** beendet werden muß. Voreingestellt ist 50.

**-B|--auto-deconfigure**
: Wenn ein Paket deinstalliert wird, so ist es möglich, daß ein anderes installiertes Paket dieses Paket gebraucht hätte. Mit dieser Option werden automatisch alle anderen Pakete auch deinstalliert, die das zu installierende Paket benötigt hätten.

**--ignore-depends=***Paket*
: Abhängigkeiten für das angegebene Paket werden ignoriert. Es werden solche Abhängigkeiten zwar überprüft, aber es werden nur Warnungen statt Fehlermeldungen angezeigt, wenn nicht erfüllte Abhängigkeiten auftreten.

**--refuse-downgrade | -G**
: Ein Paket, von dem eine neuere Version bereits installiert ist, wird nicht installiert

Eine vollständige Liste aller denkbaren Optionen sind der Handbuchseite zu entnehmen.

## dpkg-reconfigure

Wenn ein bereits installiertes Paket erneut konfiguriert werden soll, so kann das durch den Aufruf von

dpkg-reconfigure *Paketname*

vorgenommen werden. Das hat aber nur dann eine tatsächliche Bedeutung, wenn ein Paket ein `debconf`-Script enthält.

## dselect

Das Programm **dselect** ist ein menügeführtes Frontend für **dpkg**. Es übernimmt aber

nicht nur die Aufgabe, Pakete zu installieren oder zu deinstallieren, sondern auch die Verwaltung der zur Verügung stehenen Pakete und der gegenseitigen Abhängigkeiten.

**dselect** bietet verschiedene Methoden an, um die zur Verfügung stehenden Pakete zu managen. Dabei geht es hauptsächlich darum, woher die Debian-Pakete bezogen werden sollen. Folgende Methoden werden angeboten:

**cdrom**
: Installiert von einer Debian-CDROM. Die CDROM kann, muß aber nicht gemountet sein und sollte ein ISO9660 Dateisystem beinhalten.

**nfs**
: Installiert über einen (noch nicht gemounteten) NFS-Server. Der Server muß die Paketbeschreibungsdateien (`Packages.gz`) jeder Distributionsabteilung (stable, contrib und non-free) zur Verfügung stellen und natürlich die entsprechenden `.deb` Dateien.

**harddisk**
: Installiert von einer noch nicht gemounteten Festplattenpartition. Diese Partition muß die selben Elemente wie ein NFS-Server enthalten.

**mounted**
: Installation von einem bereits gemounteten Dateisystem. Dabei kann es sich entweder um eine gemountete Festplattenpartition oder um eine gemounteten NFS-Freigabe handeln. Zum Inhalt gilt das selbe wie bei NFS-Server und Festplatte.

**floppy**
: Installation über einen Stapel Disketten, von dem die erste Disk die Paketinformationen enthalten sollte. Veraltert.

**apt**
: Benutzt die neue Methode, die weiter unten bei **apt-get** beschrieben wird. Hier kann die Installationsquelle entweder über ein Netz (ftp/http) oder über das Dateisystem (file) angesprochen werden.

Das **dselect** Programm wird normalerweise ohne weitere Parameter aufgerufen. Es bietet menügeführt folgende Möglichkeiten:

**Zugriff (access)**
: Auswahl der Zugriffsmethode (siehe oben).

**Erneuern (update)**
: Erneuert die Liste der verfügbaren Pakete, wenn möglich.

**Auswählen (select)**
: Menügeführte Auswahl aller zu installierender oder zu entfernender Pakete.

**Install**
: Installiert die Pakete, die angewählt wurden.

**Konfig**

Konfiguriert Pakete, die bei der letzten Installation nicht vollständig konfiguriert wurden.

**Löschen (remove)**
Entfernt die zum Löschen markierten Pakete.

Als menügeführtes Programm ist die Bedienung von **dselect** zwar manchmal gewöhnungsbedürftig, jedoch nicht weiter schwierig.

