Vorlesung - Linux Boot

Boot

-> Starten des Rechners
-> Prüfung auf Funktionstüchtigkeit der Hardware
-> untersucht Laufwerke und Massenspeicher
-> Bootsektor vorhanden?
-> Am Ende soll das Linux-Image in den Hauptspeicher geladen werde

Was passiert?

Nach dem Start wird das BIOS oder UEFI gestartet. Die Firmware für das BIOS ist auf eine sogenannte ROM gespeichert, das UEFI hat in der Regel einen eigenen Chip auf der Platine.

-> Nach dem das BIOS/UEFI ausgeführt worden ist, übergibt dieser den Prozess an den Bootloader und dieser wird gestartet.

BIOS (Basic Input Output System):

Sehr vereinfacht:
-> Selbsttest ob grundlegende Komponenten funktionsfähig sind.
-> elektrisches Signal über ein BUS an die Hardware gesendet und erhält ein ACK zurück.
-> Initialisierung der Hardware.
-> ermittelt Taktfrequenz und Betriebsspannung der CPU

Die Abarbeitung des BIOS wird in sogenannten POSTs definiert. POSTs sind fest definierte Abläufe die abgearbeitet werden.

Ein POST für die Initialisierung der Hardware ist wie folgt aufgebaut:

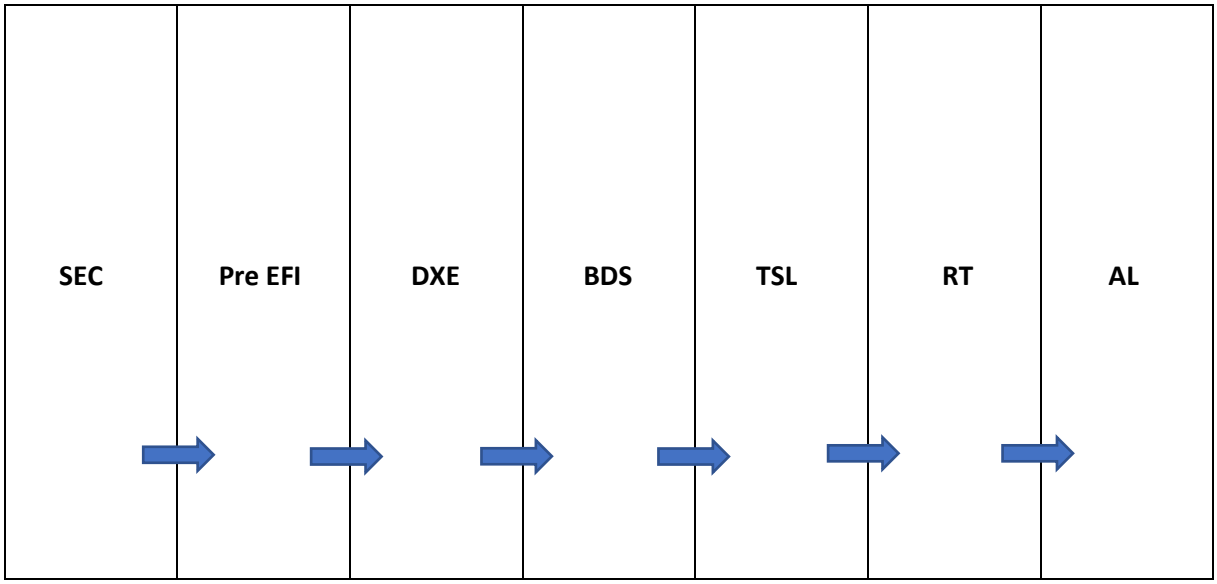
1. CPU überprüft
2. SRAM getestet (flüchtige Speicher, in der die Prüfsumme gebildet wird)
3. Cache geprüft
4. Arbeitsspeicher geprüft
5. Grafikspeicher und Grafik Output gecheckt
6. Netzwerkchips werden getestet
7. Restliche Peripherie wird geprüft

➔ Nachdem alle POSTs abgearbeitet wurden, startet der Bootloeader
-> Beispiele sind Linux-Loader (LILO) &GRUB

UEFI (Unified Extensible Firmware Interface):

-> Schittstelle zwischen OS & Firmware
-> Focus auf 64-Bit Systeme
-> grafische Benutzeroberfläche
-> Secure Boot
-> modular aufgebaut & erweiterbar

UEFI Bootphase



Security (SEC):

-> Name unglücklich gewählt
-> keine Signaturprüfung oder ähnliche Sicherheitstechnische Aspekte
-> initialisiert mit Hardwarespezifischem Code die CPU
    -> wird aus einem Flash-Speicher geladen
-> aktuell existiert in dieser Phase kein Hauptspeicher
    -> aus diesem Grund wird der CPU Cache als Speicher genutzt
-> übergibt am Ende Größe und Speicherort des Caches an die PreEFI Phase

