Michael Moese, 15.12.2003

Konzepte von Betriebssystemkomponenten

Systemstart und Programmausführung

Wir haben immer über ein Betriebssystem (Linux) gesprochen, aber wie wird es gestartet wenn man den Rechner einschaltet? Und woher kommen die Prozesse von denen immer geredet wird? Im Folgenden wird dies erläutert.

1. Systemstart

Nach dem Einschalten ist ein Rechner praktisch nutzlos, enthält er doch nur zufällige Daten im Speicher, die Hardware ist in einem unbekannten, zufälligen Zustand und es läuft kein Betriebssystem.
Der Systemstart (bootstrap) ist ein umfangreicher Vorgang, bei dem die Hardware getestet und initialisiert wird, ein Teil eines Betriebssystems geladen und zur Ausführung gebracht wird und dieser initialisiert die benötigten Datenstrukturen für den System-Kern, der dann die Kontrolle übernimmt, weitere Datenstrukturen anlegt, ein paar Prozesse startet, und noch etliche weitere Aufgaben wahrnimmt, bis er die Kontrolle an einen Benutzerprozess überträgt und selbst in den Hintergrund tritt.
Es ist offensichtlich, dass dieser Vorgang sehr stark von der verwendeten Rechnerarchitektur und dem Betriebssystem abhängt. Im Folgenden wird dieser Vorgang am Beispiel von Linux erklärt, der sich in fünf Phasen gliedert. Für andere Betriebssysteme und Rechnerarchitekturen sollte sich dieser Vorgang prinzipiell nicht grossartig unterscheiden, lediglich die Details sind hier verschieden.

1.1 Phase 1: Das BIOS

Das BIOS (Basic Input Output System) ist eine Sammlung interrupt-gestützter, hardwarenaher Prozeduren, die im Real Mode des Prozessors ausgeführt werden und für die Interaktion mit der installierten Hardware zuständig sind. Linux ist auf das BIOS zum Start angewiesen; einmal geladen ersetzt es die Prozeduren durch eigene und verlässt sich nicht länger auf das BIOS.
Der Anteil des BIOS lässt sich wiederum in vier einzelne Phasen unterteilen, die nacheinander durchlaufen werden.

Als erstes wird die Hardware einigen Tests unterzogen, dem sogenannten Power On Self Test (POST), um festzustellen, dass auch alle wichtigen Komponenten wirklich funktionieren.

Der zweite Schritt ist die absolut notwendige Initialisierung der Hardware-Komponenten, was in modernen PCI-Systemen unabdingbar ist, da diese so konfiguriert werden müssen, dass sie die benötigten Ressourcen (IRQ's, I/O-Ports) ohne Konflikte benutzen können bzw. sich diese teilen können.

Dann wird, als dritter Schritt, ein zu startendes Betriebssystem gesucht. Die Reihenfolge, in der diese gesucht werden, kann der Benutzer einstellen.

Letztlich, an vierter Stelle, wird von dem Datenträger auf dem ein System gefunden wurde der erste Sektor in den Hauptspeicher geladen und die dort enthaltenen Anweisungen werden ausgeführt (wenn von einer Festplatte gestartet wird ist der Vorgang ein wenig komplizierter, wird unter Phase 2 erklärt).

1.2 Phase 2: Der Bootloader

Der Bootloader wird vom BIOS dazu benutzt, das eigentliche Betriebssystem vom Datenträger in den Speicher zu laden und zur Ausführung zu bringen.

Wird das System von einer Diskette gestartet so wurde der erste Sektor in den Speicher geladen und ausgeführt. Diese enthaltenen Anweisungen laden die restlichen Sektoren in den Speicher die das (komprimierte) Kernel-Image enthalten. Dieses muss komprimiert sein um auf der Diskette Platz zu finden. Der Code zum entpacken des Kernels wird direkt vor das eigentliche Image gespeichert (zu finden unter arch/i386/boot/bootsect.S). Beginnend mit dem ersten Sektor wird dieses Image mit dem Code zum Entpacken ab dem 1. Sektor auf die Diskette geschrieben.

