Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2011/2012

Shell-Programmierung

Boot-Prozess

Einfaches Shell-Skript

Im folgenden soll ein Shell-Skript entwickelt werden, das alle ausführbaren Dateien in einem Verzeichnis ausführt.
Ein erster Ansatz könnte so aussehen:

```
#! /bin/bash

dir=$1

echo "Executing all in $dir"

for f in $dir/*
do
    echo "Executing $f"
    $f
done
```

Analyse

Im Prinzip funktioniert das Skript bereits: Das Argument 1 der Kommandozeile wird als Verzeichnisname übernommen und in einer `for`-Schleife werden dann alle Dateien darin ausgeführt.
Allerdings gibt es ein paar Probleme:

- Wird das Skript ohne Argumente aufgerufen, so wird versucht, alle Skripte in `/` auszuführen.
- Wird der Verzeichnisname mit `/` am Ende angegeben, erscheint dieser in der Ausgabe doppelt.
- Es wird nicht geprüft, ob das Verzeichnis überhaupt existiert.
- Es wird auch versucht, nicht ausführbare Dateien auszuführen.
- Es findet keine Überprüfung statt, ob die einzelnen Programme erfolgreich liefen.

Prüfen auf Argument

Mit folgendem Code wird ein evtl. am Ende des ersten Argumentes vorhandenes /-Zeichen entfernt und anschließend geprüft, ob der Verzeichnisname leer ist:

```
dir=${1%/}

if [ -z "$dir" ]
then
    echo "Usage: $(basename $0) dirname"
    exit 1
fi
```

(NB: Hierdurch wird auch verhindert, dass das Skript mit / direkt aufgerufen wird!)

Argumente eines Skripts

Beim Aufruf eines Skripts stehen der Befehl, mit dem das Skript aufgerufen wurde, sowie die übergebenen Argumente als Variablen zur Verfügung:

`$0`	Name des Skriptes
`$1`, `$2`, ...	Übergebene Argumente
`$#`	Zahl der Argumente
`$*`	Alle Argumente in *einer* Variablen (immer in `"` einschließen!)

Mit `shift` kann ein Argument aus der Parameterliste entfernt werden.

Testen auf Existenz des Verzeichnisses

Nun wird geprüft, ob das Verzeichnis überhaupt existiert:

```
if [ ! -d "$dir" ]
then
    echo "Directory $dir does not exist"
    exit 2
fi
```

Selektives Ausführen von Dateien

```
for f in $dir/*
do
    [ -d "$f" ] && continue
    [ -x "$f" ] || continue
    echo "Executing $f"
    $f
    ret=$?
    if [ $ret -ne 0 ]
    then
        echo "$f failed with return status $ret"
    fi
done
```

Analyse

- Nun wird in der for-Schleife zunächst geprüft, ob es sich bei der im Verzeichnis befindlichen Datei um ein Verzeichnis handelt, falls ja, wird es übersprungen,
- dann wird geprüft, ob die Datei ausführbar ist, sonst wird sie übersprungen.
- Nach dem Ausführen einer Datei wird nun der Status der Ausführung geprüft. Falls ein Fehler auftrat (Status ungleich 0), wird eine Meldung ausgegeben. Der Ausführungsstatus des letzten Befehles befindet sich dabei jeweils in der Variablen `$?`.

Prüfen auf Optionen in der Kommandozeile

Nun soll das Skript mit einigen Optionen aufgerufen werden können:

```
exec_all [--log=logfile] [--stop-on-error] dirname
```

wobei mit der ersten Option ein Dateiname angegeben wird, in den die Ausgabe bei Ausführen der Programme umgeleitet wird.

Bei Angabe der zweiten Option soll das Skript beendet werden, wenn es beim Ausführen eines Programms zu einem Fehler kam, d. h. alle weiteren Programme in dem Verzeichnis werden nicht mehr ausgeführt.

