Konzepte von Betriebsystem-Komponenten (KVBK)
Schwerpunkt Linux

Interrupts, Softirqs, Tasklets, Bottom Halves

Seminar am 10.11.2003, Referent: Thomas Engelhardt

1.) Einleitung

Interrupts sind Ereignisse, die der CPU signalisieren, dass der Auslöser sie für dringende Aufgaben in Anspruch nehmen möchte.
Der Vortrag soll die wesentlichen Grundzüge aufzeigen, wie Interrupts in der 80x86 Architektur auf der Hardwareebene behandelt werden und wie das Betriebsystem, in diesem Fall Linux, darauf reagiert.

2.) Klassifizierung

Um differenziert über Interrupts sprechen zu können, müssen diese zunächst klassifiziert werden. Dabei unterscheidet man synchrone Interrupts (Exceptions) und asynchrone Interrupts (Interrupts der E/A-Geräte, im Folgenden nur Interrupts genannt).

2.1) Exceptions

Exceptions können zum einem durch anormale Zustände in Folge eines Programmierfehlers entstehen oder bewusst vom Programmierer ausgelöst werden.

Processor Detected Exceptions werden durch anormale Zustände verursacht. Diese Art der Exceptions teilt man wiederum in drei verschiedene Klassen ein.

- *Faults:* In der Regel korrigierbare Fehler. Die Ausführung kann mit dem Befehl weitergeführt werden, der die Exception ausgelöst hat, sofern der Fehler korrigiert werden konnte.
- *Traps:* Werden von speziellen Befehlen ausgelöst, die unter anderem mit dem Debugging in Verbindung stehen. Nach der Behandlung der Trap wird das auslösende Programm normal fortgesetzt.
- *Aborts:* Schwerwiegender Fehler (Hardwarefehler oder inkonsistente Systemtabellen), der nicht korrigiert werden kann. Der betroffene Prozess wird in der Regel terminiert.

Programmed Exceptions werden bewusst vom Programmierer aufgerufen. Unter anderem werden damit Systemaufrufe realisiert. Behandelt werden sie ebenso wie *Traps*.

2.2) Interrupts

Zwischen Interrupts unterscheidet man maskierbare und nichtmaskierbare Interrupts.

Maskierbare Interrupts werden von E/A-Geräten ausgelöst. Im maskierten Zustand werden die Signale so lange von der Kontrolleinheit ignoriert, bis die Maskierung wieder aufgehoben wird.
Nichtmaskierbare Interrupts sind nur wenigen kritischen Ereignissen, wie Hardwarefehlern, vorbehalten.

3.) Interrupts auf der Hardwareebene

Damit das Betriebsystem auf Interrupts reagieren kann, muss der Prozessor über das Auftreten eines Interrupts informiert werden. Diese Aufgabe erledigt der *Programmable Interrupt Controller* (kurz *PIC*), bzw. in neueren und Multiprozessor-Systemen der *Advanced Programmable Interrupt Controller* (kurz *APIC*).

Jedes Gerät, das einen Interrupt auslösen kann, ist mit dem PIC, respektive dem APIC, verbunden. Der PIC stellt dabei 15 eingehende Interrupt Leitungen, der APIC insgesamt 24, zur Verfügung. Diese Leitungen werden ständig vom (A)PIC überwacht. Stellt der Controller einen Interrupt fest, konvertiert er ihn in eine Zahl zwischen 0 und 255, auch Vektor genannt. Im Anschluss wird dem Prozessor mitgeteilt, dass ein Vektor gelesen werden kann. Die CPU bestätigt den Empfang und der (A)PIC setzt die Überwachung der Eingangsleitungen fort. Die weitere Bearbeitung des Interrupts ist Angelegenheit des Prozessors.

Die CPU stellt zuerst den Wert des eben empfangenen Vektors fest. Anschließend ermittelt sie den Deskriptor in der *Interrupt Deskriptor Tabelle (→ 3.1. IDT)*, der an der Stelle steht, die dem Wert des Vektors entspricht.
Danach wird der Inhalt der `eflags`, `cs` und `eip` Register auf dem Stack gespeichert, um nach der Interrupt-Behandlung an der unterbrochenen Stelle fortfahren zu können. Zuletzt startet die CPU den *Interrupt-Handler* (Programmcode zur Bearbeitung eines Interrups), dessen Adresse aus dem vorher ermittelten Deskriptor gelesen wird. Nach der Terminierung des Interrupt-Handlers werden die vorher auf dem Stack gespeicherten Register wieder geladen, um die unterbrochene Ausführung wieder aufzunehmen.

