

# 11 Kernel-Aktivitäten und Zeitfluss

In Kapitel 10 („Systemaufrufe“) wurde gezeigt, wie die Ausführungszeit des Systems in zwei große, unterschiedliche Teile gespalten werden kann: den Kern- und den Benutzermodus. In diesem Kapitel werden wir die verschiedenen Aktivitätsformen des Kerns untersuchen und sehen, dass eine wesentlich feinere Unterteilung notwendig ist, um der Situation gerecht zu werden.

Systemaufrufe sind nicht die einzige Möglichkeit, um zwischen Benutzer- und Systemmodus wechseln zu können: Wie aus den vorhergehenden Kapiteln bekannt ist, verwenden alle von Linux unterstützten Plattformen das Konzept der Interrupts, um periodische Unterbrechungen für die verschiedensten Zwecke zu realisieren. Dabei müssen zwei Typen unterschieden werden:

- *Interrupts* werden automatisch vom System und den angeschlossenen Zubehörgeräten generiert. Sie dienen zum einen der effizienteren Implementierbarkeit von Gerätetreibern, werden zum anderen aber auch vom Prozessor selbst benötigt, um auf Ausnahmesituationen oder Fehler hinzuweisen, die der Interaktion mit dem Kernelcode bedürfen.
- *SoftIRQs* werden verwendet, um zeitverzögerte Vorgänge im Kernel selbst effektiv zu implementieren.

Im Gegensatz zu anderen Teilen des Kerns enthält der Code zur Behandlung der Interrupt- und Systemaufruf-spezifischen Abschnitte sehr starke Verflechtungen zwischen Assembler- und C-Code, da teilweise subtile Probleme gelöst werden müssen, die sich in C alleine nicht oder nicht vernünftig beherrschen lassen. Dies ist kein Linux-spezifisches Problem: Die meisten Betriebssystementwickler versuchen, unabhängig von ihrem individuellen Ansatz, die Low-level-Behandlung der entsprechenden Punkte so tief wie möglich in den Quellen des Kerns zu verstecken und für den restlichen Code unsichtbar werden zu lassen, was aufgrund der technischen Gegebenheiten nicht in allen Fällen möglich ist.

Auch wenn wir bestrebt sein werden, die folgende Diskussion möglichst losgelöst von einem spezifischen Prozessortyp zu führen, ist es dennoch nicht immer möglich, ganz auf der Architektur-neutralen Seite zu bleiben. Wo immer es sich nicht vermeiden lässt, werden wir uns vor allem an der IA32-Architektur orientieren. Auf Abweichungen bei anderen Prozessortypen gehen wir ein, sofern die Differenzen nicht allzu groß sind.

Häufig benötigt der Kern Mechanismen, die es ihm ermöglichen, Arbeiten auf einen bestimmten Zeitpunkt in der Zukunft zu verschieben oder in einer Warteschlange zu speichern, die nach und nach abgearbeitet wird, wenn Zeit ist. Anwendungen für Mechanismen dieser Art sind in den vorhergehenden Kapiteln bereits einige male aufgetaucht, in diesem Abschnitt werden wir genauer auf ihre Implementierung eingehen.

## 11.1 Interrupts

Die einzige Gemeinsamkeit bei der Implementierung von Interrupts auf den diversen Plattformen, die vom Linux-Kern unterstützt werden, ist die Tatsache, dass es sie gibt – damit hören die Übereinstimmungen aber auch schon auf. Auch die Nomenklatur unterscheidet sich meist

deutlich voneinander. Wir wollen daher zuerst die gebräuchlichen Typen von Systemunterbrechungen vorstellen, um eine definierte Ausgangsbasis zu besitzen, bevor wir genauer auf ihre Funktionsweise und die damit verbundenen Möglichkeiten und Probleme eingehen.

### 11.1.1 Interrupt-Typen

Man kann die in einem System auftretenden Interrupt-Typen generell in zwei verschiedene Kategorien einteilen:

- *Synchrone Interrupts, Ausnahmen* oder *Exceptions* werden von der CPU selbst ausgelöst und sind für das Programm bestimmt, das gerade ausgeführt wird. Exceptions können aus verschiedenen Gründen ausgelöst werden: zum einen kann es sich um einen Programmfehler handeln, der zur Laufzeit aufgetreten ist (klassisches Beispiel: Division durch Null), zum anderen kann – wie der Name schon sagt – eine Ausnahmesituation eingetreten sein, die der Prozessor nicht ohne „externe“ Hilfe erledigen kann.

Im ersten Fall muss der Kernel die Applikation über das Auftreten des Ausnahme informieren, wozu beispielsweise der in Kapitel 4 („Interprozesskommunikation und Locking“) vorgestellte Signalmechanismus verwendet werden kann: Die Applikation erhält dadurch die Chance, den Fehler zu korrigieren, eine entsprechende Fehlermeldung auszugeben oder sich einfach zu beenden.

Eine allgemeine Ausnahmesituation wurde nicht direkt vom Prozess verschuldet, muss aber mit Hilfe des Kerns repariert werden. Ein mögliches Beispiel hierfür ist ein Page Fault, der immer dann auftritt, wenn ein Prozess versucht, auf eine Seite des virtuellen Adressraums zuzugreifen, die nicht im RAM-Speicher enthalten ist. Wie in Kapitel 3 („Speicherverwaltung“) besprochen, muss der Kern in diesem Fall im Zusammenspiel mit der CPU dafür sorgen, dass die gewünschten Daten in den RAM-Speicher geholt werden; der Prozess kann danach an derselben Stelle weiterlaufen, an der die Ausnahme aufgetreten ist. Er hat nicht bemerkt, dass ein Seitenfehler aufgetreten ist, da der Kernel die Situation automatisch bereinigt hat.

- *Asynchrone Interrupts* entsprechen dem von Zubehörgeräten bekannten klassischen Interrupt-Typ: Sie treten zu nicht vorhersehbaren Zeitpunkten auf. Im Gegensatz zu synchronen Interrupts sind asynchrone Interrupts nicht an einen bestimmten Prozess gebunden; sie treten unabhängig davon auf, mit welchen Dingen das System gerade beschäftigt ist.<sup>1</sup>

Netzwerkadapter melden das Eintreffen neuer Pakete, indem der ihnen zugewiesene Interrupt ausgelöst wird. Da die Daten an einem zufälligen Moment ins System gelangen, ist die Wahrscheinlichkeit sehr hoch, dass gerade irgendein Prozess ausgeführt wird, der nichts mit den Daten zu tun hat. Um diesen Prozess nicht zu benachteiligen, muss der Kern dafür sorgen, dass der Interrupt so schnell wie möglich abgearbeitet wird und die Daten irgendwo „zwischenlagert“ werden, damit die Rechenzeit wieder an den Prozess zurückgegeben werden kann. Aus diesem Grund benötigt der Kern Mechanismen zum Aufschieben von Arbeit, die wir ebenfalls in diesem Kapitel besprechen werden.

Welche Gemeinsamkeiten bestehen zwischen beiden Interrupt-Typen? Die CPU wird in jedem Fall veranlasst, vom Benutzer- in den Kernelmodus zu wechseln, sofern sie sich nicht ohnehin im Kernelmodus befindet. Dort wird eine spezielle Routine ausgeführt, die als *interrupt service*

<sup>1</sup> Da Interrupts auch gesperrt werden können, wie wir gleich sehen werden, ist diese Aussage nicht völlig korrekt, da das System zumindest darauf einwirken kann, wann Unterbrechungen *nicht* auftreten.

*routine* (abgekürzt *ISR*) oder *interrupt handler* bezeichnet wird und dazu dient, mit einer Ausnahmebedingung oder einer wie auch immer veränderten Situation fertig zu werden – schließlich dient eine Unterbrechung genau dazu, den Kernel auf eine Veränderung aufmerksam zu machen, die seine Aufmerksamkeit beansprucht!

Die bloße Unterscheidung zwischen synchronen und asynchronen Interrupts beschreibt deren Eigenschaften noch nicht vollständig, da ein weiterer Punkt zu beachten ist: Viele Interrupts können abgeschaltet werden, während dies bei einigen wenigen nicht möglich ist. Zu letzterer Kategorie zählen beispielsweise Interrupts, die von Hardwarefehlern oder anderen stark systemkritischen Ereignissen ausgelöst werden.

Der Kern versucht zwar, das Abschalten von Interrupts wann immer möglich zu vermeiden, da dies aus offensichtlichen Gründen nicht gut für die Systemleistung ist. Dennoch gibt es Momente, in denen Interrupts abgeschaltet sein *müssen*, wenn sich der Kern nicht selbst in Bedrängnis bringen will: Wie wir bei der genaueren Betrachtung von Interrupt-Handlern sehen werden, können sich größere Probleme im Kernel ergeben, wenn *während* der Behandlung eines Interrupts bestimmte andere Interrupts auftreten. Wenn der Kern mitten in der Abarbeitung von ohnehin kritischem Code gestört wird, kann es zu den in Kapitel 4 („Interprozesskommunikation und Locking“) besprochenen Synchronisationsproblemen kommen, was im schlechtesten Fall zu einem Deadlock im Kern führt, der das ganze System unbrauchbar macht.

Gönnt sich der Kern zu viel Zeit beim Abarbeiten einer ISR mit abgeschalteten Interrupts, kann (und wird) es passieren, dass Interrupts verloren gehen, die für den korrekten Betrieb des Systems aber notwendig gewesen wären. Der Kernel löst dieses Problem, indem er die Möglichkeit anbietet, Interrupt-Handler in zwei Teile zu spalten: Eine Performance-kritische obere Hälfte (*top half*), die mit abgeschalteten Interrupts ausgeführt wird, und eine weniger wichtige untere Hälfte (*bottom half*), die zu einem späteren Zeitpunkt zur Durchführung aller weniger wichtigen Aktionen verwendet wird. In früheren Kernelversionen gab es einen gleichnamigen Mechanismus zum Aufschieben von Arbeit auf einen späteren Zeitpunkt, der aber durch effizientere Mechanismen ersetzt wurde, die wir weiter unten vorstellen.

Jeder Interrupt verfügt über eine Kennzahl, über die er sich identifizieren lässt: Weist man einer Netzwerkkarte die Interrupt-Kennzahl  $n$ , dem SCSI-Controller  $m \neq n$  zu, kann der Kernel zwischen beiden Geräten differenzieren und die passende ISR aufrufen, die eine gerätespezifische Aktion erledigt. Dasselbe Prinzip gilt natürlich auch für Exceptions, wo unterschiedliche Kennzahlen unterschiedliche Ausnahmen bedeuten. Leider ist die Situation durch spezielle (meist historisch bedingte) Design-„Features“, bei denen sich die IA-32-Architektur besonders hervortut, nicht immer so einfach, wie eben geschildert wurde: Da nur sehr wenige Kennzahlen für Hardware-Interrupts zur Verfügung gestellt werden, müssen Interrupts unter mehreren Geräten geteilt werden. Auf IA-32-Prozessoren beträgt die maximale Anzahl in den meisten Fällen 15, was nicht gerade üppig ist – vor allem wenn man die Tatsache bedenkt, dass einige Interrupts bereits mit einer festen Aufgabe für die Standardkomponenten des Systems (Tastatur, Timer etc.) betraut sind, die die Auswahl für andere Zubehörgeräte noch weiter einschränken.

Das Teilen von Interrupts wird als *Interrupt sharing* bezeichnet.<sup>2</sup> Um diese Technik einsetzen zu können, ist allerdings nicht nur Unterstützung von Seiten der Hardware, sondern auch vom Kernel selbst notwendig, um identifizieren zu können, von welchem Gerät eine Unterbrechung stammt, worauf wir noch genauer eingehen werden.

---

<sup>2</sup> Bussysteme, die ein durchdachtes *Gesamtdesign* besitzen, benötigen diese Möglichkeit natürlich nicht: Sie können so viele Interrupts für Hardwaregeräte bereitstellen, dass gar keine Notwendigkeit zu teilen besteht.

### 11.1.2 Hardware-IRQs

Der Begriff „Interrupt“ wurde bisher recht sorglos verwendet, um sowohl von der CPU wie auch von externer Hardware ausgehende Unterbrechungen zu beschreiben. Hardware-kundige Leser werden sicherlich bemerkt haben, dass dies nicht ganz korrekt ist: Interrupts können von prozessorfremden Zubehörgeräten eigentlich nicht direkt ausgelöst werden, sondern müssen mit Hilfe eines in jedem System befindlichen Standardbausteins angefordert werden, der als *Interrupt controller* bezeichnet wird.

Von den Zubehörgeräten (bzw. ihren Einsteckslots) führen elektronische Leitungen zu dem Baustein, die verwendet werden, um eine Interrupt-Anforderung an den Interrupt-Controller zu schicken, der diese – nach etwaigen elektrotechnischen Arbeiten, die uns hier nicht weiter interessieren – an die Interrupt-Eingänge der CPU weiterleitet. Da Zusatzgeräte Interrupts nicht direkt erzwingen können, sondern diese erst über den genannten Baustein anfordern müssen, bezeichnet man sie korrekterweise als IRQ – *Interrupt request* bzw. Interruptanforderung.

Da der Unterschied zwischen IRQs und Interrupts softwareseitig gesehen nicht allzu groß ist, werden die beiden Begriffe in der Praxis meist etwas schwammig und austauschbar verwendet. Dies ist kein Problem, solange offensichtlich ist, was gerade gemeint ist.

Dennoch gilt es, einen wichtigen Punkt beachten, der die Nummerierung von IRQs und Interrupts betrifft und auch softwareseitige Auswirkungen hat: Die meisten CPUs stellen aus dem gesamten Spektrum verfügbarer Interrupt-Nummern nur einen Ausschnitt zur Verfügung, der für Hardware-Interrupts verarbeitet werden kann. Dieser Bereich befindet sich üblicherweise mitten in der laufenden Nummerierung, bei IA-32-CPU's sind dafür beispielsweise die Kennzahlen 32 bis 47 – insgesamt 16 Stück – vorgesehen.

Wie jeder Leser weiß, der schon einmal Zubehörkarten in einem IA32-System konfiguriert oder den Inhalt von `/proc/interrupts` betrachtet hat, beginnt die Nummerierung der IRQs von Erweiterungskarten aber bei 0 und endet bei 15! Dies sind ebenfalls 16 verschiedene Möglichkeiten, aber andere Zahlenwerte. Neben dem elektrischen Handling der IRQ-Signale führt der Interruptcontroller zusätzlich eine „Umschreibung“ zwischen IRQ- und Interruptkennzahl durch, wobei dies im Fall von IA-32 einer einfachen Addition mit 32 entspricht: Die Auslösung von IRQ 9 durch ein Gerät resultiert in der Auslösung von Interrupt 41 der CPU, was bei der Installation von Interrupthandlern beachtet werden muss. Auf anderen Architekturen werden andere Abbildungen zwischen Interrupts und IRQs verwendet, die wir nicht im Detail behandeln.

### 11.1.3 Bearbeiten von Interrupts

Nachdem die CPU über eine Unterbrechung informiert wurde, delegiert sie die weitere Behandlung der Situation an eine Software-Routine, in der ein Fehler behoben, eine Sondersituation behandelt oder ein Benutzerprozess über ein externes Ereignis informiert wird. Da jeder Interrupts und jede Exception mit einer eindeutigen Kennzahl versehen ist, verwendet der Kern ein Array, in dem sich Zeiger auf Handlerfunktionen befinden. Die zugehörige Interruptkennzahl kann anhand der Arrayposition festgestellt werden, wie Abbildung 11.1 auf der gegenüberliegenden Seite zeigt.

#### Vor- und Nacharbeiten

Wie Abbildung 11.2 auf der gegenüberliegenden Seite zeigt, ist die Behandlung eines Interrupts in drei Teile aufgespalten: Zuerst muss eine passende Umgebung eingerichtet werden, in der die Handlerfunktion ablaufen kann; danach wird der Handler selbst aufgerufen, und anschließend wird der Zustand des Systems so restauriert, dass er (aus Sicht des laufenden Programms) genau

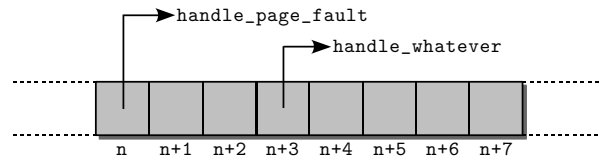


Abbildung 11.1: Verwaltung von Interrupt-Handlern

dem Zustand vor dem Auftreten des Interrupts entspricht. Die Abschnitte vor und nach Aufruf des Interrupt-Handlers werden als *entry* und *exit path*, also Ein- und Ausgangspfad bezeichnet.