## apt-get

**apt-get** ist das Kommandozeilenprogramm, das die modernste der Zugriffsmethoden auf Debian-Pakete steuert. Damit ist es möglich, die Installation und verschiedene andere Aufgaben der Paketverwaltung über ein einfaches Kommando auszuführen. Das Programm speichert seine Konfiguration in der Datei `/etc/apt/apt.conf` oder bei den moderneren Versionen in mehreren Verzeichnissen unter `/etc/apt/apt.conf.d`.

Die Quellen, von denen **apt-get** seine Debian-Pakete bezieht, werden in der Datei `/etc/apt/sources.list` angegeben. Diese Datei kann entweder von Hand erstellt/manipuliert werden oder über das Programm **dselect oder apt-setup** im Menüpunkt *Zugriff*.

**apt-get** wird immer in einer der folgenden Formen aufgerufen:

**apt-get upgrade**
Holt die neuesten Paketbeschreibungen der in `/etc/apt/sources.list` angegebenen Installationsquellen.

**apt-get install** *Paketname*
Installiert das angegebene Paket von den eingestellten Installationsquellen. Das Paket und alle weiteren notwendigen Pakete die von ihm erfordert werden, wird vollständig installiert (mit dpkg -i).

**apt-get remove** *Paketname*
Das angegebene Paket wird deinstalliert.

**apt-get source** *Paketname*
Das Quellcode-Paket des angegebenen Paketes wird installiert.

Seine wahre Stärke spielt **apt-get** aus, wenn als Quellen offizielle Debian-Server im Internet angegeben wurden und der Paketname des zu installierenden Paketes bekannt ist. Wird z.B. festgestellt, daß auf einem System das Programm **foo** nicht installiert ist, so kann durch die Angabe des Befehls

```
apt-get install foo
```

die Installation des Programms ausgeführt werden, ohne lange ein menügeführtes

Programm aufzurufen. Es existieren inzwischen auch mehrere Frontends für **apt-get**. Für Textterminals gibt es **aptitude** und für graphische Terminals kann **gnome-apt** verwendet werden.

## alien

Debian ist die Distribution, die die meisten Pakete von allen bekannten Distributionen anbietet. Falls trotzdem einmal ein Paket nicht im `.deb` Format vorliegen sollte, man also gezwungen ist, auf ein `.rpm`-Paket zurückzugreifen, gibt es speziell dafür das Programm **alien**.

**alien** konvertiert verschiedene Paketformate in jeweils andere um. Das Programm kann mit folgenden Formaten umgehen:

- RedHat (rpm)
- Debian (deb)
- Stampede (slp)
- Slackware (tgz)
- Solaris (pkg)

Die Aufrufform ist sehr einfach. **alien** erkennt das Format einer angegebenen Paketdatei automatisch und konvertiert es dann in das angegebene Format. Das gewünschte Ausgabeformat wird durch die folgenden Parameter angegeben:

**--to-deb**
: Aus dem angegebenen Paket wird ein Debian-Paket erstellt

**--to-rpm**
: Aus dem angegebenen Paket wird ein RedHat-Paket erstellt

**--to-tgz**
: Aus dem angegebenen Paket wird ein Slackware-Paket erstellt

**--to-slp**
: Aus dem angegebenen Paket wird ein Stampede-Paket erstellt

Wenn also das Paket `foo-1.2.34.i386.rpm` existiert, und daraus ein Debian-Paket erstellt werden soll, so genügt der Befehl

```
alien --to-deb foo-1.2.34.i386.rpm
```

und **alien** erzeugt daraus das Paket `foo_1.2.34_i386.deb`

Jetzt kann dieses neu erstellte Paket mit **dpkg -i** installiert werden und wird so nahtlos in die Debian-Paketverwaltung integriert.

