Virtualisierung

Teil 2: Direct Execution

Prof. Dr.-Ing. Andreas Heil

v1.0.0

Lernziele und Kompetenzen

- **Verstehen** wie Prozesse im Betriebssystem gesteuert werden.
- **Verstehen** welche Probleme bei der direkten Ausführung von Prozessen auf der CPU entstehen und wie dem im Betriebssystem begegnet wird.

Problem

Bisher haben wir gelernt, dass es Prozesse gibt, diese in irgendwelchen Listen stehen und Prozesse gelanden werden können.

Das Betriebssystem lädt also ein Programm, lädt alle Register und startet den Prozess...

- **Frage 1:** Wie stellen wir sicher, dass der Prozess nichts »Verbotenes« tut?
- **Frage 2:** Die direkte Ausführung des Prozesses auf der CPU (engl. direct execution) ist zwar schnell, aber was passiert nun, wenn der Prozess eingeschränkte Aktionen durchführen will (z.B. mehr Speicher, I/O-Operation auf Disk etc.)?
- **Frage 3:** Und wie stellen wir überhaupt sicher, dass der Prozess die Kontrolle wieder abgiebt? Solange der Prozess ausgeführt wird, wird ja nicht das Betriebssystem ausgeführt... 🤔

Lösungsidee

Programme laufen im sog. **»User Mode Linux«** oder allgemein »«User Mode«

- Es wird eingeschränkt, was das Programm »tun« kann
- Z.B. werden I/O Operationen eingeschränkt
- Wenn doch, wird eine »Exception« im Prozessor erzeugt (das heißt tatsächlich so, hat aber nichts mit Java Exceptions zu tun)

Der Gegensazu: **»Kernel Mode«**

- Hier sind alle Operationen, auch bzw. insbesondere I/O-Operationen erlaubt

System Call

Wenn ein Programm im *User Mode* etwas machen möchte, das eigentlich nicht darf, führt es einen sog »System Call« oder kurz »Syscall« aus.

- System Calls werden von allen modernen Betriebssystemen angeboten
- POSIX-Systeme (Portable Operating System Interface^1) bieten mehrere hundert solcher System Calls an

System Call Ablauf

- Das Programm führt ein sog. Trap-Instruktion aus
- Springt in Kernel, und startet im privilegierten Modus (Kernel Modus)
- Führt die Operationen aus, die im »System Call Handler« hinterlegt sind
- Führt eine sog. Return-From-Trap-Instruktion aus
- Kehrt in den User Mode zurück

Vorsicht

Die Hardware muss darauf achten „genügend Bestandteile vom Programm bestehen zu lassen“, so dass es später wieder ausgeführt werden kann.

Am Beispiel des x86:

Hier werden...

- Program Counter, Flags und weitere Register in einen sog. Per-Process-Kernel-Stack »gepusht« (Datenstruktur Stack klar? Ggf. Exkurs am Ende)
- Bei der Return-From-Trap-Instruktion werden diese wieder vom Stack geladen
- Danach kann das Programm wieder im User Mode ausgeführt werden

Dieses Vorgehen wird von Betriebssystem zu Betriebssystem zwar unterschiedlich gehandhabt, ist im Grundsatz aber immer ähnlich

Nochmal Vorsicht

Frage: Woher weiß das OS, welcher Code für System Calls ausgeführt werden soll?

Das Programm kann ja kein Speicherbereich angeben

Grundsätzlich wäre das auch eine sehr schlechte Idee… Das ist schon klar warum , oder?

Trap Table

Lösung:

- Es wird eine sog. »Trap Table« zur Boot-Zeit erstellt
- Beim Booten ist das System immer im Kernel Mode
- Das Betriebssystem kann der Hardware somit sagen, welcher Code bei welchem Ereignis ausgeführt wird
- Das Betriebssystem informiert die Hardware über diese sog. Trap Handlers oder System Call Handlers

Nur mal so... Was könnte man denn machen, wenn man eine eigene Trap Table installieren könnte? 🤔

Zusammenfassung

- Prozesse direkt (d.h. ohne Kontrolle) auf der Hardware auszuführen, ist keine gute Idee
- Prozesse weden im User Mode ausgeführt und sind eingeschränkt was bestimmte Aktionen aneht
- Mittels System Calls kann ein Prozess spezielle Aktionen ausführen (lassen), die jedoch vom Betriebssystem kontrolliert werden
- Eine Trap Table enthält die Information darüber, wo der Code steht, der durch ein System Call ausgeführt wird
- Trap Tables werden zur Bootzeitr (im Kernel Modus) erzeugt