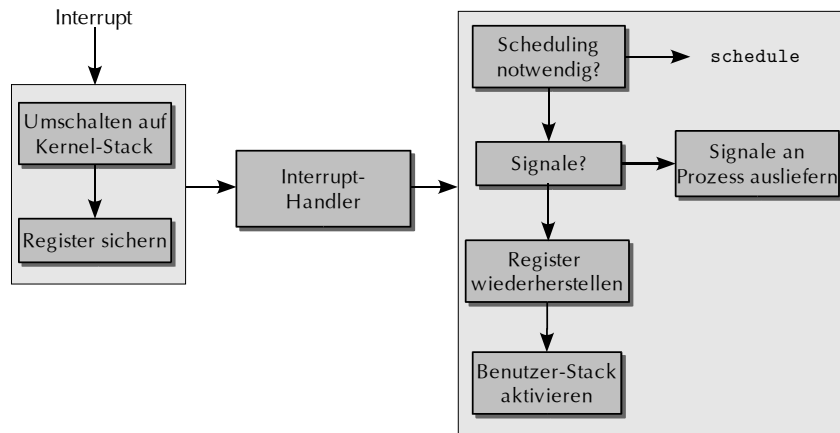


Abbildung 11.2: Abarbeitung eines Interrupts

Die Vor- und Nacharbeiten sind auch dafür verantwortlich, dass der Prozessor vom Benutzer in den Kernmodus wechselt: Eine zentrale Aufgabe des Eingangspfades ist der Wechsel vom Benutzer- auf den Kernelstack. Dies ist aber nicht genug: Da auch der Kernel bei der Ausführung seines Codes auf die Ressourcen der CPU zurückgreift, muss der Eingangspfad den aktuellen Registerstatus der Benutzerapplikation sichern, um diesen nach Beendigung der Interrupt-Aktivitäten wiederherstellen zu können. Dies ist der gleiche Mechanismus, der auch bei Kontextwechseln beim Scheduling verwendet wird. Beim Eintritt in den Kernmodus wird nicht der komplette Registersatz gespeichert, sondern nur ein kleiner Teil davon: der Kern verwendet nicht alle Register, die zur Verfügung stehen; da im Kernelcode beispielsweise keine Gleitkomma-Operationen verwendet werden (es wird nur mit Ganzzahl-Arithmetik gerechnet), brauchen die Floating-Point-Register nicht gesichert zu werden:<sup>3</sup> Ihr Wert ändert sich bei der Ausführung von Kernelcode nicht. Um die Unterschiede zwischen den verschiedenen CPUs berücksichtigen zu können, wird die plattformabhängige Datenstruktur `pt_regs` definiert, die alle im Kernmodus modifizierten Register aufzählt (Abschnitt 11.1.5 geht genauer darauf ein). In Assembler codierte Low-level-Routinen übernehmen das Ausfüllen der Struktur.

<sup>3</sup> Manche Architekturen (wie beispielsweise IA-64) weichen von dieser Regel ab, indem sie einige wenige Register aus dem Gleitkommasatz verwenden und diese bei jedem Eintritt in den Kernmodus mitsichern. Die große Masse der Gleitkomma-Register bleibt allerdings dennoch vom Kern „verschont“, und es werden auch keine expliziten Gleitkommaoperationen verwendet.

Im Ausgangspfad prüft der Kern, ob

- Der Scheduler einen neuen Prozess auswählen soll, durch den der alte ersetzt wird.
- Signale vorhanden sind, die an den Prozess ausgeliefert werden müssen.

Erst wenn diese beiden Fragen erledigt sind, kann sich der Kernel seinen Standardaufgaben widmen, die bei der Rückkehr aus einem Interrupt in jedem Fall durchgeführt werden müssen: Wiederherstellen des Registersatzes, Umschalten auf den User-Stack und Wechsel in einen für Benutzerapplikationen angepassten Prozessormodus bzw. Wechsel in einen anderen Protection Ring.<sup>4</sup>

Da ein Zusammenspiel zwischen C- und Assemblercode erforderlich ist, muss besondere Sorgfalt darauf verwendet werden, den Austausch von Daten zwischen Assembler- und C-Ebene korrekt zu gestalten. Der entsprechende Code findet sich in `arch/arch/kernel/entry.S` und macht detailliert Gebrauch von speziellen Eigenschaften der einzelnen Prozessoren. Der Inhalt der Datei wird daher so selten wie möglich – und dann auch nur mit höchster Sorgfalt – modifiziert.

Achtung: Die Arbeit im Ein- und Ausgangspfad eines Interrupts wird zusätzlich durch die Tatsache erschwert, dass sich der Prozessor bei Eintreffen eines Interrupts nicht nur im Benutzer-, sondern auch im Kernmodus befinden kann. Dies erfordert einige weitere technische Änderungen, die aus Gründen der Übersichtlichkeit nicht in der Abbildung berücksichtigt wurden (im Wesentlichen entfällt das Umschalten zwischen Kernel- und Userstack und die Tests, ob der Scheduler aufgerufen oder Signale ausgeliefert werden sollen).

Der Begriff „Interrupt-Handler“ wird zweideutig verwendet: Zum einen bezeichnet man damit den Aufruf einer ISR durch die CPU, bei der Entry/Exit-Pfad und die ISR zusammengefasst werden. Korrekter wäre es natürlich, nur die Routine anzusprechen, die *zwischen* Entry- und Exit-Path ausgeführt wird und in C implementiert ist.

### Interrupt-Handler

Interrupt-Handler werden vor allem durch die Tatsache gestört, dass während ihrer Ausführung weitere Interrupts auftreten können. Dies kann zwar durch Abschalten der Interrupts während der Abarbeitung eines Handlers vermieden werden, was aber andere Probleme wie das Versäumen wichtiger Interrupts mit sich bringt. Die „Maskierung“ (wie man die selektive Abschaltung eines oder mehrerer Interrupts auch bezeichnet) kann daher nur über kurze Zeiträume angewandt werden.

ISRs müssen demnach zwei Forderungen erfüllen:

- Die Implementierung (vor allem für die Abschnitte, in denen andere Interrupts deaktiviert sind) muss aus so wenig Code wie möglich bestehen, um zügig abgearbeitet werden zu können.
- Handler-Routinen von Interrupts, die während der Abarbeitung anderer ISRs aufgerufen werden können, dürfen sich gegenseitig nicht beeinflussen.

Während der zweite Punkt durch programmiertechnisches Können und cleveres Design der ISR erfüllt werden kann, ist die Zufriedenstellung der ersten Forderung etwas schwieriger: Je nach

<sup>4</sup> Manche Prozessoren führen diesen Wechsel automatisch ohne explizite Anforderung durch den Kern aus.

Interrupt gibt es ein festes Programm, das abgearbeitet werden *muss*, um die Minimalanforderungen zur Klärung der Situation zu erfüllen. Der Code kann daher nicht beliebig klein gemacht werden.

Wie entgeht der Kernel diesem Dilemma? Nicht jeder Teil eines ISR ist gleich wichtig; generell kann jede Handlerroutine in drei Teile aufgespalten werden, die sich in ihrer Brisanz unterscheiden:

- *Kritische* Abschnitte müssen unmittelbar nach Auslösung des Interrupts ausgeführt werden, da ansonsten die Stabilität des Systems bzw. der korrekte Ablauf der weiteren Arbeit des Computers nicht aufrechterhalten werden kann. Andere Interrupts müssen bei der Abarbeitung eines solchen Teils ausgeschaltet sein.
- *Nichtkritische* Teile sollten ebenfalls schnellstmöglich erledigt werden, können aber mit eingeschalteten Interrupts erledigt werden (und dürfen damit von anderen Systemereignissen unterbrochen werden).
- *Verschiebbare* Abschnitte sind nicht besonders wichtig und müssen nicht im Interrupthandler selbst implementiert werden: Der Kernel kann sie „auf die lange Bank“ schieben und dann ausführen, wenn er dafür Zeit findet, in der er nichts Besseres zu tun hat.

Um verschiebbare Abschnitte von ISRs zu einem späteren Zeitpunkt auszuführen, stellt der Kern beispielsweise *Tasklets* zur Verfügung, die in Abschnitt `act_tasklets` genauer besprochen werden.

#### 11.1.4 Initialisierung und Reservierung von IRQs

Die technische Seite der Interrupt-Implementierung präsentiert sich mit zwei Gesichtern: Assemblercode, der höchst Prozessor-spezifisch ist und zur Verarbeitung der Low-level-Details eingesetzt wird, die auf der jeweiligen Plattform relevant sind, und eine abstrahierte Schnittstelle, die von Gerätetreibern und anderem Kernelcode benötigt wird, um Handler für IRQs zu installieren und zu verwalten. Wir wollen uns hier vor allem auf den zweiten Punkt konzentrieren: Die zahllosen Detailerläuterungen, die zur Beschreibung der Funktion des Assemblerteils notwendig sind, sind besser in Büchern über Prozessorarchitektur oder den jeweiligen Architekturhandbüchern aufgehoben.

Achtung: Sparc und Sparc64 gehen bei der Verwaltung und Behandlung von Interrupts ihre eigenen Wege, die sich deutlich von den Standardmethoden und vor allem Standarddatenstrukturen unterscheiden, die bei den anderen Plattformen verwendet werden. Natürlich wirkt sich dies nicht auf die Schnittstelle aus, die der Kernel zur Kommunikation mit Gerätetreibern bezüglich Interruptverwaltung und -handling verwendet: Würde sich diese unterscheiden, müssten schließlich alle Gerätetreiber für Sparc neu geschrieben oder zumindest modifiziert werden, was dem Gedanken der größtmöglichen Plattformunabhängigkeit eindeutig zuwiderläuft.

Um auf den IRQ eines Zubehörgerätes reagieren zu können, muss der Kernel eine Funktion für jeden möglichen IRQ bereithalten, die sich dynamisch registrieren und auch wieder entfernen lässt: Da auch Module für Geräte geschrieben werden können, die über Interrupts mit dem restlichen System interagieren, reicht eine statische Organisation der Tabelle nicht aus.

Die zentrale Stelle, an der Informationen über IRQs verwaltet werden, ist ein globales Array, das einen Eintrag für jede IRQ-Kennzahl besitzt. Da Arrayposition und Interruptnummer identisch sind, ist es leicht, den zu einem bestimmten IRQ gehörenden Eintrag zu finden: IRQ 0 ist an

Stelle 0, IRQ 15 an Position 15 u.s.w. – auf welchen Interrupt des Prozessors die IRQs letztendlich abgebildet werden, ist an dieser Stelle nicht von Belang.

Das Array ist wie folgt definiert:

```
<irq.h> extern irq_desc_t irq_desc [NR_IRQS];
```

Obwohl ein architekturunabhängiger Datentyp für die einzelnen Einträge verwendet wird, ist die Anzahl der maximal möglichen IRQs durch eine plattformabhängige Konstante gegeben: NR\_IRQS. Sie wird in der Prozessor-spezifischen Headerdatei `include/asm-arch/irq.h` definiert; ihr Wert schwankt nicht nur stark zwischen den verschiedenen Prozessoren, sondern ändert sich auch innerhalb von Familien je nach verwendetem Hilfschip, der die CPU bei der Verwaltung von IRQs unterstützt. Alpha-Rechner unterstützen von 32 auf den „kleineren“ Systemen bis zur sagenhaften Anzahl von 2048 auf Wildfire-Boards, IA-64-Computer sind stets auf 256 festgelegt, IA-32 bietet zusammen mit dem klassischen Controller 8256A nur Support für magere 16 IRQs, eine Erhöhung dieser Anzahl auf 224 ist durch die Verwendung der IO-APIC-Erweiterung möglich, die sich in allen Multiprozessorsystemen findet, aber auch auf UP-Maschinen einsetzbar ist.<sup>5</sup>

Interessanter als die maximale Anzahl von IRQs ist der Datentyp, der für die Arrayeinträge verwendet wird (im Gegensatz zum einfachen Beispiel weiter oben handelt es sich dabei nicht nur um einen simplen Zeiger auf eine Funktion):

```
<irq.h> typedef struct {
    unsigned int status;           /* IRQ status */
    hw_irq_controller *handler;
    struct irqaction *action;     /* IRQ action list */
    unsigned int depth;          /* nested irq disables */
} ____cacheline_aligned irq_desc_t;
```

Jeder IRQ wird aus Sicht des High-level-Codes im Kernel vollständig durch diese Struktur beschrieben; alle hardware-spezifischen Besonderheiten werden von diesem Interface verborgen.

`depth` erfüllt zwei Aufgaben: Zum einen kann es verwendet werden, um festzustellen, ob eine IRQ-Linie aktiviert oder deaktiviert ist: Ein positiver Wert steht für den letztgenannten Fall, während eine 0 bei einer aktivierten Linie verwendet wird. Warum werden positive Werte für deaktivierte IRQs verwendet? Dies ermöglicht dem Kern nicht nur, zwischen aktivierten und deaktivierten Linien zu unterscheiden, sondern erlaubt auch eine mehrfache Abschaltung ein- und desselben Interrupts: Jedes Mal, wenn Code aus dem restlichen Teil des Kerns einen Interrupt abschaltet, wird der Zähler um 1 erhöht, jede erneute Aktivierung führt zu einer entsprechenden Dekrementierung. Erst wenn `depth` auf 0 zurückgefallen ist, darf der IRQ auch hardwaremäßig wieder freigeschaltet werden. Diese Vorgehensweise ermöglicht es, die verschachtelte Deaktivierung von Interrupts korrekt zu behandeln.

Ein IRQ kann seinen Zustand nicht nur bei der Installation eines Handlers, sondern auch im laufenden Betrieb ändern: `status` dient dazu, den gerade aktuellen Zustand festzuhalten. In `<irq.h>` werden diverse Konstanten definiert, die den aktuellen Zustand einer IRQ-Linie beschreiben. Jede Konstante steht für ein gesetztes Bit in einer Bitkette, weshalb auch mehrere Werte gleichzeitig gesetzt werden können, sofern sie sich nicht gegenseitig widersprechen:

- `IRQ_DISABLED` wird bei von einem Gerätetreiber abgeschaltetem IRQ verwendet, was den Kernel anweist, den Handler nicht mehr zu betreten.

<sup>5</sup> Allerdings gibt es nur sehr wenige Einzelprozessormaschinen, die dies auch tatsächlich umsetzen.

- Während der Abarbeitung eines IRQ-Handlers wird der Status auf `IRQ_INPROGRESS` gesetzt, was dem restlichen Kernelcode wie bei `IRQ_DISABLED` verbietet, den Handler auszuführen.
- `IRQ_PENDING` ist aktiv, wenn die CPU einen Interrupt bemerkt, den entsprechenden Handler aber noch nicht ausgeführt hat.
- `IRQ_PER_CPU` ist gesetzt, wenn ein IRQ nur auf einer einzigen CPU auftreten kann (dies macht auf SMP-Systemen einige Schutzmechanismen gegen konkurrierende Zugriffe überflüssig).
- `IRQ_LEVEL` wird auf Alpha und PPC verwendet, um Level- von Edge-getriggerten IRQs abzugrenzen.<sup>6</sup>
- `IRQ_REPLAY` bedeutet, dass der IRQ abgeschaltet wurde, während ein noch nicht abgearbeiteter Interrupt aus der Zeit vorher wartet.
- `IRQ_AUTODETECT` und `IRQ_WAITING` dienen der automatischen Erkennung und Konfiguration von IRQs, die bei manchen Bussen erforderlich ist.