# 1.102.6 - Verwendung des Red Hat Package Managers (RPM)

**Beschreibung:** Prüfungskandidaten sollten in der Lage sein, Paketmanagement auf Linuxdistributionen, die RPM-Pakete für die Paketverwaltung verwenden, durchzuführen. Dieses Lernziel beinhaltet die (Neu-)Installation, das Update und das Entfernen von Paketen sowie das Abfragen von Status- und Versionsinformationen. Ebenfalls enthalten ist das Abfragen von Paketinformationen wie Abhängigkeiten, Integrität und Signaturen. Kandidaten sollten auch in der Lage sein, zu bestimmen, welche Dateien von einem Paket zur Verfügung gestellt werden, und das Paket zu finden, dem eine bestimmte Datei entstammt.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- `/etc/rpmrc`
- `/usr/lib/rpm/*`
- **rpm**
- **grep**

---

Die meisten heutigen Linux-Distributionen haben das Paketformat von RedHat übernommen, weil es eine sehr weitreichende Verwaltung der installierten Software ermöglicht. Dazu zählen unter anderem die korrekte Installation und Deinstallation, die Informationen über installierte Programme und die Verifikation, daß bestimmte Pakete tatsächlich das enthalten, was sie vorgeben. Die gesamte Verwaltung der Red-Hat Pakete wird durch ein Programm realisiert, das den Namen **rpm** trägt. Auch Nicht-RedHat Distributionen unterstützen dieses Programm.

## Das Prinzip der RPM-Verwaltung

Jedes RedHat Paket hat einen Dateinamen, der auf `.rpm` endet. Die Dateinamensstruktur ähnelt der von Debian:

*Programmname*-*Versionsnummer*.*Architektur*`.rpm`

Wobei *Programmname* der Name des Programmes (oder des Paketes) ist, unter dem es später dann auch angesprochen werden kann, *Versionsnummer* ist die Version des Programms (oder Paketes) und *Architektur* bezeichnet die Hardwarearchitektur, für die dieses Paket gedacht ist, also etwa i386 für die PC-Architektur.

Jedes RPM-Paket enthält neben den zu installierenden Dateien wie Debian-Pakete

vier Scripts, die jeweils vor und nach der Installation und Deinstallation ausgeführt werden. Zusätzlich existieren Informationen über das Paket und die Abhängigkeiten, die die darin enthaltenen Programme besitzen.

Im Verzeichnis `/usr/lib/rpm` liegen die verschiedenen Hilfsmittel, die das **rpm**-Programm benötigt und unter `/var/lib/rpm` finden sich die verschiedenen Datenbanken über die installierten Pakete. Im Gegensatz zu Debian werden diese Datenbanken in einem proprietären Format abgespeichert, das nur mit dem **rpm**-Programm zugänglich ist.

## Installieren und deinstallieren von Paketen

Eine der Hauptaufgaben des **rpm**-Programmes ist das Installieren und Deinstallieren von Paketen. Dazu dienen die folgenden Befehle.

Installieren eines Paketes

Um eine bestehende `.rpm`-Datei zu installieren, wird folgender Befehl verwendet:

rpm -i | --install [*Optionen*] *Paketdatei*

das `-i | --install` bedeutet, daß entweder ein `-i` oder ein `--install` angegeben wird. Mögliche Optionen sind:

**--nodeps**
: Die Überprüfung von Abhängigkeiten wird abgeschalten.

**--noscripts**
: Es werden keine pre- oder postinstall Scripts abgearbeitet.

**--test**
: Die Installation wird nur simuliert, keine Dateien werden installiert.

**--excludedocs**
: Die im Paket enthaltenen Dokumentationen werden nicht mitinstalliert.

**--replacepkgs**
: Die Installation wird durchgeführt, auch wenn schon Teile installiert sind.

**--replacefiles**
: Die Installation wird auch durchgeführt, auch wenn dabei Dateien überschrieben werden, die von anderen Paketen stammen.

