Bootstrap, Bootup & Multiboot

Vortrag aus dem Seminar

Ausgewählte Komponenten von Betriebssystemen

Lehrstuhl Prof. Paul
Betreuer: Sebastian Bogan

Verena Kremer
27. 9. 2004

0. Inhaltsverzeichnis

1. Einleitung 3

2. Reset 4
   2.1. Reset 4
   2.2. ROM 5
   2.3. BIOS 6
   2.4. Bootfähige Speicher 7

3. Speichermedien 8
   3.1. Übersicht 8
   3.2. Booten von Diskette 8
   3.3. Booten von Festplatten 9
   3.4. Booten über Netzwerke 11

4. Der Betriebssystemstart 13
   4.1. Der Bootloader 13
   4.2. Bootstrap 13

5. Ein Beispiel: der Linux Kernel 2.4 14
   5.1. Übersicht 14
   5.2. Hardware Initialisierungen 15
   5.3. init() 19

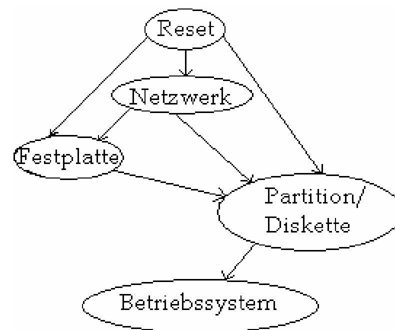
6. Literaturverzeichnis 20

1. Einleitung

Dieses Referat behandelt den Bootvorgang und die dafür benötigten Komponenten eines Rechners. Der Begriff Bootvorgang bezeichnet das Starten des Rechners, das Auswählen eines Betriebssystems und das Starten desselben.

Der Bootvorgang wird durch `Reset` initiiert.

`Reset` tritt zum Beispiel auf, wenn die Stromzufuhr eingeschaltet wird.



`Reset` und die damit zusammenhängenden Vorgänge werden in Kapitel 2 beschrieben.

Im Laufe des Bootvorganges gibt es später drei verschiedene Möglichkeiten, das zu startende Betriebssystem zu finden und zu laden – abhängig davon, wo sich der Code des Betriebssystems befindet:

1.) der Code befindet sich auf einer Diskette, auf die direkt zugegriffen werden kann. Diese Möglichkeit wird in Kapitel 3.2 beschrieben.

2.) Der Code befindet sich auf einer Festplatte.
Dieser Fall wird in Kapitel 3.3 beschrieben.

3.) Der Code befindet sich auf einem anderen Computer. In diesem Fall muss eine Netzwerkverbindung zu diesem Computer aufgebaut werden. Booten über Netzwerk wird in Kapitel 3.4 beschrieben.

In jedem Fall muss der Code des Betriebssystems gestartet werden; dieser Vorgang wird in Kapitel 4 beschrieben.

2. Reset

2.1. Reset

Mit dem Beginn der Stromzufuhr wird `Reset` ausgelöst. Dabei handelt es sich um einen externen Hardware Interrupt.

Ein Interrupt wird in Vorlesungen zum Thema Systemarchitektur oder Betriebssysteme als ein Ereignis (oder ein Signal) definiert, das den normalen Programmablauf (innerhalb des Prozessors) unterbricht. Stattdessen wird eine Interrupt Service Routine (ISR) aufgerufen. Für jeden Interrupt gibt es eine spezifische ISR. Interrupts treten dann auf, wenn ein korrekter Programmablauf nicht mehr möglich oder wünschenswert ist. Alle Interrupts haben eine Priorität. Treten mehrere Interrupts gleichzeitig auf, so werden sie nach Priorität geordnet ausgeführt, es sei denn, der Interrupt mit der höheren Priorität stoppt die weitere Programmausführung. Dann werden Interrupts mit niedrigerer Priorität, die gleichzeitig auftraten, nicht mehr ausgeführt.