PreEFI (PEI):

-> Hauptspeicher wird initialisiert & Hardware Komponenten, die für die nächste Phase benötigt werden, werden initialisiert
-> Verwendet dafür sogenannte PEIMs (PrefEFI Initialization Module).
    -> stellen APIs zur gegenseitige Kommunikation zur Verfügung
-> Code und daten Sammlung geladen (Firmware Volume Location)
    -> Flashspeicher
    -> Gerätetreiber für nächste Phase
->letzte PEIM übergibt an nächste Phase weiter

Driver Execution Environment (DXE):

-> restlichen Treiber für Hardwarekomponenten werden geladen
-> Protokolle werden registriert
-> keine Protokolle im eigentlichen Sinne, sondern Software von Funktionen & Geräten
-> realisieren z.B. Textausgaben auf Konsole
-> Zugang zu PCI Geräten (Grafikkarte etc.)

Protokollunterscheidung:

a) Boot Service
-> nur verfügbar bevor OS gestartet wird
b) Runtime Service
-> auch zur Laufzeit des OS

Boot Device Select (BDS):

-> startet Bootloader & UEFI wird deaktiviert
-> wechselt sofort ins TLS, bevor Bootloader ausgeführt wird

Transient System Load (TLS):

-> Signaturüberprüfung anhand von Secure Boot (falls aktiv)
-> ExitBootServices() Funktion beendet die Phase & räumt den Hauptspeicher auf

Run Time (RT):

-> hier wird das Betriebssystem laufen

After Life (AL):

-> für ein geordnetes shutdown gedacht
-> wird aber in der Praxis kaum genutzt

➔ Danach wird der Linux Bootloader gestartet

Runtime Services (Zusatzinfo):

Wir haben zwischen zwei unterschiedlichen Protokollen unterschied. Einmal der Boot Service und einmal die Runtime Services.

Hier ist es möglich auch eigene Runtime Services zu schreiben und über UEFI beim Booten oder während der Laufzeit auszuführen. Dafür wird ein spezielles Framework EFI Developer Kit bereitgestellt. Nähere Infos unter: http://x86asm.net/articles/uefi-programming-first-steps/

SysVinit:

Nach dem Bootvorgang ist init der erste Prozess der gestartet wird (meistens mit der PID 1).
-> dienst zum Starten, Beenden und Verwalten von Prozessen.
-> Hier legt man unter anderem fest in welcher Reihenfolge welcher Dienst gestartet wird
Nähere Informationen: https://www.pks.mpg.de/~mueller/docs/suse10.3/opensuse-manual_de/manual/sec.boot.init.html

Openrc:

-> Nachfolger von SysVinit
-> vollständig abwärtskompatible zu SysVinit
-> weitere Infos: https://wiki.gentoo.org/wiki/OpenRC

Systemd:

-> Abwärtskompatible zu SysVinit
-> Abhängigkeiten zwischen Prozessen besser verwaltet
    -> bessere Auslastung beim Systemstart
-> Aufgaben in 4 Kategorien unterteil:
    1. Hardware einrichten
    2. Datenträger einbinden
    3. Sockets anlegen
    4. Daemons starten und verwalten
        -> ähnlicher Aufgabenbereich wie SysVinit

-> Diese liegen in sogenannten Units vor
-> Verwaltung & Konfiguration & Wartung über
    systemctl
-> systemctl sind dafür da, um Dienste zu starten/stoppen, den Timer zu verwalten, Sockets zu verwalten und vieles mehr.
-> Service Dienste können vom Nutzer erstellt werden
-> Service Files liegen in {lib, etc}/systemd/system
    -> in etc liegen die System Service Files
    -> in lib liegen die User Service Files

Wir können unseren eigenen Service Files auch schreiben, diese haben folgenden minimalen Aufbau, der weitaus komplexer wird, wenn man sich mit beschäftigt.

```
[Unit]

Description = <Name des Service Files>

[Service]

Type = simple
ExecStart = /Pfad/zum/Befehl/oder/tool

[Install]

WanteBy = multi-user.target
```

Einige Beispiele für bestimme Typen und Targets, vieles muss man einfach nachlesen und sprengt die Vorlesung zu Linux Boot.

Typen:

simple := default (sind für Dienste die dauerhaft laufen sollen)
forking := Wenn Dienste, weitere Prozesse erzeugen sollen
oneshot := einmalig laufender Dienst

Target:

reboot.target := Service wird nur bei Neustart ausgeführt
poweroff.target := Service wird kurz vom runterfahren gestartet