3.1) IDT – Interrupt Deskriptor Tabelle

Die *Interrupt Descriptor Tabelle* stellt im Prinzip die Schnittstelle zwischen der Hardware und dem Betriebsystem dar. Sie speichert ebenso viele Deskriptoren, wie es Vektoren gibt. Damit enthält die IDT 256 Einträge. Jeder Eintrag speichert einen Deskriptor, über den die CPU die Adresse des Interrupt-Handlers ermittelt, um ihn zu starten; d. h. die IDT verknüpft die Vektoren mit ihren entsprechenden Handlern. Insgesamt existieren in der IDT drei Typen von Deskriptoren.

Task Gate:
Das Task Gate enthält einen TSS Selektor, der auf den Prozess verweist, der den Interrupt behandeln und den aktuell laufenden Prozess ersetzen muss. In Linux finden Task Gates allerdings keine Verwendung, da der Umschaltprozess in den entsprechenden Prozess zu lange dauern würde.

Interrupt Gate:
Das Interrupt Gate speichert die Adresse des Interrupt- oder Exception-Handlers. Außerdem veranlasst dieser Deskriptor die Löschung des IF-Flags (Interrupt Flag, Teil des `eflag` Register), um damit alle maskierbaren Interrupts zu deaktivieren.

Trap Gate:
Das Trap Gate ist mit dem Interrupt Gate identisch, bis auf die Tatsache, dass der Prozessor bei einem Aufruf eines Trap Gates das IF-Flag unberührt lässt.

Die Initialisierung der IDT muss vor der Aktivierung der Interrupts erfolgen. Wird der Computer gestartet, initialisiert das BIOS die IDT. Damit stehen BIOS-eigene Interrupt Handler zur Verwendung bereit. Sobald Linux die Kontrolle über die IDT übernimmt, werden alle Einträge zuerst mit einem Null Handler überschrieben. Im zweiten Schritt trägt der Kernel sinnvolle Interrupt und Trap Gates ein. Interrupt Gates finden dabei ausschließlich für Interrupts Verwendung, Trap Gates dagegen für Exceptions. Die ersten 32 Vektoren sind dabei für Exceptions reserviert. Die Belegung der Exceptions ist fest von dem

Prozessordesign vorgegeben. Welcher Vektor mit welchem E/A-Gerät korrespondiert, wird mit Hilfe der Gerätetreiber ermittelt.

4.) Interrupt und Exception Handling in Linux

Generell lassen sich die Behandlungsroutinen der Interrupts und Exceptions von Interrupts bzw. Exceptions unterbrechen. Damit wird gewährleistet, dass möglichst viele Interrupts bzw. Exceptions pro Zeitintervall bearbeitet werden.
Im Folgenden soll näher betrachtet werden, welche Verfahren Linux anwendet, um auf Interrupts und Exceptions zu reagieren.

4.1) Interrupts

Die Strategie, wie Linux Interrupts behandelt, ist von der Art des Interrupts abhängig.
Dabei existieren drei verschiedene Typen: *I/O Interrupts*, *Timer Interrupts* und *Interprocessor Interrupts*.

4.1.1) I/O Interrupts:

Zur Behandlung von I/O Interrupts startet der Handler eine *Interrupt Service Routine (ISR).* Die Aufgabe der *ISR* ist es, auf die Anforderungen des Interrupt auslösenden Gerätes zu reagieren und es zu bedienen, d. h. die *ISRs* sind gerätespezifisch und müssen daher vom Gerätetreiber zur Verfügung gestellt werden. Interrupt Handler müssen nach sehr kurzer Zeit wieder terminieren, da die unterbrochene Befehlsfolge sonst zu lange blockiert bleibt. Aus diesem Grund darf die ISR keine zeitaufwendigen Aufgaben ausführen. Für besonders zeitkritische Funktionen, wie das Programmieren des PIC, bleiben maskierbare Interrupts deaktiviert. Für nichtzeitkritische Funktionen, z. B. das Lesen des Scancodes der Tastatur, werden die maskierbaren Interrupts wieder aktiviert, so dass der Handler von anderen Interrupt-Signalen unterbrochen werden kann. Für besonders aufwendige Aufgaben muss die ISR verzögerbare Funktionen (→ *6. Softirqs, Tasklets, Bottom Halves*) aktivieren, damit die unterbrochene Befehlssequenz nicht zu lange blockiert bleibt.