Extern bedeutet, dass das Signal nicht innerhalb der CPU (dem eigentlichen Prozessor) ausgelöst wurde. Interrupts können von der Hardware oder von der Software gesteuert (ausgelöst und behandelt) werden, Reset ist hardwaregesteuert. Software gesteuerte Interrupts werden traps oder softirq's genannt und in der Regel vom Betriebssystem verwaltet. `Reset` ist der Interrupt mit der höchsten Priorität: Er wird in jedem Fall ausgeführt.

Das Reset Signal kann auch ausgelöst werden, ohne dass die Stromzufuhr unterbrochen wird. Dieser Fall wird als Warmstart bezeichnet. Den durch den Beginn der Stromzufuhr initiierten Start nennt man Kaltstart.

2.2. ROM

Jeder PC hat neben der CPU und dem Arbeitsspeicher auch ein ROM (Read Only Memory). Dabei handelt es sich um einen Speicherbaustein, aus dem nur Daten gelesen, in den aber keine Daten geschrieben werden können. Die gespeicherten Daten werden bereits bei der Produktion der Bausteine festgelegt und können im Nachhinein nicht mehr oder nur mit speziellen Geräten verändert werden. Das ROM verliert auch bei Spannungsabfall keine Daten.

Sobald Reset gegeben wurde, wird die Reset- ISR gestartet. Sie setzt prozessorinterne Register auf prozessorspezifische Startwerte und versetzt so den gesamten Prozessor in den Startzustand. In der Regel wird anschließend der Inhalt des ROM ins RAM (Random Acces Memory; der Arbeitsspeicher) gespiegelt, da der Zugriff auf Speicherinhalte dort schneller gewährt werden kann. Zuletzt wird der PC (Program Counter, der Befehlszähler) auf den Beginn dieses Codes gesetzt.

Dadurch wird gewährleistet, dass der im ROM befindliche Code als erstes und ohne Voreinstellungen ausgeführt wird.

2.3. BIOS

Im ROM steht der Code des Basic Input Output Systems (BIOS).

Dabei handelt es sich um ein Programm, das die Hardware überprüft und primitive Hardwareunterstützung bereitstellt. Außerdem hat das BIOS die Aufgabe, den Start eines Betriebssystems zu ermöglichen und zu initiieren. Das BIOS führt nach seinem Start zunächst Hardwaretests durch. Diese Tests werden Power On Self Tests (POST) genannt.

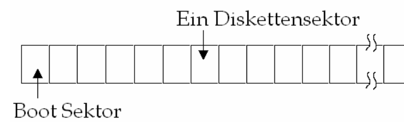
In den meisten Fällen reagiert das BIOS auf einen Warmstart, in dem es die POST's auslässt, um ein schnelleres Hochfahren zu gewährleisten. Man geht davon aus, dass die Hardware nicht beschädigt werden kann, während das Betriebssystem läuft. In einigen Lexika wird der Begriff Warmstart durch dieses Verhalten definiert, das ist falsch, dieses Verhalten ist eine Reaktion auf einen Warmstart.

Während dieser Tests wird nach Hardware gesucht und gefundene Hardware überprüft; außerdem sucht das BIOS nach passenden Treibern. Durch diese Tests können Fehler (etwa eine defekte Festplatte) frühzeitig erkannt werden. Anschließend lädt das BIOS veränderbare Daten wie zum Beispiel Datum und Uhrzeit in einen Speicher. Diese Daten befinden sich auf einem kleinen Speicherbaustein, der durch eine Batterie konstant mit Strom versorgt wird. So können auch während eines Spannungsabfalls im restlichen System keine Daten verloren gehen.

In diesem Speicher sind auch die benutzerdefinierten BIOS-Optionen gespeichert. Während des Startvorgangs kann der Benutzer in ein Konfigurationsmenü gelangen, in dem er diese Dateien teilweise selber manipulieren kann.

2.4. Bootfähige Speicher

Zuletzt sucht das BIOS nach einem bootfähigen Speichermedium. Ein Speichermedium kann zum Beispiel eine Festplatte oder eine Diskette sein.