Wird das System von einer Festplatte gestartet sieht der Vorgang ein wenig anders aus. Im ersten Sektor einer Festplatte (dem Master Boot Record MBR) befindet sich die Partitionstabelle und ein kleines Programm das den ersten Sektor derjenigen Partition lädt und ausführt, die das Betriebssystem enthält. Einige Betriebssysteme (Windows 98, MSDOS,.. benutzen ein Active-Flag das die Start-Partition kennzeichnet -> nur diese Partition kann gestartet werden). Im Falle von Linux wird dieses kleine Programm durch ein umfangreicheres ersetzt, z. B. LILO oder GRUB, welche dem Benutzer mehrere Möglichkeiten einräumen, das System wird flexibler zu starten.

1.3 Phase 3: Die setup()-Funktion

Beim kompilieren des Kernels legt der Linker die setup()-Funktion direkt hinter den im Kernel integrierten Bootloader ab, so kann dieser die Funktion leicht finden und ausführen.
Sie muss die Hardware initialisieren (Linux verlässt sich nicht auf die Initialisierung durch das BIOS), die Ausfühungsumgebung für den Kernel vorbereiten und benötigte Datenstrukturen anlegen. Neben der Initialisierung von Grafik-Karte, Controller, und einigen anderen Geräten wird die Grösse des Hauptspeichers ermittelt, APM-Unterstützung, eine Maus und die Tastatur initialisiert.
Weiterhin legt setup() eine provisorische Interrupt Descriptor Table (IDT) und eine Global Descriptor Table (GDT) an und der Interrupt-Controller wird umprogrammiert (die 16 Hardware-Interrupts werden auf die Vektoren 32 bis 47 gelegt, da die Leitungen 0 bis 15 für CPU-Exceptions reserviert sind. Bevor zur startup_32()-Funktion gesprungen wird, wird die CPU noch in den Protected Mode umgeschaltet, Paging bleibt aber hier noch deaktiviert.

1.4 Phase 4: Die startup_32()-Funktionen

Es existieren zwei verschiedene startup_32()-Funktionen, die nacheinander ausgeführt werden, was jedoch kein Problem darstellt da sie direkt durch einen Sprung an ihre Anfangs-Adresse aufgerufen werden. Die Namensgebung ist nur ein wenig verwirrend.
Die erste der beiden Funktionen findet sich unter arch/i386/boot/compressed/head.S . Diese richtet die Segmentierungs-Register und einen provisorischen Stack ein und initialisiert ungenutzten Kernel-Speicher mit Nullen.
Sie ist es auch die das Kernel-Image dekomprimiert und an seine endgültige Position im Hauptspeicher verschiebt.

Die zweite startup_32()-Funktion bereitet die Ausführungsumgebung des ersten Prozesses vor. Die Segmentierungsregister werden mit den finalen Werten geladen, es wird ein Kernel Mode Stack für Prozess 0 eingerichtet. Weiterhin werden provisorische Page Tables eingerichtet und Paging wird aktiviert. Die IDT wird mit Null-Interrupt-Vektoren gefüllt, die vom BIOS erhaltenen Informationen und die dem System übergebenen Parameter werden in die erste Speicherseite gesichert. Nun wird noch das Prozessor-Modell identifiziert und dann wird zur start_kernel()-Funktion gesprungen.

1.5 Phase 5: Die start_kernel()-Funktion

Mit der Initialisierung der Page Tables, der Page Descriptors, der IDT, des Slab Allocator und der Systemzeit (und des Datums) ist die Initialisierung des Kernels abgeschlossen.
Nun wird ein Thread für Prozess 1 angelegt und /sbin/init wird ausgefürht.
Nun folgen viele weitere Meldungen bis der gewohnte Login-Prompt erscheint und das System einsatzbereit ist.

2. Programmausführung

2.1 Überblick

Bisher wurde immer über Prozesse geredet, aber nie etwas darüber gesagt woher diese überhaupt kommen. Ein Prozess ist ja viel mehr als nur die Daten und die Anweisungen. Es wäre eine Kleinigkeit für das Betriebssystem die Anweisungen vom Datenträger zu lesen und sie auszuführen. Jedoch gibt es viele verschiedene Binärformate (zum Beispiel a.out, ELF, MSDOS EXE,..), shared Libraries und die im Programm enthaltenen Informationen zum Ausfühungskontext.