Realisierung

```
#! /bin/bash

usage() {
    echo "Usage: $(basename $0) [--log logfile] [--stop-on-
}

log=
stop_error=no
while [ "$1" != "${1#-}" ]
do
    option=$1
    case $option in
        -)
            shift
            break
            ;;
```

Realisierung (Forts.)

```
--log=*)
    log=${option#--log=}
    shift
    ;;
--log)
    log=$2
    shift 2
    ;;
--stop-on-error)
    stop_error=yes
    shift
    ;;
```

Realisierung (Forts.)

```
        *)
            usage
            echo "Unknown option $option"
            exit 1
            ;;
    esac
done
```

Realisierung (Forts.)

```
dir=${1%/}

if [ -z "$dir" ]
then
    usage
    exit 1
fi

if [ ! -d "$dir" ]
then
    echo "Directory $dir does not exist"
    exit 2
fi
```

Realisierung (Forts.)

```
echo "Executing all in $dir"

for f in $dir/*
do
    [ -d "$f" ] && continue
    [ -x "$f" ] || continue
    echo "Executing $f"
    if [ -z "$log" ]
    then
        $f
    else
        $f >$log
    fi
```

Realisierung (Forts.)

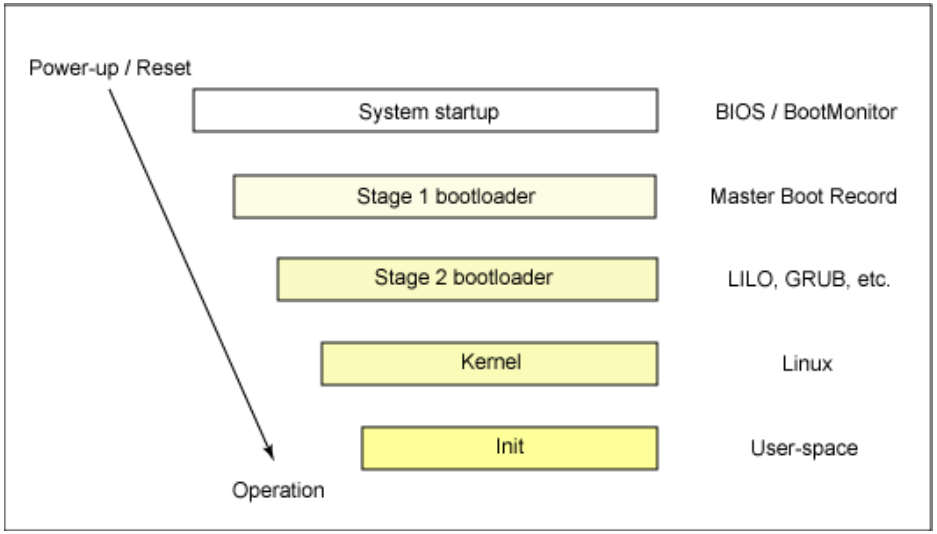
```
    ret=$?
    if [ $ret -ne 0 ]
    then
        echo "$f failed with return status $ret"
        [ "$stop_error" = "yes" ] && break
    fi
done
```

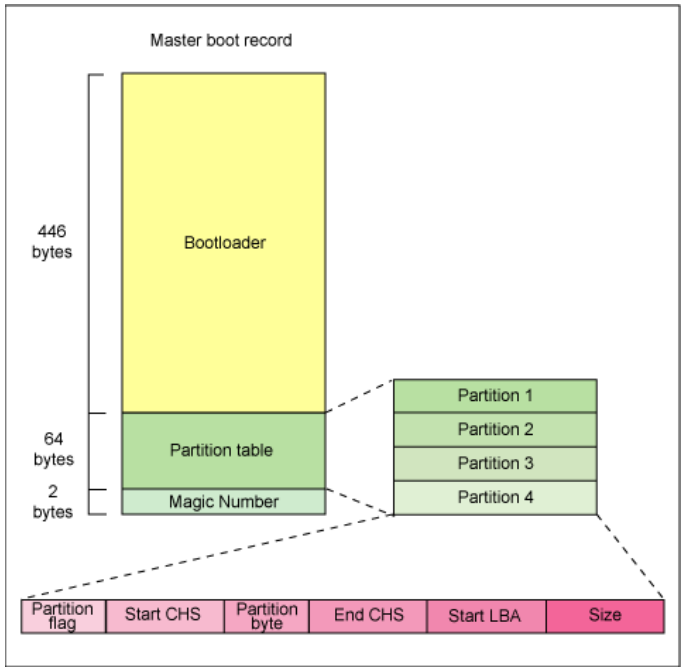
Der Linux-Boot-Prozess

Im Rest der Vorlesung sollen nun die einzelnen Phasen des Boot-Prozesses von Linux betrachtet werden:

1. Systemstart im BIOS
2. Stage 1 bootloader im MBR
3. Stage 2 bootloader
4. Starten des Linux-Kernels
5. Init

Übersicht





Bilder: Tim Jones,

http://www.ibm.com/developerworks/library/l-linuxboot/

Die ersten Stufen

Beim Starten des PC werden die ersten Schritte durch das BIOS des Rechners ausgeführt, dazu gehört insbesondere das Suchen nach einem bootbaren Medium (Festplatte, CD, USB-Stick, früher Diskette) in der im BIOS-Setup festgelegten Reihenfolge.

Die Identifizierung eines Mediums als bootbar erfolgt hierbei durch das Vorhandensein eines gültigen Master Boot Records (MBR) auf dem Medium.

Dieser MBR enthält den Linux-Bootloader der ersten Stufe (Stage 1). Dieser lädt im Wesentlichen den Bootloader Stage 2.

Linux-Bootloader

Die unter Linux verbreiteten Bootloader stellen sowohl den ausführbaren Code für Stage 1 als auch Stage 2 bereit. Der früher benutzte LILO (Linux Loader) wurde hierbei bei den meisten Distributionen inzwischen durch GRUB (GNU Grand Unified Bootloader) ersetzt.

Der im MBR befindliche Teil GRUB Stage 1 beschränkt sich weitgehend auf das Laden des GRUB Stage 2 (ggfs. nach Ausführen von Code, der zum Lesen von großen Festplatten notwendig ist, manchmal als Stage 1.5 bezeichnet).

In GRUB Stage 2 wird (je nach Konfiguration) ein Boot-Menü angezeigt, in dem z. B. zwischen verschiedenen Kerneln (oder zwischen Linux und anderen OS wie Windows) ausgewählt werden kann. Nach Auswahl des gewünschten Kernels wird dieser dann geladen.

Kernel-Phase

Im Allgemeinen liegt der Kernel in einer Image-Datei, die das (meist mit dem bzip-Algorithmus) komprimierte Kernel-Image enthält (zusammen mit dem Code, um das Image in das RAM zu entpacken).

Komplizierter wird es, wenn der Kernel noch einige Kernel-Module benötigt. Da das Dateisystem der Festplatte hier noch nicht zur Verfügung steht, müssen diese Module auf anderem Weg geladen werden. Hierzu wird eine Initial RAM Disk (früher initrd, heute initramfs) verwendet. Diese initrd kann z. B. Treiber enthalten, die der Kernel braucht, um das Dateisystem mit dem eigentlichen Linux-System überhaupt mounten zu können. Innerhalb der Inital RAM Disk befindet sich ein Init-Skript, ein normales Shell-Skript mit Code, der nach dem Laden des initramfs ausgeführt wird.

initramfs ist ein cpio-Archiv, das mit gzip komprimiert wird.

Der Init-Prozess

Nach dem Laden des Kernels werden die Dateisysteme des Systems gemountet (wie in /etc/fstab konfiguriert), dann führt der so genannte init-Prozess Programme aus, bis in den in /etc/inittab eingestellten Runlevel gewechselt wird.

Im vom Administrator gewählten Runlevel werden die für das System gewünschten Dienste gestartet. Neben üblichen Aufgaben wie Konfiguration der Netzwerkschnittstellen und -einstellungen gehören hierzu insbesondere das Mounten bzw. Exportieren weiterer Dateisystem (z. B. über Netzwerk) und das Starten von Services als Dämon, z. B. sshd, httpd.

Übliche Runlevel

Level	Bedeutung
1	Single User
2	Multi User (ohne Netzwerk)
3	Multi User, mit Netzwerk, ohne X11
5	Multi User mit X11

Die Angaben beziehen sich auf Red Hat, Fedora. Bei anderen Distributionen können die Zahlen abweichen.