Jede Diskette ist in mehrere Sektoren unterteilt, von denen jeder einzelne 512 Byte umfasst. Eine Diskette wird genau dann als bootfähig bezeichnet, wenn

1.) von ihr ein vollständiges Betriebssystem geladen werden kann oder
2.) wenn durch den auf ihr gespeicherten Code ein Betriebssystem von einer anderen Quelle geladen werden kann.

Den ersten Sektor einer bootfähigen Diskette nennt man Bootsektor (vgl. Grafik).

Das BIOS liest bei seiner Suche die letzten zwei Byte des ersten Diskettensektors ein.

Es vergleicht die dort gespeicherte Zahl mit einer festgelegten Zahl, die als Magic Number bezeichnet wird und ausschließlich in diesem Fall Verwendung findet. Die Magic Number markiert einen Sektor als Bootsektor und somit eine Diskette als bootfähig.

Bei der Suche nach bootfähigen Speichermedien geht das BIOS nach einer Reihenfolge vor, die in den BIOS-Optionen durch den Benutzer festgelegt werden kann.

Findet das BIOS die Magic Number nicht, so wird ein Fehler angezeigt. Dieses Vorgehen kann beobachtet werden – wenn man das Booten von Diskette in den BIOS-Optionen an erste Stelle gesetzt und eine nicht bootfähige Diskette eingelegt hat. Dann wird ein Fehler auf dem Bildschirm ausgegeben. Der Start kann erst fortgesetzt werden, wenn die Diskette entfernt oder durch ein bootfähiges Medium ersetzt wurde.

3. Speichermedien

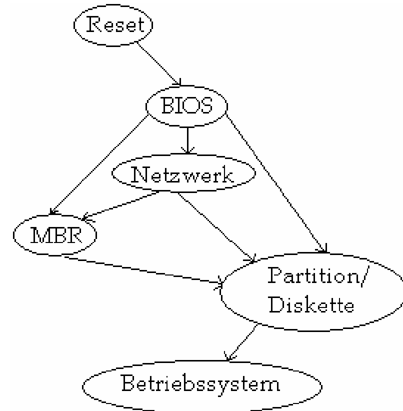
3.1. Übersicht

In diesem Kapitel werden drei Arten bootfähiger Medien beschrieben:

1.) Bootfähige Disketten.

2.) Festplatten mit einer oder mehreren aktiven Partitionen.

3.) Laufwerke oder Speicher anderer Computer.



3.2. Booten von Diskette

Disketten sind nur in Sektoren aufgeteilt (im Gegensatz zu Festplatten). Der erste Sektor einer bootfähigen Diskette ist der Bootsektor, er ist 512 Byte lang und endet mit den 2 Byte der Magic Number (Vgl. Kapitel 1.4; bootfähige Speicher).

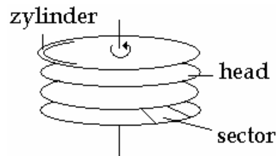
In den übrigen 510 Byte des Bootsektors befindet sich nun ein Programm, das den Start des Betriebssystems initiiert. Dieses Programm wird in Kapitel 4.1 näher beschrieben. Das eigentliche Betriebssystem befindet sich in den restlichen Sektoren der Diskette.

Das BIOS lädt – nach dem Abgleich der Magic Number – die ersten 510 Byte der Diskette in den Arbeitsspeicher (RAM, Random Access Memory).

Im Anschluss daran wird dieser Code ausgeführt.

3.3. Booten von Festplatten

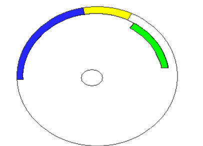
Festplatten sind aufgrund ihrer Größe erheblich stärker unterteilt als Disketten. Als Reaktion darauf läuft auch das Booten von einer Festplatte in mehreren Schritten ab.



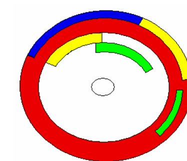
Festplatten sind in Abschnitte unterteilt, die mit den Koordinaten [cylinder, head, sector] angesprochen werden können.