Mit Hilfe des aktuellen Inhalts von `status` kann der Kernel leicht abfragen, in welchem Zustand sich ein spezieller IRQ gerade befindet, ohne auf die hardware-spezifischen Besonderheiten der darunter liegenden Implementierung zurückgreifen zu müssen. Natürlich bringt das alleinige Setzen der entsprechenden Flags nicht den gewünschten Effekt: das Abstellen eines Interrupts durch Setzen von `IRQ_DISABLED` ist nicht möglich: Auch die zugrunde liegende Hardware muss über den neuen Zustand informiert werden. Die Flags dürfen daher nur von controllerspezifischen Funktionen gesetzt werden, die zugleich die notwendigen Low-level-Einstellungen an der Hardware vornehmen, wozu in vielen Fällen Assemblercode oder das Schreiben von magischen Zahlen an magische Adressen mit Hilfe von `out`-Befehlen notwendig ist.

Die verbleibenden Elemente von `irq_desc_t` sind Zeiger auf Instanzen anderer Datenstrukturen, die zur plattformunabhängigen Verwaltung der Handlerfunktionen (`action`) und zur Handhabung der zugrunde liegenden IRQ-Controller-Hardware (`handler`) dienen.

### Abstraktion von IRQ-Controllern

`handler` ist eine Instanz des Datentyps `hw_irq_controller`, der die spezifischen Eigenschaften eines IRQ-Controllers für den architekturunabhängigen Teil des Kerns abstrahiert. Die darin enthaltenen Funktionen werden verwendet, um den Zustand eines IRQs zu ändern, weshalb sie auch das Setzen von `flag` übernehmen:

```

struct hw_interrupt_type {
    const char * typename;
    unsigned int (*startup)(unsigned int irq);
    void (*shutdown)(unsigned int irq);
    void (*enable)(unsigned int irq);
    void (*disable)(unsigned int irq);
    void (*ack)(unsigned int irq);
    void (*end)(unsigned int irq);
}

```

<irq.h>

<sup>6</sup> Edge-Triggerung bedeutet, dass die Hardware am Auftreten eines Potentialunterschieds in der Leitung erkennt, dass ein Interrupt aufgetreten ist. Bei Level-getriggerten Systemen wird ein Interrupt erkannt, wenn sich das Potential auf einem bestimmten Wert befindet, wobei nicht die Potentialänderung das ausschlaggebende Merkmal ist.

Level-Triggerung ist aus Sicht des Kerns komplizierter zu verwenden, da die Leitung nach jedem Interrupt explizit auf das Potential gesetzt werden muss, das für „kein Interrupt“ steht.

```

        void (*set_affinity)(unsigned int irq, unsigned long mask);
    };

    typedef struct hw_interrupt_type hw_irq_controller;

```

`typename` enthält eine kurze Zeichenkette, die den Hardwarecontroller identifiziert. Auf IA32-Rechnern sind die hierfür möglichen Werte „XT-PIC,“ und „IO-APIC-edge“, während die Werte auf anderen Systemen bunt gemischt sind, da viele verschiedene Controllertypen erhältlich und verbreitet sind.

Die Funktionszeiger haben folgende Bedeutung:

- `startup` verweist auf eine Funktion, die zur erstmaligen Initialisierung eines IRQs verwendet wird. In den meisten Fällen beschränkt sich die Initialisierung auf das Einschalten des IRQs, weshalb die `startup`-Funktion in vielen Fällen lediglich eine Weiterleitung zu `enable` enthält.
- `enable` aktiviert einen IRQ, d.h. sie führt den Wechsel vom aus- in den eingeschalteten Zustand durch, wozu üblicherweise hardware-spezifische Kennzahlen an ebenso hardware-spezifische Stellen im IO-Speicher bzw. den IO-Ports geschrieben werden müssen,
- `disable` ist das Gegenteil von `enable`: Die Funktion wird verwendet, um einen IRQ zu deaktivieren.
- `ack` ist eng mit der Hardware des Interrupt-Controllers verknüpft: Bei manchen Modellen muss das Eintreffen einer IRQ-Anforderung (und damit des entsprechenden Interrupts am Prozessor) explizit bestätigt werden, damit die darauf folgenden Anforderungen verarbeitet werden können. Wenn ein Chipsatz diese Anforderung nicht stellt, kann der Zeiger mit einer Dummy-Funktion oder einem Nullpointer belegt werden.
- `end` wird aufgerufen, wenn ein Interrupt in seiner eigenen Handlerfunktion deaktiviert wurde.
- In Mehrprozessorsystemen kann `set_affinity` verwendet werden, um die Affinität eines IRQ zu einer CPU zu steuern. Auf diese Weise ist es möglich, IRQs gezielt auf bestimmte CPUs zu verteilen (normalerweise werden IRQs auf SMP-Systemen gleichmäßig auf alle Prozessoren verteilt). Die Methode ergibt auf Einprozessorsystemen offensichtlich keinen Sinn, weshalb sie dort mit einem Nullzeiger belegt wird.

Der Typ des verwendeten Interrupt-Controllers kann (zusammen mit der Belegung aller IRQs des Systems) aus `/proc/interrupts` ausgelesen werden:

```

wolfgang@meitner> cat /proc/interrupts
          CPU0
0:   35633610      XT-PIC  timer
1:     124        XT-PIC  i8042
2:     0          XT-PIC  cascade
9:   585855      XT-PIC  acpi, uhci-hcd, uhci-hcd, Intel 82801BA-ICH2, eth0
12:    52        XT-PIC  i8042
14:   35615      XT-PIC  ide0
15:  378371      XT-PIC  ide1
NMI:    0
LOC:  35635761
ERR:    0
MIS:    0

```

## Repräsentation von Handlerfunktionen

Für jede Handlerfunktion existiert eine Instanz der Struktur `irqaction`, die wie folgt definiert ist:

```
struct irqaction {
    void (*handler)(int, void *, struct pt_regs *);
    unsigned long flags;
    const char *name;
    void *dev_id;
    struct irqaction *next;
};
```

<interrupt.h>

Das wichtigste Element der Struktur ist die Handlerfunktion selbst, die in Form des Zeigers `handler` an erster Stelle steht: Sie wird vom Kern aufgerufen, wenn ein Gerät über einen IRQ die Unterbrechung des Systems beantragt und der Interrupt-Controller dies durch Auslösen eines Interrupts an den Prozessor weitergeleitet hat. Bei der Registrierung von Handlerfunktionen werden wir genauer auf die Bedeutung der Argumente eingehen.

`name` und `dev_id` dienen zur genauen Identifikation eines Interrupt-Handlers: Während `name` ein kurzer String ist, der als Kennung für das Gerät verwendet wird (beispielsweise `“100“  
“ncr53c8xx“` etc.), ist `dev_id` ein Zeiger auf eine beliebige Datenstruktur, die das Gerät unter allen Datenstrukturen des Kerns eindeutig identifiziert – beispielsweise die `net_device`-Instanz einer Netzwerkkarte. Diese Angabe wird beim Entfernen einer Handlerfunktion benötigt, wenn sich mehrere Geräte einen IRQ teilen und die IRQ-Kennzahl alleine nicht ausreicht, um ein Gerät zu identifizieren.

`flags` ist eine Flagvariable, die einige Eigenschaften des IRQ (und des damit assoziierten Interrupts) mit Hilfe eines Bitmaps beschreibt, auf dessen einzelne Elemente wie üblich über vordefinierte Konstanten zugegriffen werden kann. Leider gibt es keinen einheitlichen Satz, der für alle Plattformen gültig ist; um den Besonderheiten der einzelnen Architekturen gerecht werden zu können, definiert sich jeder CPU-Typ spezifische Konstanten in `<asm-arch/signal.h>`<sup>7</sup>

Folgende Konstanten treten auf praktisch allen Architekturen auf:

- `SA_SHIRQ` ist bei *shared irqs* (geteilten IRQs) gesetzt: Dies signalisiert, dass mehr als ein Gerät eine IRQ-Leitung verwenden.
- `SA_SAMPLE_RANDOM` wird gesetzt, wenn der IRQ zum Entropie-Pool der Kerns beiträgt.<sup>8</sup>
- `SA_INTERRUPT` gibt an, dass der IRQ-Handler mit *deaktivierten* Interrupts ausgeführt werden soll.

`next` wird verwendet, um geteilte IRQ-Handler zu realisieren: Mehrere `irqaction`-Instanzen werden in einer einfach verketteten Liste verbunden; alle Elemente einer Kette müssen die gleiche IRQ-Nummer behandeln (Instanzen zu unterschiedlichen Kennzahlen befinden sich im `irq_desc`-Array an verschiedenen Positionen). Wie wir in Abschnitt 11.1.5 besprechen werden, traversiert der Kern die Liste, wenn ein geteilter Interrupt ausgelöst wurde, um herauszufinden,

<sup>7</sup> Die verwendeten Konstanten sind nicht in einer Interrupt- oder IRQ-spezifischen Datei deklariert, sondern finden sich in einem Headerfile, das eigentlich für die Signalverarbeitung zuständig ist. Dies unterstreicht die Tatsache, dass Interrupts aus Kernsicht deutliche Analogien zu Signalen aus Benutzersicht aufweisen: In beiden Fällen wird ein laufendes Programm unvorhergesehen von einer höheren Instanz unterbrochen.

<sup>8</sup> Die Informationen werden verwendet, um relativ sichere Zufallszahlen zu generieren, die aus `/dev/random` bzw. `/dev/urandom` ausgelesen werden.

für welches Gerät er eigentlich gedacht war. Vor allem auf Laptops, die viele verschiedene Zubehörgeräte (Netzwerk, USB, FireWire, Soundkarte etc.) auf einem einzigen Chip (mit nur einem Interrupt) integrieren, können solche Handlerketten aus rund fünf Elementen bestehen. Im Normalfall ist für jeden IRQ aber nur ein einziges Gerät registriert.

Abbildung 11.3 zeigt eine Übersicht der besprochenen Datenstrukturen, um ihr Zusammenspiel deutlich werden zu lassen. Da auf einem System üblicherweise nur ein Typ eines Interrupt-Controllers verwendet wird, zeigen die `handler`-Elemente aller `irq_desc`-Einträge auf die gleiche Instanz von `hw_irq_controller`.

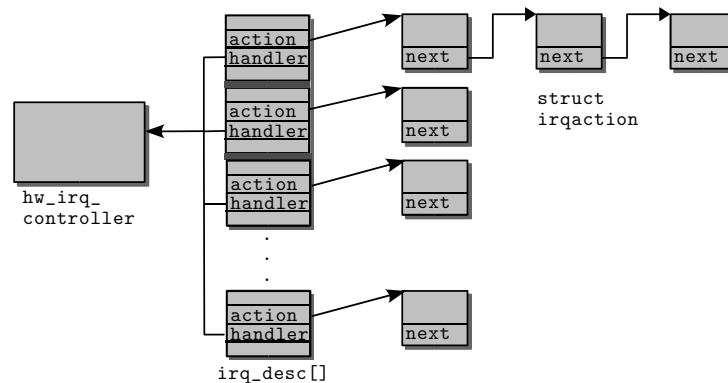


Abbildung 11.3: Datenstrukturen der IRQ-Verwaltung

### Registrierung von IRQs

Die dynamische Registrierung einer ISR durch einen Gerätetreiber kann mit Hilfe der vorgestellten Datenstrukturen recht leicht und vor allem hardwareunabhängig erfolgen, was eine unbedingte Voraussetzung für die Programmierung plattformunabhängiger Treiber ist. Auch wenn die dafür zuständige Funktion `request_irq` nicht in den allgemeinen Kernelquellen implementiert wird, sondern von den Prozessor-spezifischen Teilen abgehandelt werden muss, ist sie dennoch überall ziemlich ähnlich aufgebaut. Der Prototyp ist in allen Fällen absolut identisch:

```

arch/arch/kernel/  int request_irq(unsigned int irq,
irq.c              irqreturn_t (*handler)(int, void *, struct pt_regs *),
                  unsigned long irqflags,
                  const char * devname,
                  void *dev_id)
  
```

Abbildung 11.4 auf der gegenüberliegenden Seite zeigt das Codeflussdiagramm für eine generalisierte Variante von `request_irq`, die die zentralen Punkte aller Plattformen<sup>9</sup> beschreibt.

Zunächst erzeugt der Kern eine neue Instanz von `irqaction`, die mit den Funktionsparametern ausgefüllt wird. Besonders wichtig ist dabei natürlich die Handlerfunktion `handler`. Die weiteren Arbeiten werden an die Funktion `setup_irq` delegiert, die folgende Schritte ausführt:

- Wenn `SA_SAMPLE_RANDOM` gesetzt ist, trägt der Interrupt zur Entropiequelle des Kerns bei, die für den Zufallszahlengenerator in `/dev/random` verwendet wird. `rand_initialize_`

<sup>9</sup> Bis auf Sparc und Sparc64, die ein völlig anderes Konzept zur IRQ-Verwaltung verwenden.

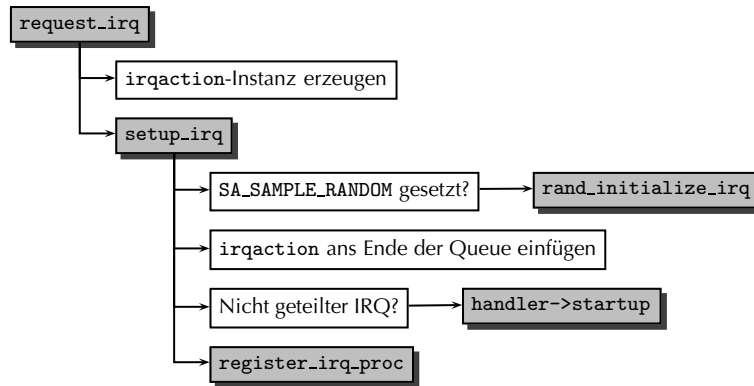


Abbildung 11.4: Codeflussdiagramm für request\_irq

irq fügt den IRQ in die entsprechenden Datenstrukturen ein, auf die wir aber nicht näher eingehen wollen.

- Die von request\_irq erzeugte irqaction-Instanz wird ans Ende der Liste für Routinen einer bestimmten IRQ-Kennzahl eingefügt, die durch irq\_desc[*NUM*]->action angeführt wird. Dadurch stellt der Kern sicher, dass im Fall geteilter Interrupts die Handler in der Reihenfolge ihrer Registrierung aufgerufen werden, wenn ein Interrupt auftritt.
- Wenn der installierte Handler der erste für die jeweilige IRQ-Kennzahl ist, wird die Initialisierungsfunktion handler->startup aufgerufen. Wenn bereits Handler für den IRQ installiert wurden, ist dies nicht notwendig.
- Mit register\_irq\_proc wird im proc-Dateisystem das Verzeichnis /proc/irq/*NUM* erzeugt, wodurch das System sieht, dass der entsprechende IRQ-Kanal verwendet wird.

### Freigeben von IRQs

Die Freigabe von Interrupts verläuft nach dem umgekehrten Schema: Zuerst wird der Interrupt-Controller mittels einer hardwarespezifische Funktion (handler->shutdown) über die Entfernung des IRQs informiert, danach werden die relevanten Einträge aus den allgemeinen Datenstrukturen des Kerns entfernt. Aus Sicht eines Gerätetreibers dient hierzu die plattformspezifische Funktion free\_irq, die auf allen Architekturen eine identische Parametersignatur besitzt und auch überall die gleiche Wirkung hat.

Wenn ein IRQ-Handler eines geteilten Interrupts entfernt werden soll, ist die Kennzahl alleine nicht ausreichend, um den IRQ zu charakterisieren: In diesem Fall muss die weiter oben angesprochene Gerätekenzahl dev\_id zusätzlich verwendet werden, um Eindeutigkeit zu erzielen: Der Kern durchläuft die Liste aller registrierten Handler so lange, bis ein passendes Element (mit übereinstimmendem dev\_id-Element) gefunden wurde. Dann kann der Eintrag entfernt werden.