Das Interrupt-Handling muss daneben auch sehr flexibel sein, da die PCI Spezifikation *IRQ Sharing* vorsieht, d.h. zwei oder mehrere Geräte dürfen sich eine IRQ Leitung teilen. Darüber hinaus ist in Linux auch ein Verfahren zur dynamischen IRQ Vergabe an Geräte, die kein *IRQ Sharing* unterstützen, implementiert (*IRQ dynamic allocation*)

IRQ Sharing: Der Handler greift über eine Kernel interne Datenstruktur, von der für jeden Vektor genau eine existiert, auf eine Liste von Interrupt Service Routinen zu.
Da keine Möglichkeit existiert, herauszufinden, welches Gerät den Interrupt ausgelöst hat, werden alle ISRs sequenziell ausgeführt. Das bedeutet für die ISRs, dass sie überprüfen müssen, ob das Gerät, das sie bedienen, eine Anforderung stellt. Das bedeutet wiederum, dass Gerät und Gerätetreiber *IRQ Sharing* fähig sein müssen.

IRQ Dynamic Allocation: Eine IRQ Leitung wird erst in dem Moment an das Gerät vergeben, wenn sie benötigt wird. Dabei fordert der Gerätetreiber eine IRQ Leitung an. Ist diese vergeben, wird ein Fehlersignal generiert, auf das der Treiber reagieren muss. Ist die Leitung frei, bekommt sie das Gerät zugewiesen und nach erfolgter Operation wird die Leitung wieder freigegeben.

4.1.2) Timer Interrupts:

Timer Interrupts werden in der Regel als I/O Interrupts behandelt und haben die Aufgabe, den Kernel über ein vergangenes Zeitintervall zu informieren. Ausgelöst werden diese Interrupts von dem *Programmable Interval Timer (PIT)* und der *Real Time Clock (RTC).* Die *PIT* ist zum Beispiel an die Interrupt-Leitung 0 angeschlossen und wird wie andere I/O Interrupts von einer ISR bearbeitet.

4.1.3) Interprocessor Interrutps:

Interprocessor Interrupts sind dazu da, Nachrichten von einer CPU an andere CPUs zu schicken. Eine solche Nachricht kann zum Beispiel ein Vektor sein, der die anderen CPUs auffordert, eine Funktion auszuführen, die von der Sender-CPU übergeben wird.

4.2.) Exception Handling in Linux

Exceptions werden in Folge eines anormalen Zustandes in der CPU ausgelöst, z. B. Division durch Null, und können nicht deaktiviert werden.
Über die IDT wird der Exception Handler gestartet. Im Anschluss wird immer überprüft, ob die Exception im User Mode oder im Kernel Mode ausgelöst wurde. Die einzige Exception, die im Kernel Mode auftreten darf, ist die Reaktion auf ein ungültiges Argument, das einem Systemaufruf übergeben wurde. Der Kernel wurde so sorgfältig programmiert, dass jeder andere Fall als ein Fehler im Kernel aufzufassen ist, der keinem Prozess zugewiesen werden kann. Infolgedessen wird die *die()* Funktion aufgerufen, die den Kernel terminiert. Tritt die Exception im User Mode auf, wird ein UNIX-Signal an den auslösenden Prozess geschickt, der mit seinem eigenen Signal-Handler darauf reagieren muss. Fehlt der Signal-Handler, wird die Exception an den Kernel weitergereicht, der darauf den entsprechenden Prozess terminiert.