Zusätzlich kann eine Festplatte in Partitionen unterteilt werden; nicht weiter unterteilte Festplatten bestehen aus einer einzigen Partition. Die im BIOS enthaltenen Treiber sind sehr stark minimiert. Daher kann das BIOS nur auf begrenzten Festplattenplatz zugreifen. Die BIOS-Treiber können nur die Zylinder 0 bis 1023 der Festplatte ansprechen und auslesen und erkennen nicht mehr als zwei Partitionen.

Falls der Code des zu ladenden Betriebssystems sich daher nicht innerhalb der ersten beiden Partitionen und innerhalb der ersten 1024 Zylinder befindet, müssen andere Treiber verwendet werden.



Jede einzelne Partition sieht für den Benutzer wie ein eigenes Laufwerk aus. Demzufolge kann auch auf jeder Partition ein Betriebssystem gespeichert werden. Partitionen, von denen ein Betriebssystem geladen werden kann, heißen bootfähige oder aktive Partitionen.



Für die Verwaltung der Partitionen liegt in dem äußersten Zylinder jeder Festplatte der Master Boot Record (MBR). Der Master Boot Record enthält alle Informationen, die für den Bootvorgang benötigt werden.

Er ersetzt in den nun folgenden Vorgängen das BIOS. Beim Booten von einer Festplatte wird das BIOS nicht mehr benötigt, sobald der im MBR enthaltene Code gestartet wurde. Jede Partition funktioniert ähnlich wie eine Diskette: sie beginnt mit einem Bootsektor, in dem sich ein Programm befindet, das vom MBR gestartet wird. Dieses Programm hat die Aufgabe, den Betriebssystemstart zu initialisieren.

Der Master Boot Record

Der MBR enthält von Fabrik aus einen Code, der das weitere Hochfahren steuert. Dieser Code wird Master Boot Code (MBC) genannt. Mit dem MBC kann von Festplatte gebootet werden, wenn nur eine aktive Partition existiert (andere Fälle werden später behandelt). Außerdem enthält der MBR eine Datei, in der alle Partitionen verzeichnet sind; diese Datei wird Partitionstabelle genannt.

Die Partitionstabelle enthält je Partition:

- Boot-Flag (1 Byte)
  Markiert die Partition als aktiv. Das könnte man auch mit einem Bit ausdrücken, in der Tabelle wird dennoch ein Byte freigehalten.
- Kopfnummer des Partitionsbeginns (1 Byte)
- Sektor und Zylindernummer des Boot-Sektors (2 Byte)
- Systemcode (1 Byte)
  Bezeichnet den Typ der Partitionen: NTFS, unformatiert…
- Kopfnummer des Partitionsendes (1 Byte)
- Sektor und Zylindernummer des letzten Sektors der Partition (2 Byte)
- relative Sektornummer des Startsektors (4 Byte)
- Anzahl der Sektoren in der Partition (4 Byte)

Der MBC lädt die Partitionstabelle und wählt den aktiven Sektor aus. Anschließend lädt er den Inhalt des Bootsektors der ausgewählten Partition. Der dort gespeicherte Code wird gestartet und somit der Start des Betriebssystems initiiert.

Multiboot

Wenn die Festplatte mehrere verschiedene Betriebssysteme auf mehreren aktiven Partitionen enthält, ist der MBC überfordert. In diesem Fall wird ein Bootmanager benötigt.

Ein Bootmanager ist eine erweiterte Version des MBC. Er liest die Partitionstabelle ein und gibt über den Bildschirm eine Liste der als aktiv markierten Partitionen aus. Der Benutzer wählt eine dieser Partitionen aus.

Der Bootmanager lädt den Bootsektor der ausgewählten Partition und startet den dort gespeicherten Code.

Moderne Bootmanager enthalten eigene Treiber. Dadurch sind sie nicht auf die eingeschränkten BIOS Treiber angewiesen und können zum Beispiel auch Betriebssysteme starten, die sich auf der dritten oder vierten Partition befinden.

3.4. Booten über Netzwerke

Eine weitere Möglichkeit (neben dem Booten von Disketten und Festplatten) ist das Booten über Netzwerke. Da man für diese Möglichkeit nur ein spezielles BIOS und eine Netzwerkverbindung braucht, ist das Booten über Netzwerk eine Möglichkeit, ein Terminal ohne eigene Laufwerke zu booten.