### Registrierung von Interrupts

Die bisher gezeigten Mechanismen sind nur bei Unterbrechungen sinnvoll, die von einem Zubehörgerät des Systems über eine Interruptanfrage ausgelöst werden. Der Kern muss sich aber

auch um Interrupts kümmern, die entweder vom Prozessor selbst oder über Softwaremechanismen von einem laufenden Benutzerprogramm ausgelöst werden. Im Gegensatz zu IRQs muss der Kernel für diese Art von Interrupt keine Schnittstelle zur Verfügung stellen, um Handler dynamisch registrieren zu können, da die verwendeten Nummern bereits zur Initialisierungszeit bekannt sind und sich im Laufe der Zeit auch nicht ändern: Die Registrierung von Interrupts, Exceptions und Traps wird zur Initialisierungszeit des Kerns vorgenommen, da sich die Belegung zur Laufzeit nicht ändert.

In den plattformspezifischen Kernelquellen diese Problems gibt es nur wenige Gemeinsamkeiten, was angesichts der teilweise großen technischen Unterschiede nicht verwunderlich ist: Auch wenn sich die verschiedenen Varianten manchmal konzeptionell ähnlich sind, unterscheidet sich die konkrete Implementierung zwischen den einzelnen Plattformen sehr stark, da man sich meist hart an der Grenze zwischen C- und Assemblercode bewegt, um den spezifischen Besonderheiten eines Systems gerecht zu werden.

Die größte Gemeinsamkeit zwischen den verschiedenen Plattformen ist ein Dateiname: In `arch/arch/kernel/traps.c` findet sich die systemspezifische Implementierung zur Registrierung von Interrupt-Handlern.

Das Resultat aller Implementierungen ist, dass beim Auftreten eines Interrupts automatisch eine Handler-Funktion aufgerufen wird. Da es für Systeminterrupts kein Interrupt-Sharing gibt, braucht wirklich nur eine Verbindung zwischen Interruptkennzahl und Funktionszeiger hergestellt werden.

Generell können zwei Arten unterschieden werden, mit denen der Kern auf Interrupts reagiert:

- Ein Signal wird an den aktuellen Benutzerprozess geschickt, um diesen über den Fehler zu informieren. Dies ist auf IA-32-Systemen beispielsweise bei einer Division durch 0 der Fall, die durch Interrupt 0 signalisiert wird. Die in diesem Fall automatisch aufgerufene Assembler-Routine `divide_error` schickt das Signal `SIGPFPE` an den Benutzerprozess.
- Der Kern korrigiert die Fehlersituation transparent für den Benutzerprozess. Dies ist auf IA-32-Systemen beispielsweise der Fall, wenn über Interrupt 14 ein Seitenfehler mitgeteilt wird, den der Kern mit den in Kapitel 14 („Swapping“) beschriebenen Methoden korrigieren kann.

### 11.1.5 Abarbeiten von IRQs

Nachdem ein IRQ-Handler registriert wurde, wird die Handlerroutine jedes Mal ausgeführt, wenn ein Interrupt auftritt. Auch hier stellt sich das Problem, die Unterschiede zwischen den verschiedenen Plattformen unter einen Hut zu bringen. Aufgrund der Natur der Dinge beschränken sich die Unterschiede hier allerdings nicht nur auf verschiedene C-Funktionen, die plattformspezifisch implementiert werden müssen, sondern beginnen tief im Reich handoptimierten Assemblercodes, der zur Low-level-Verarbeitung verwendet wird.

Glücklicherweise finden sich dennoch einige strukturelle Ähnlichkeiten zwischen den einzelnen Plattformen: So setzt sich der Ablauf eines Interrupts auf jeder Plattform aus drei Teilen zusammen, wie bereits weiter oben besprochen wurde: Der Entry-Path wechselt vom Benutzer in den Kernmodus, danach erfolgt die Ausführung der eigentlichen Handlerroutine, und schließlich muss der Kern wieder in den Benutzermodus zurückwechseln. Auch wenn viel Assembler im Spiel ist, finden sich zumindest einige C-Abschnitte, die auf allen Plattformen ähnlich sind und die wir in den folgenden Ausführungen genauer betrachten wollen.

### Wechsel in den Kernmodus

Grundlage des Wechsels in den Kernmodus ist Assemblercode, der vom Prozessor automatisch nach jeder Unterbrechung ausgeführt wird. Die Aufgaben diese Codes haben wir bereits weiter oben erwähnt. Die Implementierung findet sich in `arch/arch/kernel/entry.S`, in der üblicherweise verschiedene Einsprungpunkte definiert werden, an die der Prozessor den Kontrollfluss nach dem Auftreten eines Interrupts setzt.

Nur die nötigsten Aktionen werden direkt im Assemblercode durchgeführt. Der Kernel ist bestrebt, so schnell wie möglich zu regulärem C-Code zurückkehren zu können, da dies wesentlich angenehmer zu handhaben ist. Dazu muss eine Umgebung geschaffen werden, die mit den Erwartungen des C-Compilers verträglich ist.

Funktionen werden in C aufgerufen, indem die benötigten Daten – Rücksprungadresse und Parameter – in einer bestimmten Reihenfolge auf dem Stack abgelegt werden. Beim Wechsel zwischen Benutzer- und Kernmodus müssen zusätzlich die wichtigsten Register auf dem Stack gesichert werden, um diese anschließend wiederherstellen zu können. Diese beiden Aktionen werden vom plattformabhängigen Assemblercode durchgeführt; anschließend wird der Kontrollfluss auf den meisten Plattformen an die C-Funktion `do_IRQ` weitergegeben,<sup>10</sup> die ebenfalls plattformabhängig implementiert ist, aber die Situation wesentlich vereinfacht. Je nach Plattform erhält die Funktion entweder die Prozessorregister

```
asmlinkage unsigned int do_IRQ(struct pt_regs regs) arch/arch/kernel/irq.c
```

oder die Kennzahl des Interrupts zusammen mit einem Zeiger auf die Prozessorregister

```
unsigned int do_IRQ(unsigned long irq, struct pt_regs *regs) arch/arch/kernel/irq.c
```

als Parameter. `pt_regs` wird verwendet, um die vom Kern verwendeten Register zu sichern: Die Werte werden (durch Assemblercode) der Reihe nach auf den Stack geschoben und dort belassen, bevor die C-Funktion aufgerufen wird.

`pt_regs` ist genau so definiert, dass die auf dem Stapel enthaltenen Registereinträge mit den Elementen der Struktur zusammentreffen, weshalb die Werte nicht nur für später gespeichert sind, sondern auch vom C-Code ausgelesen werden können. Abbildung 11.5 verdeutlicht die Situation.

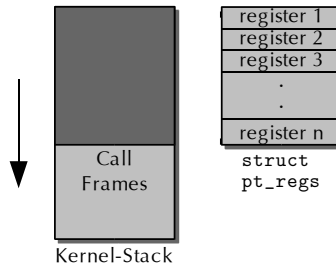


Abbildung 11.5: Stacklayout nach Eintritt in den Kernmodus

Alternativ können die Register auch an eine Position des Adressraum kopiert werden, die nicht mit dem Stack identisch ist. In diesem Fall erhält `do_IRQ` einen Zeiger auf `pt_regs` als

<sup>10</sup> Ausnahmen sind im Wesentlichen Sparc, Sparc64 und Alpha.

Parameter, was nichts an der Tatsache ändert, dass die Registerinhalte gerettet sind und vom C-Code eingesehen werden können.

Die Definition von `struct pt_regs` ist plattformabhängig, da unterschiedliche Prozessoren verschiedene Registersätze zur Verfügung stellen. Der vom Kern benutzte Ausschnitt daraus ist in `pt_regs` enthalten, die nicht darin aufgeführten Register dürfen nur von Usermode-Applikationen verwendet werden. Auf IA-32-Systemen ist `pt_regs` beispielsweise wie folgt definiert:

```
include/asm-i386/    struct pt_regs {
ptrace.h            long ebx;
                   long ecx;
                   long edx;
                   long esi;
                   long edi;
                   long ebp;
                   long eax;
                   int  xds;
                   int  xes;
                   long orig_eax;
                   long eip;
                   int  xcs;
                   long eflags;
                   long esp;
                   int  xss;
};
```

PA-Risc-Prozessoren verwenden einen komplett unterschiedlichen Registersatz:

```
include/            struct pt_regs {
asm-parisc/        unsigned long gr[32]; /* PSW is in gr[0] */
ptrace.h           _u64 fr[32];
                   unsigned long sr[ 8];
                   unsigned long ias{2};
                   unsigned long iao{2};
                   unsigned long cr27;
                   unsigned long pad0; /* available for other uses */
                   unsigned long orig_r28;
                   unsigned long ksp;
                   unsigned long kpc;
                   unsigned long sar; /* CR11 */
                   unsigned long iir; /* CR19 */
                   unsigned long isr; /* CR20 */
                   unsigned long ior; /* CR21 */
                   unsigned long ipsw; /* CR22 */
};
```

Allgemeiner Trend bei 64-Bit-Architekturen sind immer mehr verfügbare Register, weshalb die `pt_regs`-Definitionen immer umfangreicher werden. IA-64 besitzt beispielsweise beinahe 50 Einträge in `pt_regs`, weshalb wir die Definition hier nicht wiedergeben.

Die Kennzahl des aufgetretenen Interrupts wird auf IA-32-Systemen in den höherwertigen 8 Bits von `orig_eax` gespeichert; andere Architekturen finden andere Plätze dafür. Einige Plattformen gehen wie weiter oben erwähnt allerdings den Weg, die Interrupt-Kennzahl als direktes Argument auf den Stack zu legen.

### Aufruf der Handlerroutine

Abbildung 11.6 auf der gegenüberliegenden Seite zeigt ein Codeflussdiagramm von `do_IRQ` für eine verallgemeinerte Variante, die bis auf kleine Details äquivalent zu den verschiedenen Architektur-spezifischen Definitionen ist.

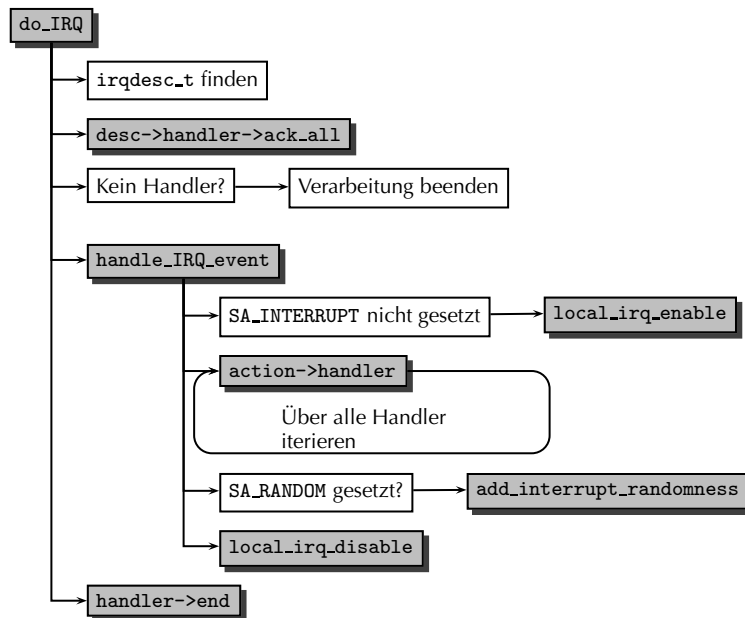


Abbildung 11.6: Codeflussdiagramm für do\_IRQ

Als erster Schritt muss der Kern anhand der IRQ-Kennzahl den passende Eintrag aus der `irq_desc`-Tabelle ausgewählt, indem er anhand der Interrupt-Kennzahl den jeweiligen Array-Eintrag selektiert. Dies liefert die betroffene Instanz von `irq_desc_t`.

Durch Aufruf von `handler->ack` wird die controllerspezifische Acknowledgement-Funktion aufgerufen, die das Eintreffen des IRQs bestätigt und weitere Unterbrechungen dieses Typs maskiert, um bei der Abarbeitung nicht gestört zu werden (die anderen Interrupts des Systems sind davon natürlich nicht betroffen).

Wenn kein Handler für den IRQ registriert ist, kann die Funktion beendet werden; anderenfalls wird die weitere Arbeit an `handle_IRQ_event` delegiert, die zwei Aktionen ausführt:

- Wird `SA_INTERRUPT` in der ersten Handlerfunktion *nicht* gesetzt wurde, werden die Interrupts (für die aktuelle CPU) mit `local_irq_enable` aktiviert, d.h. die Handler können durch andere IRQs unterbrochen werden. Der gerade bearbeitete IRQ ist aber in allen Fällen abgeschaltet!
- Die `action`-Funktion der registrierten IRQ-Handler werden der Reihe nach aufgerufen.
- Wenn `SA_RANDOM` für den jeweiligen IRQ gesetzt ist, wird `add_interrupt_randomness` aufgerufen, um den Zeitpunkt des Auftretens als Quelle für den Entropie-Pool zu verwenden (Interrupts sind als Quelle gut geeignet, wenn sie möglichst zufällig auftreten).
- Mit `local_irq_disable` werden die Interrupts wieder abgeschaltet. Da An- und Abschalten von Interrupts nicht schachtelt, ist es egal, ob sie am Anfang der Bearbeitung aktiviert wurden oder nicht.

Der Kernel hat keine Möglichkeit, herauszufinden, welches Gerät bei einem geteilten IRQ für die Auslösung verantwortlich war. Diese Entscheidung wird vollständig den Handlerroutinen überlassen, die dies anhand gerätespezifischer Register oder anderer Charakteristika der Hardware herausfinden können – die nicht betroffenen Routinen erkennen natürlich ebenfalls, dass die Unterbrechung nicht für sie gedacht war, und geben die Kontrolle so schnell wie möglich wieder an den übergeordneten Code zurück. Ebenso gibt es keine Möglichkeit, mit der eine Handlerroutine an den darüber liegenden Code melden kann, ob die Unterbrechung für sie bestimmt war oder nicht: Der Kern führt immer *alle* Handlerroutinen der Reihe nach aus, unabhängig davon, ob die erste oder die letzte aus der Reihe zum Erfolg führt.

Allerdings kann der Kern prüfen, ob sich *irgendein* Handler für den IRQ verantwortlich gefunden hat. Als Returntyp von Handlerfunktionen ist `irqreturn_t` definiert,<sup>11</sup> die entweder den Wert `IRQ_NONE` oder `IRQ_HANDLED` annehmen kann, je nachdem, ob der IRQ von der Handlerroutine bearbeitet wurde oder nicht.

Während der Abarbeitung aller Handlerroutinen kumuliert der Kern deren Ergebnis über eine Oder-Verknüpfung, weshalb er am Ende feststellen kann, ob der IRQ bearbeitet wurde oder nicht:

```
arch/arch/kernel/ do {
irq.c             status |= action->flags;
                  retval |= action->handler(irq, action->dev_id, regs);
                  action = action->next;
                  } while (action);
```

Wenn `action` anschließend nicht auf `IRQ_HANDLED` steht, wird dies als Fehler gemeldet (normalerweise sollte dies aber nicht vorkommen).

Als letzter Schritt wird (wenn der Kern sich nicht durch die Bearbeitung eines geschachtelten Interrupts noch im Interrupt-Modus befindet) noch `do_softirq` aufgerufen, um eventuell anliegende Software-IRQs abarbeiten zu können. Wir werden diesen Mechanismus in Abschnitt 11.2 genauer besprechen.

## Implementierung von Handlerroutinen

Bei der Implementierung von Handlerroutinen müssen einige wichtige Punkte beachtet werden, die nicht nur die Geschwindigkeit des Systems betreffen: Auch die Stabilität des Computers hängt wesentlich davon ab.

**Einschränkungen** Das Hauptproblem bei der Implementierung von ISRs ist, dass diese im sogenannten *Interrupt-Kontext* ablaufen. Kernelcode kann manchmal sowohl im regulären wie auch im Interrupt-Kontext ausgeführt werden. Um zwischen beiden Varianten unterscheiden und den Code entsprechend gestalten zu können, stellt der Kern die Funktion `in_interrupt` zur Verfügung, die angibt, ob gerade ein Interrupt bearbeitet wird oder nicht.

Der Interrupt-Kontext unterscheidet sich in drei wesentlichen Punkten vom normalen Kontext, in dem der Kern sonst ausgeführt wird:

- Die Ausführung eines Interrupts erfolgt asynchron, kann also zu jeder beliebigen Zeit auftreten. Die Ausführung der Handlerroutine erfolgt daher nicht in einer klar definierten Umgebung, was die Belegung des Userspaces betrifft. Dies verbietet Zugriffe auf den Userspace, vor allem das Kopieren von Speicherinhalten in oder aus dem Benutzer-Adressraum.