**--oldpackage**
: Erlaubt ein Upgrade von älteren Versionen als die bereits installierten.

**--force**
: Entspricht der Kombination von `--replacepkgs --replacefiles --oldpackage`.

Vorsicht ist geboten, wenn die Überprüfung der Abhängigkeiten abgeschalten wird,

oder das zwingende Installieren von Paketen aktiviert wird. Das kann zu Inkonsistenzen des Systems führen und sollte nur verwendet werden, wenn man genau weiß, was man tut.

Deinstallieren eines Paketes

Ein Paket wird mit folgendem Befehl deinstalliert:

rpm -e | --uninstall [*Optionen*] *Paketname*

Auch für die Deinstallation gibt es einige Wichtige Optionen:

**--nodeps**
: Die Überprüfung von Abhängigkeiten wird abgeschalten.

**--noscripts**
: Es werden keine pre- oder postuninstall Scripts abgearbeitet.

**--test**
: Die Deinstallation wird nur simuliert, keine Dateien werden gelöscht.

**--allmatches**
: Deinstalliert alle Pakete, auf die das Namensmuster zutrifft. In diesem Fall ist *Paketname* also ein Muster und nicht ein Paketnamen.

Upgrade eines Paketes

Wenn eine neue Version eines bereits installierten Programmes aufgespielt werden soll, so kommen zwei verschiedene Techniken zum Einsatz:

rpm -U|--upgrade [*Optionen*] *Paketdatei*

Als Optionen kommen exakt die selben Optionen wie beim Installieren zum Einsatz. Der Befehl entspricht im Übrigen genau dem, der zum Installieren benutzt wird, nur daß eventuell vorher existierende Versionen des Paketes vor der Installation entfernt werden.

rpm -F|--freshen [*Optionen*] *Paketdatei*

Dieser Befehl, der auch wieder die selben Optionen wie beim Installieren kennt, installiert die angegebene Paketdatei nur, wenn eine ältere Version des selben Programms vorher schon installiert war. Dann verhält er sich wie der vorige (-U)

## Abfage von Paketinformationen

Die generelle Form, wie Paketinformationen abgefragt werden, beginnt immer mit einem `-q` oder `--query`. Dem folgen bestimmte andere Befehle, die dann die

eigentliche Abfrage stellen.

Wenn ein normaler Querry-Befehl angegeben wird, so bezieht er sich auf schon installierte Pakete. **rpm** durchsucht also nicht eine bestimmte Paketdatei, sondern die Datenbanken, in denen die Informationen über installierte Pakete liegen. Soll aber Information über eine Paketdatei erfragt werden, die noch nicht installiert ist, so muß die Option `-p` angegeben werden. Statt der Angabe eines Paketnamens ist dann die Angabe einer Paketdatei notwendig.

Abfrage von Versionsinformationen

Um die Versionsnummer eines installierten Paketes abzufragen wird einer der folgende Befehle benutzt:

```
rpm -qi Paketname
rpm --query --info Paketname
```

Soll stattdessen die Versionsinformation eines nicht installierten RPM-Paketes erfragt werden (was eigentlich unnötig ist, da diese Information im Dateinamen schon vorliegt) kommt einer der folgenden Befehle zum Einsatz

```
rpm -qi -p Paketdatei
rpm --query --info --package Paketdatei
```

Natürlich können die langen und kurzen Optionen auch durcheinander benutzt werden wie `--query -i` oder `-q --info`.

Neben der reinen Versionsinformation werden auch noch andere Paketinformationen ausgegeben. Wenn tatsächlich nur die Information über die Version abgefragt werden soll, kann auch der eingebaute Variablensubstitutionsmechanismus benutzt werden:

```
rpm -q Paketname --queryformat "%{VERSION}\n"
```

Auf diese Art und Weise können beliebige Informationen in einem beliebigen Format ausgegeben werden. Eine komplette Liste aller so benutzbaren Variablensubstitutionen ist mit dem Befehl

```
rpm --querytags
```

einsehbar. In der gegenwärtigen Version sind es 98 verschiedene Typen wie etwa NAME, VERSION, RELEASE, DESCRIPTION, SIZE, VENDOR, FILENAMES, ...