In diesem Fall enthält das BIOS Treiber für die Netzwerkkarte und die notwendigen Informationen für eine Verbindung, Protokolle, und so weiter. Bei Terminals ohne eigene Festplatte wird das ROM direkt an die Netzwerkkarte angeschlossen. Im Handel werden außerdem Karten mit Steckplätzen für das ROM angeboten.

Anstatt dass während POST (Power On Self Tests, vgl. Abschnitt 2.2) die Festplatte gesucht wird, wird in diesem Fall eine Netzwerkverbindung aufgebaut. Anschließend sucht das BIOS nach einem freigegebenen Laufwerk mit bootfähigem Inhalt.

Das restliche Vorgehen unterscheidet sich nicht von dem Bootvorgang bei ausschließlich lokalen Laufwerken.

Beispiel für einen Bootmanager: GRUB

Zuletzt noch ein Beispiel zu diesem Thema:

GNU GRUB, der GRand Unified Bootloader. GRUB ist der Linux Standard spätestens seit Suse 8.1.

Dere wichtigste Vorteil dieses Bootmanagers ist seine Flexibilität. GRUB verwendet, wie auch andere moderne Bootmanager, eigene Treiber, um nicht auf die Hardwareunterstützung durch das BIOS angewiesen zu sein. Dadurch sind nicht nur die BIOS Beschränkungen außer Kraft gesetzt. GRUB kann mit den eigenen Treibern die kernelspezifischen Formate des zu bootenden Betriebssystems verwenden. So können sehr viele Systeme geladen werden, auch ohne dass die physikalischen Adressen des Codes bekannt sind.

Dadurch wird andererseits das Programm sehr groß und der Code passt nicht mehr im Ganzen in den MBR. Deshalb wurde GRUB in zwei Teile unterteilt ist:

GRUB Stages 1 & 2.

GRUB Stage 1 hat dabei hauptsächlich die Aufgabe, Stage 2 zu laden und auszuführen. Stage 2 befindet sich auf der Festplatte. Da das Programm meist Teil einer Linux Distribution ist, befindet sich Stage 2 in der Regel auf der entsprechenden Linux Partition.

4. Der Betriebssystemstart

4.1. Der Bootloader

In diesem Kapitel wird das Programm beschrieben, das sich im Bootsektor eines bootfähigen Speichermediums befindet. Dieses Programm heißt Bootstrap Loader oder abgekürzt Bootloader. Es ist ein Teil des zu ladenden Betriebssystems; die folgenden Vorgänge sind also Betriebssystem-spezifisch.

Die Aufgabe des Bootloaders besteht darin, die einzelnen Bestandteile des Betriebssystemkerns (Kernel) in vorherbestimmte Positionen im Arbeitsspeicher zu laden und ihren Start zu initiieren. Außerdem stellt er eine primitive Laufzeit-Umgebung bereit, so dass der Kernel, falls nötig, kompiliert werden kann.

Der Bootloader ist meistens zu groß, um komplett in den ersten 510 Byte der Diskette/ Partition gespeichert werden zu können. Daher sind viele Bootloader in zwei Teile unterteilt, von denen einer im Bootblock und der andere auf dem Rest der Festplatte (oder Diskette) gespeichert ist. Hierbei übernimmt der zweite und größere Teil die Aufgabe des Bootloaders, während der erste Teil den zweiten Teil lädt.

Besonders typisch ist diese Vorgehensweise für die Betriebssysteme NetBSD und DOS. Bei DOS werden die beiden Teile des Bootloaders als Bootblock 1 und Bootblock 2 bezeichnet.

4.2. Bootstrap

Als Bootstrap (zu Deutsch Schnürsenkel) wird das Starten des Betriebssystems bezeichnet. Der Bootstrap Loader ist nach diesem Vorgang benannt worden, da er den Bootstrap initiiert.

Der Bootstrap beginnt mit dem Start des Bootstrap Loaders und endet mit dem Start des ersten User Programms.