<sup>11</sup> Dabei handelt es sich um einen Typdef auf eine Integer-Variable.

Es ist daher für Netzwerktreiber beispielsweise nicht möglich, eingetroffene Daten direkt an die Applikation weiterzureichen, die auf sie wartet, schließlich ist nicht sichergestellt, dass gerade die Applikation läuft, die auf die Daten wartet (dieser Fall ist sogar äußerst unwahrscheinlich).

- Der Scheduler darf im Interrupt-Kontext nicht aufgerufen werden; es ist daher unmöglich, die Kontrolle freiwillig abzugeben.
- Die Handlerroutine darf nicht in den Schlafzustand übergehen. Schlafzustände können nur aufgelöst werden, wenn durch Auftreten eines externen Ereignisses eine Zustandsänderung eintritt, die den Prozess wieder lafbereit macht. Da Interrupts im Interrupt-Kontext aber nicht erlaubt sind, würde der schlafende Prozess ewig auf die erlösenden Neuigkeiten warten. Da auch der Scheduler nicht aufgerufen werden darf, kann kein anderer Prozess ausgewählt werden, um den aktuell schlafenden zu ersetzen.

Natürlich genügt es nicht, nur den direkten Code einer Handlerroutine frei von möglicherweise schlafenden Anweisungen zu halten. Auch alle aufgerufenen Prozeduren und Funktionen (und natürlich auch alle von diesen wiederum aufgerufenen Funktionen usw.) müssen frei von Ausdrücken sein, die sich schlafen legen können. Vor allem bei Kontrollpfaden, die sich auf vielfältigen Wegen verzweigen, ist diese Überprüfung nicht immer einfach und muss sorgsam durchgeführt werden.

**Implementierung von Handlern** Alle Handlerroutinen müssen folgenden Prototyp besitzen:

```
irqreturn_t (*handler)(int irq, void *dev_id, struct pt_regs *regs);
```

`irq` gibt die Kennzahl des IRQs an, `dev_id` ist der Identifikationszeiger, der bei der Registrierung des Handlers übergeben wurde, und `regs` ist eine Abbildung des Registerzustands zum Zeitpunkt des Wechsels in den Kernmodus.

Was sind die Aufgaben einer Handlerroutine? Wenn ein geteilter Interrupt behandelt wird, muss die Routine zunächst prüfen, ob der IRQ überhaupt für sie bestimmt war. Wenn es sich um ein Zubehörgerät neueren Designs handelt, bietet die Hardware eine einfache Möglichkeit zur Überprüfung an, die häufig durch ein spezielles Register des Gerätes realisiert wird: Wenn das Gerät eine Unterbrechung verursacht hat, ist der Wert auf 1 gesetzt. In diesem Fall muss die Handlerroutine den Wert auf die Standardeinstellung (üblicherweise 0) zurücksetzen und die normale Bearbeitung des Interrupts aufnehmen; findet es den Wert 0 vor, ist sichergestellt, dass das verwaltete Gerät nicht die Quelle der Unterbrechung war, weshalb die Kontrolle an den übergeordneten Code zurückgegeben werden kann.

Wenn ein Gerät kein Statusregister dieser Art besitzt, bleibt die Möglichkeit des manuellen Pollings: Bei jedem Auftreten des Interrupts muss der Handler überprüfen, ob Daten am Gerät anliegen; ist dies der Fall, werden diese verarbeitet, ansonsten wird die Routine beendet.

Natürlich kann eine Handlerroutine auch für mehrere Geräte gleichzeitig zuständig sein, beispielsweise zwei Netzwerkkarten des gleichen Typs: Bei beiden Karten wird im Fall eines IRQs der identische Code ausgeführt, da beide Handlerfunktionen auf die gleiche Stelle im Kernelcode zeigen. Wenn beide Geräte unterschiedliche IRQ-Nummern verwenden, kann die Handlerroutine beide daran unterscheiden. Teilen sich die Geräte einen gemeinsamen IRQ, steht immer noch das gerätespezifische Feld `dev_id` zur Verfügung, das die vorhandenen Karten eindeutig voneinander abtrennt.

Der dritte Parameter der Handlerfunktion, in dem die vor dem Aufruf gespeicherten Registerwerte enthalten sind, wird von normalen Gerätetreibern nicht verwendet, ist aber beim Debugging von Kernproblemen dennoch in manchen Fällen nützlich.

## 11.2 Software-Interrupts

Software-Interrupts sind ein Mittel des Kerns, um Arbeiten aufschieben zu können. Da ihre Funktionsweise sich an den eben beschriebenen Interrupts orientiert, sie aber vollständig softwaremäßig implementiert werden, bezeichnet der Kern sie dementsprechend als *SoftIRQs* oder Software-Interrupts.

Eine besondere Situation wird dem Kern mitgeteilt, indem ein Software-Interrupt ausgelöst wird; die Bereinigung der Situation erfolgt mit speziellen Handlerrouninen zu einem späteren Zeitpunkt. Wie wir bereits bemerkt haben, arbeitet der Kernel am Ende von `do_IRQ` alle anstehenden Software-Interrupts ab, so dass für eine regelmäßige Aktivierung gesorgt ist.

Etwas abstrahierter kann man Software-Interrupts daher als Aktivitätsform des Kerns beschreiben, deren Ausführung auf einen späteren Zeitpunkt verlagert wird. Ein hundertprozentiger Vergleich mit Hardware-Interrupts ist trotz der deutlichen Ähnlichkeit allerdings nicht immer ohne weiteres möglich.

Zentrale Komponente des SoftIRQ-Mechanismus ist eine Tabelle mit 32 Einträgen, die Elemente des Typs `softirq_action` aufnimmt. Der Datentyp ist sehr einfach aufgebaut und besteht aus nur zwei Elementen:

```
<interrupt.h> struct softirq_action
                {
                void (*action)(struct softirq_action *);
                void *data;
                };
```

Während `action` ein Zeiger auf die Handlerroutine ist, die beim Auftreten eines Software-Interrupts vom Kernel ausgeführt wird, nimmt `data` einen nicht weiter spezifizierten Zeiger auf, der auf private Daten der Handlerfunktion verweist.

Die Definition der Datenstruktur ist Architektur-unabhängig, ebenso wie die komplette Implementierung des SoftIRQ-Mechanismus: Bis auf die Aktivierung der Abarbeitung werden keine Prozessor-spezifischen Funktionen oder Besonderheiten verwendet, was ein deutlicher Unterschied zu normalen Interrupts ist.

Software-Interrupts müssen registriert werden, bevor der Kernel sie ausführen kann. Zu diesem Zweck wird die Funktion `open_softirq` verwendet, die den neuen SoftIRQ nur an die gewünschte Position in der `softirq_vec`-Tabelle zu schreiben braucht:

```
void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
{
    softirq_vec[nr].data = data;
    softirq_vec[nr].action = action;
}
```

`data` wird bei jedem Aufruf des SoftIRQ-Handlers `action` als Parameter verwendet.

Die Tatsache, dass jeder SoftIRQ mit einer eindeutigen Kennzahl versehen ist, weist bereits darauf hin, dass es sich dabei um eine relativ knappe Ressource handelt, die nicht kreuz und quer von Gerätetreibern und Kernelteilen aller Art verwendet werden sollte, sondern nur wohlüberlegt eingesetzt werden darf. Standardmäßig können nur 32 SoftIRQs auf einem System eingesetzt werden. Diese Beschränkung ist jedoch nicht allzu restriktiv, da SoftIRQs als Basis zur Implementierung anderer Mechanismen eingesetzt werden, die ebenfalls zum Aufschieben von Arbeit verwendet werden und die außerdem besser auf die Bedürfnisse von Gerätetreibern zugeschnitten sind. Wir werden auf die entsprechenden Techniken (Tasklets, Work Queues und Kerntimer) weiter unten eingehen.

Nur der zentrale Kernelcode verwendet Software-Interrupts; bisher werden SoftIRQs nur an wenigen Stellen verwendet, die dafür aber um so wichtiger sind:

```
enum <interrupt.h>
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    SCSI_SOFTIRQ,
    TASKLET_SOFTIRQ
};
```

Zwei dienen zur Implementierung von Tasklets (HI\_SOFTIRQ und TASKLET\_SOFTIRQ), zwei werden für Sende- und Empfangsvorgänge im Netzwerkbereich verwendet (NET\_TX\_SOFTIRQ und NET\_RX\_SOFTIRQ, der Ursprung des SoftIRQ-Mechanismus und dessen wichtigste Anwendung) und eines davon dient für das SCSI-Subsystem (SCSI\_SOFTIRQ). Durch Nummerierung der Softirqs wird eine Prioritätsfolge festgelegt, die zwar nicht die Häufigkeit der Ausführung einzelner Handleroutinen oder deren Priorität gegenüber anderen Aktivitäten des Systems beeinflusst, aber die Reihenfolge festlegt, in der die Routinen ausgeführt werden, wenn mehrere gleichzeitig als aktiv markiert sind.

Um einen Software-Interrupt (ähnlich wie einen normalen Interrupt) auszulösen, wird `raise_softirq(int nr)` verwendet. Die Kennzahl des gewünschten SoftIRQs wird als Parameter übergeben.

Die Funktion setzt das entsprechende Bit in der CPU-spezifischen Variable `irq_stat[smp_processor_id].__softirq_pending`, wodurch der SoftIRQ zwar zur Ausführung markiert, aber noch nicht ausgeführt wird. Durch Verwendung eines Prozessor-spezifischen Bitmaps erreicht der Kern, dass mehrere SoftIRQs – auch identische – auf verschiedenen CPUs gleichzeitig ausgeführt werden können.

Sofern der Aufruf von `cpu_raise_softirq` nicht im Interrupt-Kontext erfolgt ist, wird anschließend mittels `wake_up_softirq` der SoftIRQ-Daemon gestartet, der eine der beiden Alternativen ist, mit denen die Abarbeitung von SoftIRQs angestoßen werden kann. Wir werden in Abschnitt 11.2.2 genauer auf den Daemon eingehen.

### 11.2.1 Starten der SoftIRQ-Verarbeitung

Es gibt mehrere Wege, auf denen die Abarbeitung von SoftIRQs angestoßen werden kann. Alle laufen aber auf den Aufruf der gleichen Funktion hinaus, die deshalb zuerst genauer untersucht werden soll (genau genommen rufen sich die beiden Funktionen wechselseitig auf). Abbildung 11.7 auf der nächsten Seite zeigt das entsprechende Codeflussdiagramm.

Zuerst stellt die Funktion sicher hat, dass sie sich *nicht* im Interrupt-Kontext (womit natürlich ein Hardware-Interrupt gemeint ist); sollte dies der Fall sein, wird sie sofort beendet. Da SoftIRQs zur Ausführung von weniger zeitkritischen Teilen von ISRs verwendet werden, darf es nicht vorkommen, dass der Code in einem Interrupt-Handler selbst aufgerufen wird.

Mit Hilfe von `local_softirq_pending` wird das Bitmap aller gesetzten SoftIRQs der aktuellen CPU ermittelt und das Originalbitmap auf 0 zurückgesetzt, also alle SoftIRQs gelöscht. Beide Aktionen finden mit (auf dem aktuellen Prozessor) deaktivierten Interrupts statt, um die Modifikation des Bitmaps durch störende andere Prozesse zu verhindern; der nachfolgende Code läuft hingegen wieder mit aktivierten Interrupts ab. Das Originalbitmap kann auf diese Weise während der Abarbeitung der SoftIRQ-Handler jederzeit modifiziert werden!

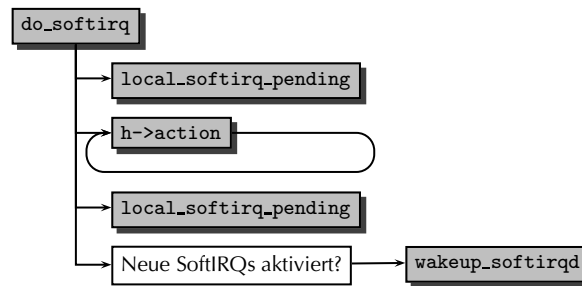


Abbildung 11.7: Codeflussdiagramm für `do_softirq`

In einer `while`-Schleife werden die `action`-Funktionen aus `softirq_vec` für jeden aktivierten SoftIRQ aufgerufen.

Nachdem alle markierten SoftIRQs abgearbeitet wurden, prüft der Kernel, ob zwischenzeitlich im Originalbitmap neue SoftIRQs markiert worden sind. Dabei muss mindestens ein SoftIRQ darunter sein, der in der vorhergehenden Runde noch nicht bearbeitet wurde! Wenn dies der Fall ist, werden die markierten SoftIRQs wieder der Reihe nach abgearbeitet. Dieser Vorgang wird so lange wiederholt, bis nach der Ausführung aller Handler keine neuen, bisher noch nicht bearbeiteten SoftIRQs mehr aktiviert sind.

Wenn sich am Ende der Funktion noch markierte Elemente im Bitmap befinden (dabei handelt es sich auf jeden Fall um SoftIRQs, die bereits mindestens einmal abgearbeitet wurden), wird `wakeup_softirqd` aufgerufen, um den SoftIRQ-Daemon zu aktivieren.

### 11.2.2 Der SoftIRQ-Daemon

Die Aufgabe des SoftIRQ-Daemons besteht darin, SoftIRQs asynchron zum restlichen Kernelcode auszuführen. Dafür steht jedem Prozessor des Systems ein eigener Daemon zur Verfügung, der die Bezeichnung `ksoftirqd` trägt.

`wakeup_softirqd` zum Aufwecken des Daemons wird an zwei Stellen des Kerns aufgerufen:

- In `do_softirq`, wie eben besprochen.
- Am Ende von `cpu_raise_softirq` (wenn sich der Kern nicht gerade im Interrupt-Modus befindet).

Die Wakeup-Funktion selbst ist in wenigen Zeilen erledigt: Ein Zeiger auf die `task_struct` des SoftIRQ-Daemons wird mittels einiger Makros aus einer CPU-spezifischen Variable ausgelesen. Wenn der aktuelle Zustand des Tasks nicht ohnehin `TASK_RUNNING` ist, wird er mittels `wake_up_process` wieder in die Reihe der ablaufbereiten Prozesse eingegliedert (siehe Kapitel 2 („Prozessverwaltung“)). Dadurch wird zwar nicht die sofortige Bearbeitung aller anstehenden Software-Interrupts angestoßen; sobald der Scheduler aber nichts Besseres zu tun hat, wird der Daemon (der mit Priorität 19 läuft) ausgewählt.

Die SoftIRQ-Daemonen des Systems werden kurz nach Aufruf von `init` beim Systemstart erzeugt, wozu der in Anhang D („Systemstart“) beschriebene `initcall`-Mechanismus verwendet wird. Jeder Daemon führt nach seiner Initialisierung folgende Endlosschleife aus:

```

static int ksoftirqd(void * __bind_cpu)
...
    for (;;) {
        if (!local_softirq_pending())
            schedule();

        __set_current_state(TASK_RUNNING);

        while (local_softirq_pending()) {
            do_softirq();
            cond_resched();
        }

        __set_current_state(TASK_INTERRUPTIBLE);
    }
...
}

```

kernel/softirq.c

Bei jeder Aktivierung wird zunächst überprüft, ob sich markierte SoftIRQs auf der Warteliste befinden, da ansonsten die Kontrolle durch expliziten Aufruf des Schedulers an einen anderen Prozess übergeben werden kann.

Sind markierte SoftIRQs vorhanden, macht sich der Daemon an deren Abarbeitung: In einer `while`-Schleife werden die beiden Funktionen `do_softirq` und `cond_resched` so lange aufgerufen, bis keine markierten SoftIRQs mehr vorhanden sind. `cond_resched` stellt sicher, dass der Scheduler aufgerufen wird, wenn das Flag `TIF_NEED_RESCHED` beim aktuellen Prozess gesetzt wurde (siehe Kapitel 2). Dies ist möglich, da alle Funktionen mit angeschalteten Hardware-Interrupts ablaufen.