Auflisten aller Dateien eines Paketes

Oft soll überprüft werden, welche Dateien ein bestimmtes Paket geliefert hat. Mit dem Befehl

rpm -ql *Paketname*

wird eine Liste aller Dateien des angegebenen Paketes ausgegeben. Statt `-l` kann auch `--list` verwendet werden. Soll stattdessen wieder eine noch nicht installierte Paketdatei untersucht werden, so gilt wieder der Parameter `-p`

rpm -ql -p *Paketdatei*

Die Ausgabe erfolgt auf die Standard-Ausgabe, kann also problemlos mit Tools wie grep weiterverarbeitet werden.

Auflisten der Dokumentationsdateien eines Paketes

Um nur die Dateien eines Paketes aufzulisten, die reine Dokumentationsdateien (Handbuchseiten, READMEs, ...) sind, existiert der Befehl

rpm -qd *Paketname*

oder wieder mit langen Optionen `--docfiles` statt dem `d`. Auch dieser Befehl ist mit `-p` auf eine Paketdatei anwendbar.

Auflisten der Konfigurationsdateien eines Paketes

Analog zur Ausgabe der Dokumentationsdateien gibt es eine Möglichkeit nur die Konfigurationsdateien auszugeben:

rpm -qc *Paketname*

oder wieder mit langen Optionen `--configfiles` statt dem `c`. Auch dieser Befehl ist mit `-p` auf eine Paketdatei anwendbar.

Auflisten der Scriptdateien eines Paketes

Manchmal ist es auch interessant, herauszufinden, was die Installatiopsscripts eines Paketes eigentlich tun. Hierfür gibt es den Befehl

rpm -q --scripts *Paketname*

Er listet hintereinander alle vier Scripts auf die Standard-Ausgabe, sollte also

entweder in eine Datei umgeleitet oder an less weitergepiped werden.

Gerade dieser Befehl ist natürlich wichtig für noch nicht installierte Paketdateien, um vorher sicherzustellen, daß das Script nichts unerwünschtes macht. Also steht uns auch hier wieder die Option -p zur Verfügung.

## Abfragen über installierte Pakete

Die nächsten zwei Befehle beziehen sich nur auf bereits installierte Pakete, sind aber auch Query-Formen des **rpm** Programms.

Auflisten aller installierten Pakete

Wenn einmal nicht klar ist, welche Pakete bereits installiert sind, so kann **rpm** auch eine Liste aller installierten Pakete ausgeben. Das geschieht mit dem Befehl

```
rpm -qa
```

oder

```
rpm -q --all
```

**rpm** verwaltet auch bestimmte Gruppen, so daß auch Abfragen möglich sind, die alle installierten Pakete einer Gruppe ausgeben:

```
rpm -qg Gruppe
```

oder

```
rpm -q --group Gruppe
```

Diese Ausgaben können natürlich sehr umfangreich sein, können aber wiederum z.B. mit **grep** durchsucht werden. Wenn z.B. erforscht werden soll, welche Pakete installiert sind, die etwas mit dem Editor **joe** zu tun haben, können wir einfach schreiben

```
rpm -qa | grep joe
```

Die Ausgabe wäre dann z.B.:

```
joe-2.8-154
```

Daraus können wir jetzt weitere Informationen gewinnen, mit einem der oben genannten Abfragemöglichkeiten.

Aus welchem Paket stammt eine bestimmte Datei?

Oft ist einfach die Frage, aus welchem Paket eine bestimmte Datei stammt. Auch diese Frage lässt sich mit **rpm** lösen.

```
rpm -qf Dateiname
```

oder

```
rpm -q --file Dateiname
```

**rpm** durchsucht jetzt seine Datenbanken nach diesem Dateinamen und gibt uns das Paket zurück, aus dem diese Datei stammt.

## Abhängigkeiten

Oft benötigen bestimmte Pakete andere Pakete, um zu funktionieren. Diese Abhängigkeiten sind auch mit **rpm** herausfindbar.