5. Ein Beispiel: der Linux Kernel 2.4

Im Folgenden wird der Bootstrap anhand eines Beispiels genauer beschrieben. Als Beispiel dient der Linux Kernel 2.4.

Dieses Kapitel erfordert Vorkenntnisse in Betriebssystem Strukturen, da nicht alle Bestandteile des Linux Kernel ausführlich beschrieben werden können, sobald sie während des Bootstrap initialisiert werden. Der Leser sei an dieser Stelle an die Vorträge und Ausarbeitungen anderer Teilnehmer des Seminars verwiesen.

5.1. Übersicht

Die Initialisierung des Kernel läuft in mehreren Schritten ab, die nach einander ausgeführt werden:

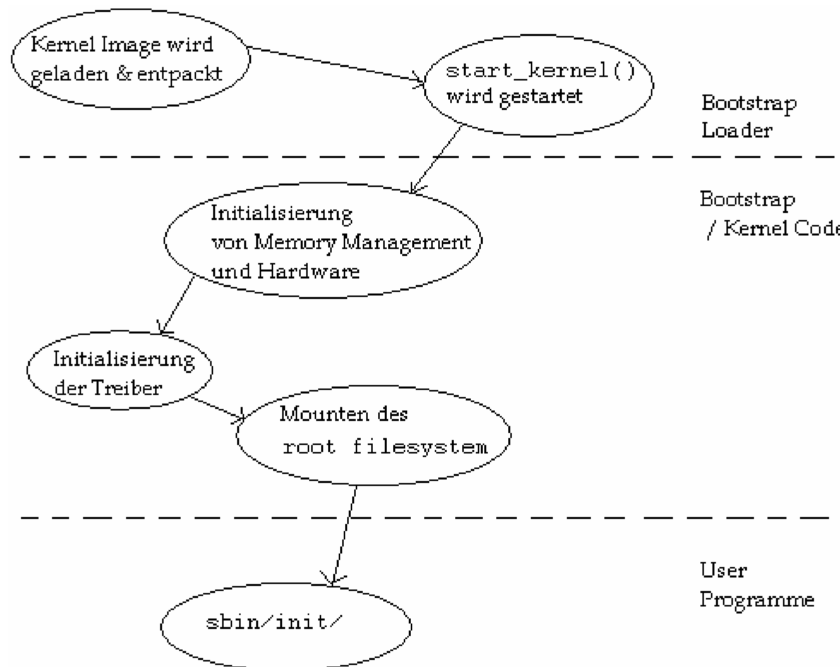
1.) Die Funktion `_stext()` wird von dem Bootloader gestartet. Falls von einer Diskette oder über ein Netzwerk gebootet wird, lädt der Bootloader das Kernel Image in einen Speicher, auf den leichter zugegriffen werden kann (Festplatte oder RAM). Falls von Festplatte gebootet wird, kann dieser Schritt übersprungen werden. Außerdem entpackt die Funktion die Daten, falls dies nötig sein sollte. Externe Interrupts (mit Ausnahme von Reset) können zu diesem Zeitpunkt nicht auftreten.

2.) `_stext()`stellt eine minimale C-Laufzeitumgebung bereit. Zu diesem Zweck initialisiert `_stext()`Interrupt Mask, Status Register und Cache. Schließlich wird die Funktion `start_kernel()` gestartet.
Diese Funktion organisiert die Kernel Initialisierung, indem sie die dafür nötigen Funktionen lädt und startet.

3.) Von `start_kernel()` werden Funktionen gestartet, die die einzelnen Bestandteile des Betriebssystems initialisieren (zum Beispiel die Memory Management Unit). Hierbei werden zuerst die Funktionen gestartet, mit deren Hilfe die hardwarenahen Bestandteile des Kernels initialisiert werden.

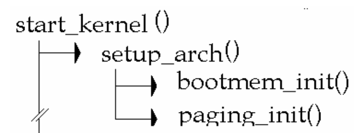
4.) Mit `init()` wird der erste Userprozess, `sbin/init/` aufgerufen.