## 11.3 Tasklets und Work Queues

Software-Interrupts sind die leistungsfähigste Möglichkeit, um die Ausführung von Aktivitäten auf einen zukünftigen Zeitpunkt zu verlagern. Dabei handelt es sich aber um den Aufschiebemechanismus, der am kompliziertesten handzuhaben ist: Da die SoftwareIRQs auf mehreren Prozessoren gleichzeitig und unabhängig voneinander abgearbeitet werden können, kann die Handleroutine desselben SoftIRQs auf mehreren CPUs gleichzeitig laufen. Dies ist ein wesentlicher Vorteil für die Leistungsfähigkeit des Konzepts – vor allem die Netzwerkimplementierung gewinnt auf Mehrprozessorsystemen deutlich –, allerdings müssen die Handleroutinen aber so ausgelegt werden, dass sie entweder vollständig reentrant und Thread-sicher implementiert sind oder die kritischen Bereiche mit Spinlocks (oder anderen IPC-Mechanismen, siehe Kapitel 4) schützen, was mit viel Überlegungsarbeit verbunden ist.

Tasklets und Workqueues sind Mechanismen zur verzögerten Ausführung von Arbeit, deren Implementierung auf SoftIRQs basiert, die aber leichter zu verwenden und daher besser für Gerätetreiber (oder auch anderen allgemeinen Kernelcode) geeignet sind.

Bevor wir uns an die Beschreibung der technischen Details machen, muss eine Bemerkung zur Nomenklatur angebracht werden: Aus historischen Gründen wird die Bezeichnung *Bottom Half* für zwei unterschiedliche Dinge verwendet. Zum einen ist damit die untere Hälfte des Codes einer ISR gemeint, die keine zeitkritischen Aktionen durchführen muss. Leider wurde der dafür in früheren Kernelversionen verwendete Mechanismus zur verzögerten Ausführung von Aktionen ebenfalls als Bottom Half bezeichnet, weshalb der Begriff in der Literatur häufig zweideutig verwendet wird. Mittlerweile existieren Bottom Halves als Mechanismus der Kerns nicht mehr: Sie wurden während der Entwicklung von 2.5 entfernt, da mit Tasklets ein wesentlich besserer Ersatz geschaffen wurde.

### 11.3.1 Tasklets

Tasklets sind „kleine Tasks“, die eine kurze Aufgabe erfüllen, für die ein vollständiger Prozess zu viel Aufwand wäre.

#### Erzeugen von Tasklets

Die zentrale Datenstruktur jedes Tasklets trägt den bezeichnenden Namen `tasklet_struct` und ist wie folgt definiert:

```
<interrupt.h> struct tasklet_struct
                {
                    struct tasklet_struct *next;
                    unsigned long state;
                    atomic_t count;
                    void (*func)(unsigned long);
                    unsigned long data;
                };
```

Das aus Sicht eines Gerätetreibers wichtigste Element ist `func`: Der Zeiger verweist auf die Adresse einer Funktion, deren Ausführung aufgeschoben werden soll. `data` wird bei der Ausführung als Parameter an die Funktion übergeben.

`next` ist ein Zeiger zum Aufbau einer einfach verketteten Liste aus `tasklet_struct`-Instanzen. Dies ermöglicht das Queuing mehrerer Tasklets, die auf ihre Ausführung warten.

`state` gibt – ähnlich wie bei einem echten Task – den aktuellen Zustand des Tasklets an. Allerdings gibt es hier nur zwei Möglichkeiten, die durch je ein eigenes Bit in `state` repräsentiert werden, weshalb sie unabhängig voneinander gesetzt und entfernt werden können:

- `TASK_STATE_SCHED` wird gesetzt, wenn das Tasklet im Kern registriert wird.
- Ein Tasklet im Zustand `TASK_STATE_RUNNING` wird gerade ausgeführt.

Der zweite Zustand ist nur auf SMP-Systemen von Bedeutung, da er beim Schutz des Tasklets vor der parallelen Ausführung auf mehreren Prozessoren gleichzeitig verwendet wird.

Der atomare Zähler `count` kann verwendet werden, um bereits geschedulte Tasklets wieder zu deaktivieren: Wenn sein Wert ungleich 0 ist, wird das entsprechende Tasklet bei der nächsten Ausführung aller wartenden Tasklets einfach ignoriert.

#### Registrieren von Tasklets

`tasklet_schedule` dient dazu, ein Tasklet im System zu registrieren. Funktion `tasklet_schedule`. Wenn das `TASK_STATE_SCHED`-Bit gesetzt ist, wird der Vorgang abgebrochen, da das Tasklet in diesem Fall bereits registriert wurde. Anderenfalls wird das Tasklet an den Anfang einer Liste gesetzt, die von der CPU-spezifischen Variable `tasklet_vec` als Listenkopf angeführt wird und alle registrierten Tasklets enthält, wobei das `next`-Element zur Verknüpfung verwendet wird.

Nachdem ein Tasklet registriert wurde, wird die Tasklet-Liste zur Abarbeitung markiert.

#### Abarbeitung von Tasklets

Der wichtigste Schritt im Leben eines Tasklets ist die Abarbeitung. Da Tasklets auf Basis von Software-IRQs implementiert werden, wird die Abarbeitung immer dann durchgeführt, wenn Software-Interrupts abgearbeitet werden.

Tasklets sind mit dem SoftIRQ `TASKLET_SOFTIRQ` verknüpft. Deshalb genügt der Aufruf von `cpu_raise_softirq(smp_processor_id(), TASKLET_SOFTIRQ)`, um die Tasklets des aktuellen Prozessors bei der nächsten Gelegenheit abzuarbeiten. Als Action-Funktion des SoftIRQs verwendet der Kern `tasklet_action`.

Die Funktion ermittelt zunächst die CPU-spezifische Liste, auf der die zur Ausführung markierten Tasklets verkettet sind, lenkt den Listenkopf auf ein lokales Element um und entfernt dadurch alle Einträge aus der öffentlichen Liste. Anschließend werden sie in folgender Schleife der Reihe nach abgearbeitet:

```
static void tasklet_action(struct softirq_action *a)                                kernel/softirq.c
...
    while (list) {
        struct tasklet_struct *t = list;
        list = list->next;

        if (tasklet_trylock(t)) {
            if (!atomic_read(&t->count)) {
                clear_bit(TASKLET_STATE_SCHED, &t->state);
                t->func(t->data);
                tasklet_unlock(t);
                continue;
            }
            tasklet_unlock(t);
        }
        ...
    }
    ...
}
```

Die Abarbeitung von Tasklets in einer `while`-Schleife ähnelt dem Mechanismus, der bei der Verarbeitung von SoftIRQs eingesetzt wurde.

Da ein Tasklet immer nur auf einem Prozessor zugleich ausgeführt werden darf, andere Tasklets aber dennoch parallel laufen dürfen, ist ein Tasklet-spezifisches Locking erforderlich. Als Lockingvariable wird der Zustand `state` verwendet. Bevor die Handler-Funktion eines Tasklets ausgeführt wird, überprüft der Kern mit `tasklet_trylock`, ob der Zustand des Tasklets `TASKLET_STATE_RUN` ist, ob es also bereits auf einem anderen Prozessor des Systems abgearbeitet wird:

```
static inline int tasklet_trylock(struct tasklet_struct *t)                       <interrupt.h>
{
    return !test_and_set_bit(TASKLET_STATE_RUN, &(t->state));
}
```

Wenn das entsprechende Bit noch nicht gesetzt ist, wird dies nun erledigt.

Wenn das `count`-Element ungleich 0 ist, gilt das Tasklet als deaktiviert. Der Code wird in diesem Fall nicht ausgeführt.

Erst wenn beide Prüfungen erfolgreich überstanden sind, führt der Kernel durch Aufruf von `t->func(t->data)` die Handlerfunktion des Tasklets mit dem entsprechenden Funktionsparameter aus. Anschließend wird mit `tasklet_unlock` das `TASKLET_SCHED_RUN`-Bit des Tasklets gelöscht.

Wenn während der Ausführung der Tasklets neue Tasklets für den aktuellen Prozessor gequeued wurden, wird der SoftIRQ `TASKLET_SOFTIRQ` aufgerufen, um diese so bald wie möglich abzuarbeiten (da der dafür notwendige Code nicht sehr interessant ist, haben wir ihn oben nicht wiedergegeben).

Neben den normalen Tasklets verwendet der Kern noch eine zweite Tasklet-Sorte „höherer“ Priorität. Ihre Implementierung gleicht der eben vorgestellten bis auf folgende Änderungen Buchstabe für Buchstabe:

- Als SoftIRQ wird `HI_SOFTIRQ` anstelle von `TASKLET_SOFTIRQ` verwendet; die zugehörige Action-Funktion ist `tasklet_hi_action`.
- Die registrierten Tasklets werden in der CPU-spezifischen Variable `tasklet_hi_vec` gequeued, was mit `tasklet_hi_schedule` erledigt wird.

„Höhere Priorität“ bedeutet in diesem Zusammenhang, dass die Abarbeitung des SoftIRQ-Handlers `HI_SOFTIRQ` vor allen anderen Handlern abläuft – also vor allem vor den Netzwerk-handlern, die die Hauptlast der Software-Interrupt-Tätigkeit ausmachen.

Momentan wird diese Alternative nur von wenigen Soundkartentreibern verwendet, da eine zu lange Verzögerung aufgeschobener Aktionen hier zu schlechterer akustischer Qualität der Audioausgabe führen kann.

## 11.4 Wait Queues und Completions

Wait Queues werden verwendet, um Prozessen zu ermöglichen, auf das Eintreten eines bestimmten Ereignisses zu warten, ohne dies ständig pollen zu müssen: Sie legen sich während der Wartezeit schlafen und werden vom Kern automatisch aufgeweckt, nachdem das Ereignis eingetroffen ist. Completions sind darauf aufbauende Mechanismen, die der Kern verwendet, um auf das Ende einer Aktion zu warten. Beide Mechanismen werden vor allem von Gerätetreibern häufig verwendet, wie Kapitel 5 („Gerätetreiber“) zeigt.

### 11.4.1 Wait Queues

#### Datenstrukturen

Jede Wait Queue besitzt ein Kopfelement, das durch folgende Datenstruktur repräsentiert wird:

```
<wait.h> struct __wait_queue_head {
        spinlock_t lock;
        struct list_head task_list;
    };
typedef struct __wait_queue_head wait_queue_head_t;
```

Da Wait Queues auch in Interrupts modifiziert werden können, wird ein Spinlock `lock` verwendet, das vor Manipulationen der Queue belegt werden muss (siehe Kapitel 4 („Interprozesskommunikation und Locking“)). `task_list` ist eine doppelt verkettete Liste, die zur Realisierung der Queue verwendet wird.

Die Elemente auf der Queue sind Instanzen folgender Datenstruktur:

```
<wait.h> struct __wait_queue {
        unsigned int flags;
        struct task_struct * task;
        wait_queue_func_t func;
        struct list_head task_list;
    };
typedef struct __wait_queue wait_queue_t;
```



```

        current->state = TASK_UNINTERRUPTIBLE;

        SLEEP_ON_HEAD
        schedule();
        SLEEP_ON_TAIL
    }

```

Da der Kern noch einige weitere `sleep_on`-Varianten definiert, die alle ähnlich aufgebaut sind, werden Makros zur Implementierung definiert. `SLEEP_ON_VAR` deklariert lokale Variablen und erstellt mit `init_wait_queue_entry` eine neue Instanz von `wait_queue_t`, die den aktuellen Prozess (`current`) repräsentiert:

```

kernel/sched.c  #define SLEEP_ON_VAR                \
                unsigned long flags;    \
                wait_queue_t wait;      \
                init_waitqueue_entry(&wait, current);

```

`SLEEP_ON_HEAD` belegt das notwendige Spinlock und ruft `__add_wait_queue` auf, um den Prozess in die Wait Queue einzugliedern. Aufgrund der Tatsache, dass der Taskzustand vorher auf `TASK_UNINTERRUPTIBLE` gesetzt wurde und anschließend mit `schedule` der Scheduler aktiviert wird, befindet sich der Prozess nun im Schlafzustand und muss explizit wieder aufgeweckt werden, bevor er weiterläuft.

Nachdem der Prozess aufgeweckt wurde, wird der Code in `SLEEP_ON_TAIL` ausgeführt, der nur dafür verantwortlich ist, den Prozess mit der nicht näher besprochenen Hilfsfunktion `__remove_wait_queue` aus der Wait Queue zu entfernen.

Neben `sleep_on` definiert der Kern einige weitere Funktionen, die verwendet werden, um den aktuellen Prozess auf einer Wait Queue schlafen zu legen. Ihre Implementierung ist beinahe identisch mit `sleep_on`:

- `interruptible_sleep_on` verwendet den Taskzustand `TASK_INTERRUPTIBLE`, weshalb der schlafende Prozess durch Signale unterbrochen werden kann.
- `sleep_on_timeout` ruft `schedule_timeout` anstelle von `schedule` auf. Die Funktion definiert vor Aufruf des Schedulers einen Kerntimer, der den Prozess nach einer gegebenen Zeitspanne wieder aufweckt, wenn dies noch nicht erfolgt ist. Dadurch wird verhindert, dass der Prozess ewig schläft.
- `interruptible_sleep_on_timeout` legt den Prozess so schlafen, dass er durch Signale aufgeweckt werden kann, und registriert zusätzlich einen Timeout.

`add_wait_queue_exclusive` arbeitet ebenso wie `add_wait_queue`, fügt den Prozess aber ans Ende der Queue ein und setzt zusätzlich sein Flag auf `WQ_EXCLUSIVE`, dessen Bedeutung gleich klar werden wird.

### Aufwecken

Der Kern definiert eine Reihe von Makros, die zum Aufwecken von Prozessen in einer Wait Queue verwendet werden können. Sie basieren alle auf der gleichen Funktion:

```

<wait.h>  #define wake_up(x)                __wake_up((x), TASK_UNINTERRUPTIBLE |
                TASK_INTERRUPTIBLE, 1)
          #define wake_up_nr(x, nr)  __wake_up((x), TASK_UNINTERRUPTIBLE |
                TASK_INTERRUPTIBLE, nr)

```

```

#define wake_up_all(x)                __wake_up((x),TASK_UNINTERRUPTIBLE |
                                        TASK_INTERRUPTIBLE, 0)
#define wake_up_interruptible(x)     __wake_up((x),TASK_INTERRUPTIBLE, 1)
#define wake_up_interruptible_nr(x, nr) __wake_up((x),TASK_INTERRUPTIBLE, nr)
#define wake_up_interruptible_all(x) __wake_up((x),TASK_INTERRUPTIBLE, 0)

```

`__wait_queue` delegiert die Arbeit nach Belegung des benötigten Locks an `__wake_up_common`:

```

static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
                             int nr_exclusive, int sync)
{
    struct list_head *tmp, *next;

```

`q` selektiert die gewünschte Wait Queue, während `mode` angibt, welche Zustände Prozesse besitzen dürfen, um aufgeweckt zu werden. `nr_exclusive` gibt an, wie viele Tasks mit gesetztem `WQ_FLAG_EXCLUSIVE` aufgeweckt werden sollen.

Anschließend iteriert der Kern über die schlafenden Tasks und ruft ihre Wakeup-Funktion `func` auf:

```

list_for_each_safe(tmp, next, &q->task_list) {
    wait_queue_t *curr;
    unsigned flags;
    curr = list_entry(tmp, wait_queue_t, task_list);
    flags = curr->flags;
    if (curr->func(curr, mode, sync) &&
        (flags & WQ_FLAG_EXCLUSIVE) &&
        !--nr_exclusive)
        break;
}

```

Die Liste wird so lange traversiert, bis entweder keine Tasks mehr vorhanden sind oder die durch `nr_exclusive` gegebene Anzahl exklusiver Tasks aufgeweckt wurde. Die Beschränkung wird verwendet, um das so genannte *Thundering Herd*-Problem zu vermeiden: Wenn mehrere Prozesse auf eine Ressource warten, die nur von einem Benutzer gleichzeitig verwendet werden kann, ist es wenig sinnvoll, alle wartenden Prozesse aufzuwecken, da sich alle bis auf einen ohnehin gleich wieder schlafen legen müssen. `nr_exclusive` generalisiert diese Beschränkung.