Eine ausführbare Datei ist nichts anderes als eine Anleitung für das System wie der neue Ausgführungskontext für einen neuen Prozess anzulegen ist.

Programme liegen in sogenannten ausführbaren Dateien, Binärdateien die sowohl die Anweisungen als auch die Daten eines Programmes beinhalten. Beim Start kann der Benutzer dem neuen Prozess auf zwei Arten Informationen mitgeben: entweder durch Kommandozeilenparameter oder durch Umgebungsvariablen.

Soll ein neues Programm gestartet werden erzeugt die Shell einen neuen Prozess der das Programm ausführt (vgl. Systemprogrammierung 1, Aufgabe 4 und 5). Die exec-Systemaufrufe ermitteln das Binärformat der ausführbaren Datei, laden diese und passen den Ausfühungskontext an. Danach beginnt die Ausfühung der Anweisungen aus der Datei.
Mit der Änderung des Kontextes ändern sich auch einige andere Parameter: der neue Prozess erhält neue Argumente, eine neue Shell-Umgebung, einen neuen Adressraum, u.U. veränderte Privilegien, nur die PID und die Datei-Deskriptoren bleiben erhalten.

2.2 Ausführung eines Programmes

Ein Programm wird durch den Aufruf einer der exec()-Funktionen gestartet wie bereits aus Systemprogrammierung 1 bekannt ist. Alle diese Funktionen sind nur Wrapperfunktionen für die execve(). Sie erhält als Parameter einen Zeiger auf den Dateinamen mit Pfad, ein Pointer-Array auf Strings für die Parameter, und ein weiteres Array für die Umgebungsvariablen.
Bei einem Aufruf wird zunächst die auszuführende Datei und der zugehörige Inode gesucht, dann wird überprüft ob die Datei gerade geschrieben wird, die Zugriffsrechte werden überprüft um sicherzustellen dass die Datei auch wirklich ausführbar ist, die Benutzer- und Gruppen-ID wird gesetzt und letztlich werden die ersten 128 Byte der Datei geladen die die Magic Number enthalten, anhand derer das Format bestimmt werden kann. Daraufhin werden die Programmsegmente und die Abhängigkeiten von Shared Libraries analysiert.
Jetzt wird der passende Programm-Interpreter gesucht, dessen Zugriffsrechte überprüft und dieser wird, ähnlich einem normalen Programm, geladen, zumindest die ersten 128 Byte, und seine Magic-Number wird wiederum identifiziert. Der Prozess wird nun in eine neue Thread-Gruppe verschoben, es werden einige Felder im Process-Descriptor angepasst, und man erreicht den Punkt ab dem eine Rückkehr ins aufrufende Programm nicht mehr möglich ist. Dann werden neue Speicherbereiche angelegt für den Stack, das Text- und das Datensegment des Programms und diese werden per Memory Mapping eingeblendet. Nun wird der Programm-Interpreter geladen, der einige benötigte Datenstrukturen anlegt.
Dann ist das Programm fast bereit zur Ausführung. Der Programminterpreter muss sich nur noch um die Dynamischen Bibliotheken kümmern indem er alle Referenzen auf Funktionen anpasst entsprechend der Speicherbereiche der Bibliothek.
Dann wird zum Einstiegspunkt des Programmes (die main()-Funktion in C)

gesprungen, und fortan der Code des Programms und der Bibliothek ausgeführt.

2.3 Formate ausführbarer Dateien

Es gibt etliche verschiedene Formate für ausführbare Dateien, darunter das moderne ELF (Executable and Linking Format), das in Linux, Solaris 2 und System V R4 benutzt wird, den alten Standard a.out (Assembler OUTput format) von dem es viele verschiedene Versionen gab und das heute kaum noch benutzt wird, sowie viele andere wie MSDOS-EXE, BSD Unix's COFF und Plattformunabhängige wie Java.
Die Beschreibung eines Binärformats umfasst unter Linux drei Methoden:

- Laden der Binärdatei und Aufsetzen eines Ausführungskontextes
- Dynamisch gelinkte Bibliotheken zu laufendem Prozess hinzufügen
- Ausführungskontext in eine Datei schreiben (Core Dump)