Die folgende Grafik bietet eine Übersicht über den Linux Bootvorgang:



5.2. Hardware Initialisierungen

1) `Setup_arch ()`



Diese Funktion ist für die ersten maschinenspezifischen Hardware Initialisierungen verantwortlich.

Dazu gehört das Kreieren des `machine vector` für den Host. Der `machine vector` ist eine Datenstruktur, die den Namen des Hosts und Pointer für die maschinenspezifischen input/output Funktionen enthält:

(`hostname, pointer` auf i/o Treiber)

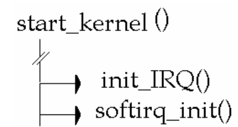
Anschließend sucht die Funktion nach verfügbarem Speicherplatz; sie gibt die Größe und die physikalischen Adressen des Speichers zurück.

Zu diesem Zweck initialisiert `setup_arch()`eine Funktion `bootmem()`, die außerdem den für den Bootprozess nötigen Speicherbereich allokiert.

Schließlich startet `setup_arch()` die Funktion `paging_init(),` um die Memory Management Unit (MMU) des Hosts freizugeben.

2) Software Interrupts - traps

Die Initialisierung der Interrupt Kontrollmechanismen des Betriebssystems werden von den drei Funktionen `trap_init()`, `init_IRQ()` und `softirq_init()` behandelt.



Hierbei übernimmt `trap_init()` die ersten Schritte zur Initialisierung. Dazu mountet die Funktion eine Tabelle: die `interrupt vector table`.

Zu diesem Zeitpunkt sind die Interrupts immer noch ausgeschaltet (disabled). Sie werden freigegeben, sobald die Funktion `calibrate_delay()` aufgerufen wurde.

`init_IRQ()` initialisiert die hardwarespezifische Seite des Kernel Interrupt Subsystems.

Die Funktion `softirq_init()` schließlich initialisiert das `softirq subsystem`.

Die `Softirq's` werden z. Bsp. zur Verbesserung des Interrupthandling bei Netzwerk-prozessen verwendet.

3) Die Funktionen time_init() und console_init()

`time_init()`

Die Funktion `time_init()` initialisiert die Timer - Hardware des Rechners.

Sie installiert den dazugehörigen Interrupt Handler und konfiguriert den Timer, so dass ein periodisch auftretender `system tick` produziert wird. Als Interrupt Handler wird meistens die Funktion `do_timer_interrupt()` verwendet. Nachdem der Interrupt Handler installiert wurde, wird die Frequenz der `ticks` berechnet und der Timer gestartet.

Das folgende Code Beispiel stammt aus der Hitachi SH's Version der Funktion:

```
static struct irqaction irq0
=
{timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
void __init time_init(void)
{  unsigned long interval;
   ...
   setup_irq(TIMER_IRQ, &irq0);
   ...
   interval = (module_clock/4 + HZ/2) / HZ;
   printk("Interval = %ld\n", interval);
   ...
   /* Start TMU0 */
   ctrl_outb(0, TMU_TSTR);
   ctrl_outb(TMU_TOCR_INIT, TMU_TOCR);
   ctrl_outw(TMU0_TCR_INIT, TMU0_TCR);
   ctrl_outl(interval, TMU0_TCOR);
   ctrl_outl(interval, TMU0_TCNT);
   ctrl_outb(TMU_TSTR_INIT, TMU_TSTR);
}
```

`console_init()`

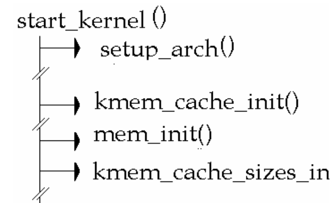
Diese Funktion wird aufgerufen, wenn eine Hardware für die Ausgabe der Konsole konfiguriert wurde. Diese Hardware wird genutzt, um die `kernel boot messages` anzuzeigen, bevor eine komplette (virtuelle) Linux Konsole initialisiert werden kann.

`console_init()` ruft eine hostspezifische Funktion auf, die die Initiierung der Konsole übernimmt. Ein Beispiel für eine solche Funktion wäre `sci_console_init()`, die den Hitachi SH's SCI als Konsolen Hardware nutzt.