Die am häufigsten verwendete Funktion `wake_up` setzt `nr_exclusive` gleich 1 und stellt auf diese Weise sicher, dass nur ein exklusiver Task aufgeweckt wird.

`WQ_FLAG_EXCLUSIVE`-Tasks werden ans Ende der Wait Queue eingefügt, wie weiter oben erwähnt wurde. Die Implementierung stellt dadurch sicher, dass auf gemischten Queues zunächst alle normalen Tasks aufgeweckt werden und anschließend die Beschränkung für exklusive Tasks beachtet wird.

Es ist sinnvoll, alle Prozesse einer Wait Queue aufzuwecken, wenn diese beispielsweise auf das Ende eines Datentransfers warten, da die Daten von mehreren Prozessen gleichzeitig gelesen werden können, ohne sich gegenseitig zu behindern.

## 11.4.2 Completions

Completions ähneln den in Kapitel 4 besprochenen Semaphoren, werden allerdings auf der Basis von Wait Queues implementiert. Wir wollen nur auf ihr Interface eingehen.

`init_completion` wird verwendet, um eine neue Completion-Liste zu erzeugen. Prozesse können sich mit `wait_for_completion` in die Liste eintragen und warten dort (im exklusiven Schlafzustand) so lange, bis ihre Anforderung von irgendeinem Teil des Kerns bearbeitet wurde.

Nachdem die Anforderung von einem anderen Teil des Kerns bearbeitet wurde, muss dort `complete` aufgerufen werden, um die wartenden Prozesse wieder aufzuwecken. Da bei jedem Aufruf nur ein Prozess von der Complete-Liste entfernt wird, muss die Funktion für  $n$  wartende Prozesse genau  $n$  mal aufgerufen werden.

### 11.4.3 Work Queues

Work Queues sind ein weiteres Mittel, um Aktionen auf einen späteren Zeitpunkt zu verschieben. Da sie mit Hilfe von Daemonen im Benutzerkontext ausgeführt werden, dürfen die Funktionen beliebig lange schlafen. Während der Entwicklung von 2.5 wurden Wait Queues als Ersatz für den bisher verwendeten `keventd`-Mechanismus entwickelt.

Für jede Workqueue existiert ein Array mit so vielen Einträgen, wie Prozessoren im System vorhanden sind. In jedem Eintrag werden Arbeiten aufgereiht, die später abgearbeitet werden sollen.

Der Kern erzeugt für jede Workqueue einen neuen Kerneldaemon, in dessen Kontext die aufgeschobenen Arbeiten erledigt werden. Dazu verwendet der Daemon den eben besprochenen Wait Queue-Mechanismus.

Eine neue Wait-Queue wird erzeugt, indem

```
kernel/ struct workqueue_struct *create_workqueue(const char *name)
workqueue.c
```

aufgerufen wird. Das `name`-Argument gibt an, unter welchem Namen der erzeugte Daemon in der Prozessliste sichtbar ist.

Sämtliche Arbeiten, die auf Wait Queues aufgeschoben werden, müssen in Instanzen der Struktur `work_struct` verpackt werden, in der aus Sicht des Workqueue-Benutzers folgende Elemente wichtig sind:

```
<workqueue.h> struct work_struct {
                struct list_head entry;
                void (*func)(void *);
                void *data;
            };
```

`entry` wird wie üblich verwendet, um mehrere `work_struct`-Instanzen in einer verketteten Liste zusammenfassen zu können. `func` ist ein Zeiger auf die Funktion, die aufgeschoben werden soll; sie wird mit `data` als Argument versorgt. Um das Ausfüllen dieser Struktur zu vereinfachen, stellt der Kern das Makro `INIT_WORK` zur Verfügung, das eine bereits bestehende Instanz von `work_struct` mit Funktions- und Datenargument ausfüllt:

```
<work_ #define INIT_WORK(struct work_struct _work, void (*func)(void*)_func, void *_data)
queue.h>
```

Es gibt zwei Möglichkeiten, um eine `work_queue`-Instanz in eine Work Queue einzugliedern.

```
kernel/ int queue_work(struct workqueue_struct *wq, struct work_struct *work)
workqueue.c
```

fügt `work` in die Work-Queue `wq` ein, wobei die Arbeit zu einem nicht genauer definierten Zeitpunkt (wenn der Scheduler den Daemon auswählt) ausgeführt wird.

Um sicherzustellen, dass *mindestens* eine durch `delay` (in Jiffies) angegebene Zeitspanne verstreicht, bevor die aufgeschobene Arbeit ausgeführt wird, kann `queue_delayed_work` verwendet werden:

```
kernel/ int queue_delayed_work(struct workqueue_struct *wq,
workqueue.c struct work_struct *work, unsigned long delay)
```

Die Funktion erzeugt zunächst einen Kerntimer, dessen Timeout in `delayed` Jiffies auftritt. Die zugehörigen Handlerfunktion verwendet anschließend `queue_work`, um die Arbeit ganz normal in die Workqueue einzufügen.

Der Kern erzeugt eine Standard-Waitqueue, die die Bezeichnung `events` trägt. Sie kann von allen Teilen des Kerns verwendet werden, für die es sich nicht lohnt, eine eigene Work-Queue zu erstellen. Um neue Arbeiten auf diese Standard-Queue zu legen, müssen die beiden folgenden Funktionen verwendet werden, auf deren Implementierung wir nicht näher eingehen wollen:

```
int schedule_work(struct work_struct *work)
int schedule_delayed_work(struct work_struct *work, unsigned long delay)
```

kernel/work-queue.c

## 11.5 Kerntimer

Alle bisher behandelten Möglichkeiten zum Aufschieben von Arbeit auf einen späteren Zeitpunkt decken einen Bereich nicht ab: das *zeitgesteuerte* Verschieben von Aufgaben. Natürlich machen die verschiedenen Varianten Angaben über den Zeitpunkt, zu dem ein aufgeschobener Task ausgeführt werden wird (beispielsweise Tasklets bei der Abarbeitung von SoftIRQs), es ist aber nicht möglich, einen genauen Zeitpunkt bzw. ein Zeitintervall anzugeben, nach dessen Ablauf eine aufgeschobene Tätigkeit vom Kernel aufgenommen wird. Die einfachste Anwendung dafür ist offensichtlich die Realisierung von Timeouts, bei denen der Kernel eine bestimmte Zeitspanne lang auf das Eintreffen eines Ereignisses wartet – beispielsweise 10 Sekunden auf den Tastendruck des Benutzers, um eine letzte Möglichkeit zum Abbruch zu bieten, bevor irgendeine schwer wiegende Operation durchgeführt wird. Weitere Anwendungen in Benutzerapplikationen sind breit gestreut; wie üblich werden Systemaufrufe verwendet, um von den entsprechenden Fähigkeiten des Kerns Gebrauch machen zu können. Aber auch der Kernel selbst verwendet Timer für verschiedene Aufgaben; als Beispiel sei nur die Kommunikation eines Gerätes mit seiner anvertrauten Hardware genannt, die in vielen Fällen nach einem zeitlich genau festgelegten Protokoll stattfinden muss. In der TCP-Implementierung kommen sehr viele Timer zum Einsatz, die Auszeiten beim Warten auf Daten vorgeben.

### 11.5.1 Einsatz von Timern

Bevor wir auf die kernelseitige Implementierung von Timern eingehen wollen, werden wir uns kurz mit der Schnittstelle beschäftigen, die der Kern im Bereich des Zeitmanagements in Form von Systemaufrufen zur Verfügung stellt. Nach Aktivierung eines Timers legen sich Prozesse entweder schlafen oder widmen sich anderen Tätigkeiten; durch Versenden eines Signals, wie in Kapitel 4 besprochen, weist der Kern den Benutzerprozess auf das Eintreten des Timeouts hin. In vielen Fällen werden Timer nicht direkt verwendet, sondern in diversen Routinen der Standardbibliothek eingekapselt und für den Anwendungsprogrammierer dadurch transparent gemacht. Auch ist es möglich, nicht nur einmalige Timer zu definieren, sondern auch das periodische Senden von Signalen in bestimmten Abständen anzufordern.

Mit `setitimer` und `alarm` stellt der Kernel zwei Systemaufrufe zur Verfügung, die zur Installation periodischer bzw. einmaliger Timer verwendet werden. `sleep` und `nanosleep` sind die Systemaufrufe, die einen Prozess für eine bestimmte Zeitspanne in den Schlafzustand versetzen; während `sleep` für den Sekundenbereich verwendet wird, kann `nanosleep` Unterbrechungen im Bereich von bis zu 10 Millisekunden durchzuführen – angesichts der Bezeichnung des Systemaufrufs eine etwas verwirrende Tatsache, da man auf den ersten Blick wohl Nanosekundenpräzisi-

on erwartet hätte (es ist allerdings bereits möglich, Zeitintervalle im Nanosekundenbereich beim Aufruf des System-Calls zu spezifizieren).

Folgendes Beispiel installiert einen `alarm`-Timer, der nach 5 Sekunden abläuft und den Prozess über das Signal `SIGALRM` über das Ablaufen des Timers informiert. Unmittelbar nach Installation des Alarm-Timer wird der Kern durch Aufruf von `sleep` in einen 30-sekündigen Schlafzustand versetzt, der durch den Timeout des Alarm-Timers unterbrochen wird:

```
#include<unistd.h>
#include<stdio.h>
#include<signal.h>

void do_alarm(int sig) {
    printf("Alarm!\n");
}

void install_handler() {
    struct sigaction sa, oldsa;
    sa.sa_handler = do_alarm;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGALRM, &sa, &oldsa) < 0) {
        printf("Handler not installed\n");
    }
}

int main() {
    install_handler();
    alarm(5);
    sleep(30);
    return 0;
}
```

Die Ausführung der Applikation führt zu folgender Interaktion in der Shell:

```
wolfgang@meitner> ./alarm
[5 Sekunden warten]
Alarm!
```

Wie erwartet wird der durch `sleep` initiierte 30-Sekunden-Schlaf des Prozesses bereits nach 5 Sekunden durch den Timeout des Alarms unterbrochen. Die Handlerfunktion `do_alarm` quittiert dies mit der Ausgabe von `Alarm!`, danach wird der Prozess beendet.

Nach Ablauf des Timers hat der Kernel automatisch das `SIGALRM`-Bit im Signalbitmap der Taskstruktur des jeweiligen Prozesses gesetzt; beim nächsten Scheduling des Prozesses wird das Signal ausgeliefert und der installierte Handler `do_alarm` ausgeführt. Die Zeitspanne, nach der der Timeout erfolgen soll, wird beim Aufruf des `alarm`-Systemaufrufs (bzw. der entsprechenden Kapselungsroutine der C-Bibliothek) als Argument in Sekunden festgelegt.

Die Verwendung von Intervall-Timern gestaltet sich ähnlich, weshalb wir hier nicht weiter darauf eingehen wollen, sondern auf die entsprechende Literatur zur Systemprogrammierung verweisen.

## 11.5.2 Zeitdomänen

Bei der Verwendung von Timern gibt es drei Möglichkeiten, die unterscheiden, wie die abgelaufene Zeit gezählt werden soll bzw. in welcher Zeitbasis<sup>12</sup> sich ein Timer befindet. Der Kernel stellt folgende Varianten zur Verfügung, die sich nach Ablauf des Timeouts durch verschiedene Signale bemerkbar machen.

<sup>12</sup> Oft auch als Zeitdomäne bzw. *time domain* bezeichnet.

- `ITIMER_REAL` misst die verstrichene Echtzeit, um von der Aktivierung des Timers bis zum Timeout und zur Auslösung des Signals zu gelangen. Die Timerzeit tickt in diesem Fall immer, egal ob sich der Rechner im Kernel- oder Benutzermodus befindet bzw. ob die Applikation, die den Timer verwendet, gerade ausgeführt wird oder nicht. Wie aus obigem Beispiel ersichtlich ist, wird beim Timeout ein Signal des Typs `SIGALRM` gesendet.
- `ITIMER_VIRTUAL` läuft nur während der Zeit ab, in der der Eigentümerprozess des Timers im Benutzermodus ausgeführt wird. Systemzeit, die im Kernel verbracht wird (oder wenn der Prozessor mit einer anderen Applikation beschäftigt ist), wird in diesem Fall ignoriert. Das Erreichen des Timeouts wird über `SIGVTALRM`-Signal mitgeteilt.
- `ITIMER_PROF` rechnet die Dauer des Prozesses, in der er sich entweder im Benutzer- oder Systemmodus befindet – die Zeit läuft also auch weiter, wenn ein Systemaufruf im Auftrag des Tasks ausgeführt wird. Andere Prozesses des Systems werden ignoriert. Das beim Timeout ausgelöste Signal ist `SIGPROF`.

Wie bereits aus dem Namen des Timers ersichtlich wird, liegt sein vorrangiger Anwendungszweck im *Profiling* von Applikationen, bei dem die rechenintensivsten Teile eines Programms gesucht werden, um entsprechend optimiert werden zu können, was vor allem bei wissenschaftlichen oder betriebssystemnahen Applikationen ein wichtiger Punkt ist.

Der Typ des Timers muss – neben der Periodenlänge – bei der Installation des Intervalltimers angegeben werden; in obigem Beispiel wurde `TIMER_REAL` für einen Echtzeittimer gewählt.

Das Verhalten eines Alarm-Timers kann mit Intervall-Timern simuliert werden, indem `ITIMER_REAL` als Timertyp gewählt und der Timer nach Auftreten des ersten Timeouts deinstalliert wird. Intervall-Timer sind daher eine verallgemeinerte Form von Alarm-Timern.

### 11.5.3 Der Timer-Interrupt

Als Zeitbasis für Timer verwendet der Kern den Timer-Interrupt des Prozessors, der in regelmäßigen Abständen auftritt – genau HZ mal pro Sekunde. HZ ist ein Architektur-abhängig definiertes Präprozessorsymbol, das in `<asm-arch/param.h>` zu finden ist. Tabelle 11.1 fasst die Werte zusammen, die auf einzelnen Plattformen verwendet werden. Auf den meisten Prozessoren wird `HZ=100` verwendet.

Tabelle 11.1: HZ-Werte der unterstützten Plattformen

HZ	Architektur
100	SuperH, Mips64, Cris, Sparc, Sparc64, H8300, S390, Mips, PA-Risc, M68k, PPC
1000	IA-32, PA-Risc (PA-20), PPC64, AMD64
1024	Alpha (Rawhide), IA-64
1200	Alpha

Bei jedem „Tick“ des Timer-Interrupts wird automatisch die Funktion `do_timer` aufgerufen, deren Codeflussdiagramm in Abbildung 11.8 auf der nächsten Seite zu sehen ist.

Zunächst wird die globale Variable `jiffies_64`, bei der es sich auf allen Architekturen um eine Ganzzahl-Variablen mit 64 Bits handelt,<sup>13</sup> um 1 erhöht. Dies bedeutet nichts anderes, als dass `jiffies_64` die exakte Anzahl an Timer-Interrupts angibt, die seit dem Start des Systems vergangen sind. Ihr Wert wird mit konstanter Regelmäßigkeit erhöht.

<sup>13</sup> Auf 32-Bit-Prozessoren wird dies erreicht, indem zwei 32 Bit breite Variablen zusammengesetzt werden.

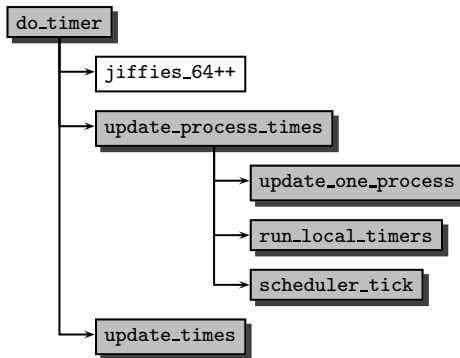


Abbildung 11.8: Codeflussdiagramm für `do_timer`

Aus historischen Gründen gibt es in den Kernelquellen aber eine andere Zeitbasis: `jiffies` ist eine Variable des Typs `unsigned long` und daher auf 32-Bit-Prozessoren nur 4 Bytes breit, was 32 und nicht 64 Bit entspricht. Dabei tritt ein Problem auf: Nach einer längeren Uptime des Systems erreicht der Zähler seinen Maximalwert und muss wieder auf 0 zurückgesetzt werden; diese Situation tritt bei einer Timer-Frequenz von 100 Hz nach etwas weniger als 500 Tagen auf.<sup>14</sup> Bei Verwendung eines 64-Bit-Datentyps tritt das Problem nie auf, denn Uptimes von  $10^{12}$  Tagen sind selbst für einen sehr stabilen Kernel wie Linux etwas utopisch.