Diese Formate werden von Linux in einer verketteten Liste abgelegt, so können zur Laufzeit beliebige Formate hinzugenommen oder entfernt werden. Linux kann mit allen ausführbaren Dateien nativ umgehen die POSIX-Systemaufrufe verwenden, für andere (wie z.B. Windows-EXE wird aufgrund des unterschiedlichen API's) ein Interpreter benötigt.
Die Ausführung anderer Formate setzt neben der POSIX-Systemaufrufe (bzw. dem Vorhandensein eines Interpreters) auch vorraus, dass die ausführbaren Dateien auf derselben Prozessorarchitektur ausgeführt werden, für die sie erzeugt wurden.

2.4 Programmsegmente

Ein Programm teilt sich in mehre Segmente auf, und zwar in ein Text-Segment das den ausführbaren Code enthält, ein Segment für die initialisierten Daten (statische Variablen, globale Variablen mit Anfangswert), das Segment für die uninitialisierten Daten (uninitialisierte globale Variablen) und das Stack-Segment für lokale Variablen, Rücksprung-Adressen, Aufruf-Parameter und so weiter. Wie diese initialisiert werden steht in der ausführbaren Datei.

2.5 Bibliotheken

Eine Quellcode-Datei wird in mehreren Schritten zu einer Objektdatei transformiert. Diese kann jedoch nicht direkt ausgeführt werden, da sie nicht aus einen linearen Adressraum besteht. An dieser Stelle übernimmt der Linker, bindet Bibliotheksfunktionen ein und bindet sie mit der Objektdatei zu einer ausführbaren Datei zusammen. Selbst Systemaufrufe werden über sog. Wrapper-Funktionen aus der C-Bibliothek aufgerufen. Früher kannte man nur statische gelinkte Programme, so dass jedes Programm Teile der Bibliothek enthielt. Dadurch wird u. U. sehr viel Platz verschwendet.

Durch dynamisch gelinkte Bibliotheken (shared Libraries) wird dieses Problem umgangen, denn dynamisch gelinkte Programme enthalten statt dem Objektcode nur Referenzen auf Funktionsnamen aus den Bibliotheken. Beim Start eines Programmes untersucht der Programm-Interpreter die ausführbare Datei nach diesen Verweisen und sucht die entsprechenden Bibliotheken. Diese, sofern noch nicht geladen, lädt er in den Speicher. Oder ein Prozess lädt selbst eine benötigte Bibliothek, bei Bedarf zur Laufzeit mit einem Systemaufruf.
Dieser Mechanismus bietet sich für Systeme an die “File Memory Mapping” beherrschen (hierbei wird der Speicherbereich des Objectcodes in den Adressraum des Prozesses eingeblendet und es muss kein Code kopiert werden).
Allerdings erkauft man sich diese Vorteile durch etwas längere Startzeiten für Programme, da der Programm-Interpreter jede Datei untersuchen muss. Ausserdem könnten dynamisch gelinkte Programme auf anderen Systemen die eine andere Version einer Bibliothek installiert haben zu Kompatibilitätsproblemen führen.

2.6 Ausführungsüberwachung

Die Ausführungsüberwachung erlaubt es Programmen andere Programme zu überwachen, was z. B. für Debugger absolut notwenig ist. Es können Ereignisse wie das Ende der Ausführung eines Maschinenbefehls, Beginn und Ende eines Systemaufrufs und der Erhalt eines Signals überwacht werden. Tritt eines der Ereignisse ein wird die Ausführung des überwachten Programmes unterbrochen und der Vater erhält ein SIGCHLD-Signal auf das er angemessem reagieren kann. Weiterhin wird eine Einzelschritt-Ausführung unterstützt, um ein Programm schrittweise abzuarbeiten.

Literaturverzeichnis

1. Bovet D.P., Ceasti M., Understanding the Linux Kernel, Second Edition, O'Reilly & Associates
2. Wettstein, H., Architektur von Betriebsystemen, 3. Auflage, Hanser Studienbücher
3. Bach, M. J., The Design of the UNIX Operating System, Prentice-Hall