Welche installierten Pakete benützt ein bestimmtes Paket?

Wenn ein Paket entfernt werden soll oder eine Datei gelöscht werden soll, dann stellt sich die Frage, ob das Paket bzw. die Datei nicht noch von einem anderen Paket benötigt werden würde. Mit dem Befehl

```
rpm -q --whatrequires Paket
```

oder

```
rpm -q --whatrequires Datei
```

kann herausgefunden werden, welches installierte Paket das angegebene Paket oder die angegebene Datei benötigt.

Welche Pakete werden von einem bestimmten Paket benötigt?

Umgekehrt kann es auch interessant sein, welche Libraries und andere Pakete von einem bestimmten Paket benötigt werden. Mit

```
rpm -qR Paketname
```

oder

rpm -q --requires *Paketname*

wird eine Liste aller Libraries und Pakete ausgegeben, die von dem angegebenen Paket benötigt werden.

## Sicherheitsabfragen

Wenn Pakete installiert werden sollen, dann kann es interessant sein, ob diese Pakete wirklich aus der Quelle kommen, aus der sie vorgeben zu kommen. Es wäre ja auch denkbar, daß jemand ein Paket mit einem Trojanischen Pferd einschmuggeln will, und dazu ein echtes Paket mit seinem Paket austauscht. Auf der anderen Seite kann der Inhalt eines Paketes auch durch Übertragungsfehler unbeabsichtigt verfälscht werden. Gegen beide dieser Gefahren kann mit **rpm** vorgegangen werden.

Verifizierung der Paketintegrität eines Paketes

Eine Paketverifizierung vergleicht die Informationen, über die installierten Dateien eines Paketes mit den Informationen über diese Dateien, die in den Metadaten des Paketes in der Paketdatenbank gespeichert sind. Neben anderen Dingen vergleicht die Verifizierung die Größe, dei MD5-Prüfsumme, Zugriffsrechte, Typ, Eigentümer und Zugriffsrechte jeder Datei. Jede Diskrepanz wird dargestellt.

rpm -V |--verify *Paketname*

Das Format der Ausgabe ist eine acht Zeichen lange Zeichenkette, ein mögliches "**c**" (für Konfigurationsdatei) und der Dateiname. Jedes der acht Zeichen bezeichnet das Ergebnis eines Vergleichs zwischen den Attributen einer Datei und denen, die in der Paketdatenbank gespeichert sind. Ein einfacher Punkt (.) bedeutet, daß der Test durchgelaufen ist, während ein Fragezeichen (?) anzeigt, daß der Test nicht durchgeführt werden konnte (z.B. weil Zugriffsrechte zum Lesen gefehlt hatten). In jedem anderen Fall repräsentieren die Buchstaben Fehler des jeweiligen Verifizierungstests:

**S**
: (Size) Die Dateigröße stimmt nicht

**M**
: (Mode) Der Zugriffsmodus stimmt nicht

**5**
: Die MD5 Prüfsumme stimmt nicht

**D**
: Die Major/Minor Numer der Gerätedatei stimmt nicht

**L**

ReadLink(2) Systemaufruffehler

**U**

Eigentümer stimmt nicht

**G**

Gruppenzugehörigkeit stimmt nicht

**T**

Die Zeitmarke der mtime stimmt nicht

Abfrage der PGP/GPG Signatur eines Paketes

Abschließend bietet **rpm** noch die Möglichkeit, daß ein Paket von seinem Autor mit einer PGP oder GPG Signatur (einer Art elektronischer Unterschrift) versehen wird. Diese Signatur kann mit folgendem Befehl überprüft werden:

```
rpm --checksig Paketdatei
```

Damit diese Überprüfung funktioniert muß eine korrekte Installation von PGP oder GPG vorliegen und die Konfiguration von **rpm** muß dies berücksichtigen. Die Darstellung dieser Technik würde den Rahmen dieser Dokumentation sprengen.