5.) Softirqs, Tasklets, Bottom Halves

Softirqs, Tasklets und Bottom Halves (allgemein auch als verzögerbare Funktionen bezeichnet) erlauben es, aufwendige und nichtzeitkritische Funktionen, z. B. Kopieren eines Pufferinhalts, aus dem Interrupt-Handler herauszunehmen und zu einem späteren Zeitpunkt auszuführen, damit die unterbrochene Befehlssequenz nicht zu lange durch den Interrupt-Handler blockiert bleibt. Dennoch werden die verzögerbaren Funktionen nicht in einem eigenen Prozess ausgeführt, sondern, wie alle anderen Interrupt-Handler, im so genannten *Kernel-Kontroll-Pfad*. Damit bezeichnet man die Befehlssequenz, die zur Interrupt-Behandlung ausgeführt wird; d. h. alle Befehlssequenzen (Interrupt-Handler, ISR, Exception-Handler, etc.), die im Kernel Mode und nicht in einem Prozess ablaufen.
Softirqs, Tasklets und Bottom Halves sind sehr eng miteinander verwandt. Die Softirqs bilden die Basis, auf der die Tasklets aufbauen. Tasklets sind also nichts anderes als Softirqs mit zusätzlichen bzw. veränderten Eigenschaften. Bottom Halves sind wiederum nichts anderes als Tasklets mit anderen Eigenschaften.
Im Folgenden sollen Unterschiede und Gemeinsamkeiten näher betrachtet werden.

5.1) Softirqs:

Insgesamt können 32 Softirqs über eine statische Tabelle verwaltet werden. Jeder Softirq zeigt auf eine Funktion, mit der er verknüpft ist und die den eigentlichen Funktionsumfang zur Verfügung stellt. Das Herstellen dieser Verknüpfung wird Initialisierung genannt und geschieht während der Kernelinitialisierung. Eine nachträgliche Initialisierung zur Laufzeit ist nicht mehr möglich, da nach der Initialisierung keine Veränderungen mehr an der Tabelle vorgenommen werden kann. Damit ein Softirq ausgeführt werden kann, muss er zunächst als „wartend“ markiert werden (Aktivierung). Die Aktivierung geschieht in der Regel durch ISRs, kann aber auch von Softirqs selbst (während der Ausführung) geschehen. Die Behandlung der Softirqs (Ausführung) wird durch bestimmte Ereignisse, z. B. Beenden eines I/O Interrupt-Handlers, gestartet. Die Ausführung darf und kann nicht durch einen anderen Softirq-Handler auf derselben CPU unterbrochen werden. Während der Ausführung auftretende Softirqs werden von der noch laufenden Routine erfasst und bearbeitet. Die Ausführungsreihenfolge ist dabei von den Prioritäten abhängig. Die Priorität ist mit dem Index identisch, der die Softirqs identifiziert. Der Softirq Typ mit der Indexzahl 0 hat die höchste Priorität. Softirqs dürfen in einem Multiprozessorsystem grundsätzlich parallel behandelt werden, auch wenn zwei oder mehrere parallel laufende Softirqs vom selben Typ

sind. Das macht sie zum einen sehr flexibel, zum anderen muss der Programmierer darauf achten, dass parallel laufende Funktionen vom selben Softirq-Typ ihre gemeinsamen Datenstrukturen nicht willkürlich verändern. Dennoch werden Softirqs im Vergleich zu Tasklets und Bottom Halves, wegen des geringeren Verwaltungsaufwand für den Kernel, schneller ausgeführt. Teile des Netzwerkverkehrs sind deshalb mit Hilfe von Softirqs realisiert.

5.2) Tasklets:

Insgesamt existieren zwei Tasklet-Typen, die sich in ihrer Priorität unterscheiden. Das *high-priority Tasklet* ist ein Softirq mit der höchstmöglichen Prioritätsstufe 0, der zweite Tasklet-Typ ein Softirq der Prioritätsstufe 3. Für Tasklets besteht keine Beschränkung in ihrer möglichen Anzahl, da diese mit Hilfe einer dynamisch wachsenden Struktur verwaltet werden, so dass sie auch zur Laufzeit initialisiert werden können, z. B. durch ladbare Kernelmodule. Zwei oder mehrere Tasklets vom selben Typ dürfen nicht parallel ausgeführt werden. Wegen dieser Einschränkung muss sich der Programmierer nicht um Datenstrukturen kümmern, die von Tasklets desselben Typs benutzt werden.
Allerdings bringt die Einschränkung der Nebenläufigkeit zusätzlichen Verwaltungsaufwand für den Kernel mit sich. Zum einem entsteht dieser durch mehrere parallel laufende Tasklets desselben Typs, zum anderen muss vor der Aktivierung überprüft werden, ob bereits ein Tasklet desselben Typs auf der CPU aktiviert ist, auf der es aktiviert werden soll. In diesem Fall ist eine zweite Aktivierung unzulässig. Vor der Ausführung eines Tasklets muss geprüft werden, ob im gesamten System bereits ein Tasklet desselben Typs bearbeitet wird. Gegebenenfalls muss das Tasklet zu einem späteren Zeitpunkt ausgeführt werden.
Wie im Falle der Softirqs geschieht die Aktivierung in der Regel durch Interrupt Service Routinen, die damit ihre gerätespezifische Tasklet-Funktion aktivieren. Die Ausführung wird ebenso ausgelöst wie im Falle der Softirqs. Die Ausführungsstrategie ist mit der der Softirqs identisch, d. h. während der Ausführung anfallende Tasklets werden von der aktuell laufend Routine erfasst. Da high-priority Tasklets Softirqs mit höchstmöglicher Prioritätsstufe sind, werden diese immer vor allen anderen Softirqs ausgeführt.
Wegen der dynamischen Initialisierbarkeit und dem geringen Verwaltungsaufwand für den Programmierer, kommen Tasklets bevorzugt in Gerätetreibern zum Einsatz.

5.3) Bottom Halves:

Bottom Halves sind high-priority Tasklets. Verwaltet werden sie, ebenso wie Softirqs, mit Hilfe einer statischen Tabelle mit 32 möglichen Einträgen. Daher kann die Initialisierung nur während der Kernelinitialisierung stattfinden und nicht zur Laufzeit. Ihre Nebenläufigkeit ist noch weiter eingeschränkt als die der Tasklets. Im gesamten System darf zum selben Zeitpunkt nur ein Bottom Half ausgeführt werden. In Multiprozessorsystem führt diese Einschränkung im Vergleich zu Tasklets oder Softirqs zu einer geringeren Gesamt-ausführungsgeschwindigkeit. Da Bottom Halves high-priority Tasklets sind, werden diese, ohne viel Aufwand seitens des Programmierers immer eine der ersten Funktionen sein, die von dem Softirq-Handler ausgeführt werden. Die Ausführungsstrategie ist identisch mit der der Softirqs. Vor der Ausführung muss allerdings zuerst überprüft werden, ob bereits ein Bottom Half im gesamten System bearbeitet wird. Die Aktivierung entspricht der der Tasklets.
Neben der Möglichkeit eine einzelne Funktion mit einem Bottom Half zu verknüpfen, ist es auch erlaubt, eine *Task Queue* (Gruppe von Funktionen) zu registrieren. Eine solche Gruppierung von Funktionen kann einerseits aus Interrupt spezifischen Funktionen bestehen, andererseits auch aus Kernel-Funktionen, die nichts mit dem Interrupt-Handling zu tun haben.
Neben sehr wenigen Gerätetreibern, in denen Bottom Halves zum Einsatz kommen, werden sie ebenfalls vom Kernel selbst verwendet. Zum Beispiel führt der Kernel über ein Timer Bottom Half, das bei jedem Timer Interrupt aktiviert wird, eine Task Queue aus.

Literaturverzeichnis:

Bovet D.P.; Ceasti M.: Understanding Linux Kernel, Second Edition. – O'Reilly & Associates 2002

Silberschatz A.; Gagne G.; Bear Galvin P.: Operating System Concepts, Sixth Edition. – John Wiley & Sons 2002

Tanenbaum A.: Modern Operating Systems, Second Edition. – Prentice Hall 2001

Rohde J.: Assembler GE-PACKT, 1. Auflage. – MITP-Verlag 2001

Kay M.: Hardwarenahe Datenstrukturen. – http://www.matthiaskay.de

Stamm T.: Register. – http://n.ethz.ch/student/stammt/index.html