In den meisten Fällen ist die Konsolen Hardware der Bildschirm des Host; es kann aber auch ein anderer, beliebiger Ausgang sein, für den das BIOS Treiber bereitstellt.

2) Memory Management

`kmem_cache_init()`



Diese Funktion initialisiert das SLAB Memory Management Subsystem. SLABs werden vom Kernel (ausschließlich intern) für dynamisches Memory Management genutzt.

Die Funktion `mem_init()` gibt eine Tabelle aus und legt einige Variablen fest. In dieser Tabelle werden Adressen und Status des gesamte Speicher aufgeführt, der entweder vom Kernel verwendet wird oder zur Verfügung steht.

Die Funktion `kmem_cache_sizes_init()` beendet die Initialisierung der MMU.

5) `calibrate_delay()`

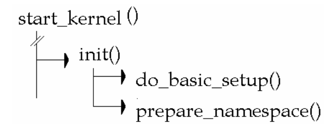
Die Funktion `calibrate_delay()` wird zur Berechnung der `BogoMip` verwendet. Ein `BogoMip` ist eine Zahl, die in der Berechnung interner Delays Verwendung findet.

Sie wird aus der Anzahl der `system ticks` pro Sekunde (`jiffies`) berechnet, die aus dem BIOS ausgelesen werden. Dadurch wird ermöglicht, dass Delays auf verschiedenen Prozessoren ungefähr gleich lange andauern.

Daneben können die `BogoMip` auch zur Geschwindigkeitsmessung benutzt werden (wie schnell ist mein Prozessor?)

5.3. init()

Die Funktion init() ist die letzte Funktion des Bootstrap. Sie beendet das Starten des Betriebssystems, indem sie die Netzwerksysteme des Kernels initialisiert und schließlich den ersten Userprozess startet.



Zu diesem Zweck ruft sie ihrerseits drei Funktionen auf: `do_basic_setup()`, `prepare_namespace()` und `execve()`.

`do_basic_setup()` initialisiert das Netzwerk. Anschließend gibt sie den Speicher frei, der für den Bootstrap allokiert wurde, startet den `scheduling` Prozess und öffnet die Standart `input`, `output` und `error streams`.

`prepare_namespace()` versucht, das `root filesystem` zu mounten.
Das `root filesystem` besteht einerseits aus der Festplatte und andererseits aus der Partition, auf der das root-Verzeichnis liegt.

`execve()` ruft schließlich die Funktion `/sbin/init` auf.

Diese ist der erste Userprozess. Mit dem Start der Funktion `init` ist das Hochfahren beendet, das System läuft (hoffentlich stabil).

Das folgende Code Beispiel stammt aus der Hitachi SH's Version der Funktion `init()`:

```
static int init(void * unused)
{ lock_kernel();
   do_basic_setup();
   prepare_namespace();
   free_initmem();
   unlock_kernel();
   if (open("/dev/console", O_RDWR, 0) < 0)
   printk("Warning: unable to open an initial console.\n");
   (void) dup(0); (void) dup(0);
   if (execute_command)
   execve(execute_command,argv_init,envp_init);
   execve("/sbin/init", argv_init, envp_init);
   execve("/bin/init",argv_init,envp_init);
   execve("/bin/sh",argv_init,envp_init);
   panic("No init found. Try passing init= option to kernel.");
}
```

6. Literaturverzeichnis

W. Almesberger: Booting Linux – the History and the Future

(http://www.almesberger.net/cv/papers/ols2k-9.pdf)

W. Gatcliff: The Linux 2.4 Kernel's Startup Procedure

(http://billgatliff.com/articles/emb-linux/startup.pdf)

Tanenbaum, Woodhull: "Operating Systems – Design & Implementation", 2. edition, Prentice Hall, 1997

Tim Kientzle: Network Booting

(http://www.linux-mag.com/2002-10/netbooting_01.html)

Anonym: Guide to x86 Bootstrapping (and Partitioning)

(http://www.nondot.org/sabre/os/files/Booting/x86Bootstrapping.html)

Das GRUB Manual (http://www.gnu.org/software/grub/manual/grub.html)