Um Effizienzverluste bei der Konvertierung zwischen beiden Zeitbasen zu vermeiden, bedient sich der Kernel eines Tricks: `jiffies` und `jiffies_64` stimmen in den niederwertigen Teilen überein, zeigen also auf dieselbe Speicherstelle bzw. das gleiche Register! Um dies zu erreichen, werden beide Variablen zwar separat voneinander deklariert; das zum Zusammenbinden des fertigen Kerns verwendete Linkerskript bestimmt aber, dass `jiffies` aus den niederwertigen 4 Bytes von `jiffies_64` besteht, wobei je nach Byteordnung des Prozessortyps entweder die ersten oder letzten 4 Bytes verwendet werden müssen. Auf 64-Bit-Maschinen sind beiden Variablen Synonyme für einander.

Die restlichen Aktionen, die bei jedem Timer-Interrupt ausgeführt werden müssen, sind auf zwei Funktionen verteilt:

- `update_times` bringt die Laststatistik des Systems auf den aktuellen Stand, die beispielsweise durch das Kommando `w` ausgegeben wird. Außerdem kümmert sie sich um die Fälle, wenn Timer-Interrupts „verloren“ gehen, was hier aber nicht detaillierter behandelt werden soll.
- `update_process_times` wird verwendet, um die Zeitverwaltung des gerade laufenden Prozesses auf den aktuellen Stand zu bringen, die bei der Implementierung von Timern verwendet wird, die wir im nächsten Abschnitt besprechen. Außerdem ruft sie die aus Kapitel 2 („Prozessverwaltung“) bekannte Funktion `scheduler_tick` auf, die vom Scheduler zur periodischen Kontrolle verwendet wird, ob der aktuelle Prozess sein erlaubtes Zeitquantum überschritten hat.

<sup>14</sup> Die meisten Computer laufen natürlich nicht so lange ohne Unterbrechung, weshalb das Problem auf den ersten Blick etwas marginal erscheint. Dennoch gibt es einige Anwendungen – beispielsweise Server in eingebetteten Systemen –, bei denen Uptimes dieser Größenordnung durchaus erreicht werden können. Für diese Fälle muss sichergestellt werden, dass die Zeitbasis auch hier zuverlässig arbeitet. Während der Entwicklung von 2.5 wurde ein Patch integriert, der den Jiffies-Wert direkt nach Systemstart so setzt, dass ein Wrap nach 5 Minuten Uptime auftritt. Mögliche Probleme können dadurch schnell gefunden werden, ohne jahrelang auf den Wrap warten zu müssen ...

### 11.5.4 Datenstrukturen

Einige Datenstrukturen bilden den Kern der Timer-Verwaltung. Da Intervall-Timer und Alarmer sehr ähnlich sind, brauchen keine unterschiedlichen Strukturen definiert zu werden, beide kommen mit den gleichen Elementen aus. Auch die Implementierung überschneidet sich gegenseitig, wie wir gleich sehen werden.

Zunächst soll betrachtet werden, wie eine Timer-Liste definiert ist:

```

struct timer_list {
    struct list_head entry;
    unsigned long expires;

    void (*function)(unsigned long);
    unsigned long data;

    struct tvec_t_base_s *base;
};

```

<timer.h>

Um registrierte Timer miteinander verknüpfen zu können, wird wie üblich eine doppelt verknüpfte Liste verwendet, deren Listenelement durch `entry` gegeben ist. Die anderen Einträge haben folgende Bedeutung:

- `function` speichert einen Zeiger auf die Callback-Funktion, die nach Ablauf des Timeouts aufgerufen wird.
- `data` wird als Argument für die Callback-Funktion verwendet.
- `expires` bestimmt den Zeitpunkt in Jiffies, zu dem der Timer abläuft.
- `base` ist ein Zeiger auf ein Basiselement, in dem die Timer nach ihrer Ablaufzeit sortiert gespeichert werden (wir gehen gleich genauer darauf ein). Für jeden Prozessor des Systems existiert ein Basiselement, weshalb über `base` die CPU ermittelt werden kann, auf der der Timer abläuft.

Zeiten werden im Kernel in zwei Formaten angegeben, die sich beide auf `jiffies` als Zeitbasis beziehen: entweder als Offset oder als Absolutwert. Während Offsets bei der Installation eines neuen Timers genutzt werden, verwenden alle Datenstrukturen des Kerns Absolutwerte, da diese leicht mit der aktuellen `jiffies`-Zeit verglichen werden können. Auch das `expires`-Element von `timer_list` verwendet eine absolute Zeitangabe und keinen Offset.

Da Programmierer eher in Sekunden als in HZ-Einheiten denken, um Zeitabstände festzulegen, stellt der Kernel eine entsprechende Datenstruktur samt Möglichkeit zur Konvertierung in `jiffies` (und natürlich auch in die andere Richtung) zur Verfügung:

```

struct timeval {
    time_t      tv_sec;      /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};

```

<timer.h>

Die Elemente sind selbsterklärend; die gesamte Zeitspanne wird durch Addition der angegebenen Sekunden- und Mikrosekundenwerte gebildet. Zum Hin- und Herkonvertieren zwischen dieser Darstellung und einem `jiffies`-Wert werden die Funktionen `timespec_to_jiffies` und `jiffies_to_timespec` verwendet, die in `<timer.h>` implementiert sind.

Die Angaben in `timer_list` reichen noch nicht aus, um einen Intervalltimer vollständig zu charakterisieren. In der Taskstruktur jedes Prozesses existieren daher einige Elemente, in denen die fehlenden Informationen untergebracht werden:

```
<sched.h> struct task_struct {
    ...
    unsigned long it_real_value, it_prof_value, it_virt_value;
    unsigned long it_real_incr, it_prof_incr, it_virt_incr;
    struct timer_list real_timer;
    struct list_head posix_timers; /* POSIX.1b Interval Timers */...
}
```

Für jeden Timertyp sind zwei Felder vorbehalten:

- Der Zeitpunkt, zu dem der nächste Timeout erfolgen soll (`it_real_value`, `it_prof_value` und `it_virt_value`)
- Die Periode, mit der der Timer aufgerufen wird. (`it_real_incr`, `it_virt_prof_incr` und `it_virt_incr`).

`real_timer` ist eine Instanz von `timer_list` (*kein* Zeiger darauf), die in die anderen Datenstrukturen des Kerns eingefügt und für die Implementierung von Echtzeit-Timern verwendet wird. Die beiden anderen Timertypen (virtuell und profiling) kommen ohne einen solchen Eintrag aus.

Es ist also nur möglich, genau 3 verschiedene Timer mit *unterschiedlichen* Sorten per Prozess zu besitzen – mehr kann der Kern mit den vorhandenen Datenstrukturen nicht verwalten. Ein Prozess kann beispielsweise einen virtuellen und einen Echtzeit-Timer gleichzeitig laufen lassen, nicht aber zwei Echtzeit-Timer.

### 11.5.5 Dynamische Timer

Der Kern braucht Datenstrukturen, um *alle* im System registrierten Timer verwalten zu können (die sowohl einem Prozess wie auch dem Kern selbst zugewiesen sein können). Die Strukturen müssen eine schnelle und effiziente Kontrolle auf abgelaufene Timer ermöglichen, um nicht zu viel Rechenzeit in Anspruch zu nehmen, da dies bei jedem Timerinterrupt geprüft werden muss.

#### Funktionsweise

Bevor wir die vorhandenen Datenstrukturen im Detail betrachten und auf die Implementierung der Algorithmen eingehen, wollen wir das Prinzip der Timerverwaltung zuerst an einem vereinfachten Beispiel klarmachen, da der vom Kernel verwendete Algorithmus komplizierter ist, als man auf den ersten Blick erwarten würde (die Komplexität wird aber mit einer entsprechenden Leistungsfähigkeit belohnt, die mit einfacheren Algorithmen und Strukturen nicht zu erreichen ist). Abbildung 11.9 auf der gegenüberliegenden Seite zeigt, wie Timer vom Kern verwaltet werden, während Tabelle 11.2 die Intervalle der einzelnen Kategorien angibt.

Tabelle 11.2: Intervalllängen für Timer

Kategorie	Intervall
tv1	0 – 255
tv2	$256 - 2^{14} - 1$
tv3	$2^{14} - 2^{20} - 1$
tv4	$2^{20} - 2^{26} - 1$
tv5	$2^{26} - 2^{32} - 1$

Zum einen muss die Datenstruktur alle Informationen enthalten, die zur Verwaltung von Timern notwendig sind (von den Zusatzdaten, die für prozessspezifische Intervall-Timer gebraucht

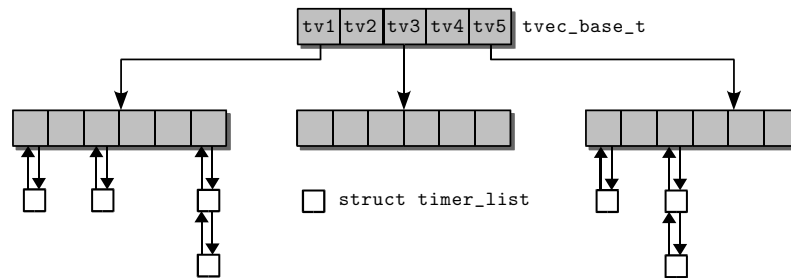


Abbildung 11.9: Datenstrukturen zur Verwaltung von Timern

werden, sehen wir hier vorerst ab); zum anderen muss sie leicht in periodischen Abständen durchsuchbar sein, damit die abgelaufenen Timer ausgeführt und entfernt werden können.

Die hauptsächliche Schwierigkeit besteht darin, die Liste nach Timern zu durchsuchen, die gerade ablaufen oder vor kurzem abgelaufen sind. Da die simple Aneinanderreihung aller `timer_list`-Instanzen nicht zufrieden stellend ist, erstellt der Kern unterschiedliche Kategorien, in die die Timer je nach Ablaufzeit einsortiert werden. Die Grundlage bildet das Hauptarray mit fünf Einträgen, dessen Elemente wiederum aus Arrays bestehen. Die fünf Positionen des Hauptarrays sortieren die vorhandenen Timer grob nach ihren Ablaufzeiten: In der ersten Spalte werden alle Timer gesammelt, deren verbleibende Wartezeit bis zum Timeout noch 0 bis 255 (bzw.  $2^8$ ) Ticks beträgt; die zweite Spalte sammelt alle Timer mit einer verbleibenden Zeit zwischen 256 und  $2^{8+6} - 1 = 2^{14} - 1$  Ticks; das Intervall für die dritte Spalte umfasst den Bereich zwischen  $2^{14}$  und  $2^{8+2*6} - 1$  und so weiter. Die Einträge der Haupttabelle werden als *Grobkategorien* bezeichnet.

Jede Grobkategorie besteht wiederum aus einem Array, in dem die Timer nochmals sortiert werden. Das Array der ersten Grobkategorie besteht aus 256 Elementen, von denen jede Position für einen möglichen `expires`-Wert zwischen 0 und 256 steht. Existieren im System Timer, die den gleichen `expires`-Wert besitzen, werden sie über eine doppelt verknüpfte Standardliste (und das `entry`-Element von `timer_list`) miteinander verknüpft.

Auch die restlichen Grobkategorien bestehen aus Arrays, die allerdings etwas weniger Einträge besitzen, nämlich 64 Stück. Die Array-Einträge nehmen ebenfalls `timer_list`-Instanzen auf, die auf einer doppelt verknüpften Liste verknüpft sind. Allerdings ist pro Array-Eintrag nicht mehr nur ein möglicher Wert von `expires` zugelassen, sondern ein ganzes Intervall. Die Länge des Intervalls hängt von der Grobkategorie ab: Während in der zweiten Kategorie  $256 = 2^8$  aufeinander folgende Timerwerte pro Array-Element zugelassen sind, sind es in der dritten Kategorie  $2^{14}$ , in der vierten  $2^{20}$ , in der fünften  $2^{26}$  und in der fünften und letzten schließlich  $2^{32}$ . Warum die Wahl dieser Intervallgrößen sinnvoll ist, wird deutlich, wenn wir die sukzessive Abarbeitung der vorhandenen Timer im Laufe der Zeit und die damit verbundene Änderung der Datenstruktur betrachten.

Wie geht die Abarbeitung vor sich? Vorrangig muss sich der Kern um die erste Grobkategorie kümmern, da sich darin alle Timer befinden, die in Kürze ablaufen werden. Der Einfachheit halber nehmen wir vorerst an, dass in jeder Grobkategorie ein Zähler vorhanden ist, der die Kennzahl einer Array-Position speichern kann (die tatsächliche Implementierung des Kerns ist funktional gleichwertig, doch wesentlich unanschaulicher, wie wir weiter unten sehen werden).

Der Indexeintrag der ersten Grobkategorie verweist auf das Array-Element, in dem sich die `timer_list`-Instanzen der Timer befinden, die gerade abgearbeitet werden müssen. Bei jedem Aufruf des Timer-Interrupts durchläuft der Kern diese Liste, führt alle Timerfunktionen aus und erhöht die Indexexposition um 1. Die eben ausgeführten Timer werden aus der Datenstruktur

entfernt. Beim nächsten Auftreten des Timer-Interrupts werden die Timer an der neuen Arrayposition ausgeführt, aus der Datenstruktur gelöscht und der Index wieder um 1 erhöht, usw. Wenn alle Einträge abgearbeitet wurden, hat der Index den Wert 255 erreicht. Da die Addition modulo 256 erfolgt, springt der Index wieder in die Ausgangslage (an Position 0) zurück.

Da der Inhalt der ersten Grobkategorie nach spätestens 256 Ticks erschöpft ist, müssen Timer aus den Grobkategorien höherer Ordnung sukzessive nach vorne verschoben werden, um die erste Kategorie wieder aufzufüllen. Nachdem die Indexposition der ersten Grobkategorie auf die Anfangsposition zurückgegangen ist, wird die Kategorie mit allen Timern *eines* Array-Eintrags der zweiten Grobkategorie wieder aufgefüllt. Dies macht die Wahl der Intervallgrößen in den einzelnen Grobkategorien verständlich: Da in der zweiten Grobkategorie 256 verschiedene Zeiten pro Array-Element möglich sind, genügen die Daten aus *einem* Eintrag der zweiten Grobkategorie, um das komplette Array der ersten Grobkategorie auffüllen zu können. Das Gleiche gilt für höhere Ordnungen: Die Daten aus einem Array-Element der dritten Grobkategorie genügen, um die komplette zweite Grobkategorie aufzufüllen, ein Element der vierten reicht für die komplette dritte und ein Element der fünften reicht für die komplette vierte Grobkategorie.

Natürlich werden die Array-Positionen der Grobkategorien höherer Ordnung nicht zufällig ausgewählt; auch hier kommt der besagte Indexeintrag ins Spiel. Sein Wert wird nun allerdings nicht mehr bei jedem Timer-Tick um 1 erhöht, sondern nur bei jedem  $256^{i-1}$ -ten Tick, wobei  $i$  für die Nummer der Grobkategorie steht.

Betrachten wir die Vorgehensweise an einem konkreten Beispiel, das damit beginnt, dass 256 Jiffies abgelaufen sind, nachdem die Abarbeitung der ersten Grobkategorie gestartet wurde, weshalb der Index daraufhin wieder auf 0 zurückgesetzt wird. Zugleich wird der Inhalt des ersten Array-Elements der zweiten Grobkategorie verwendet, um den Datenbestand der ersten Grobkategorie wieder aufzufüllen. Nehmen wir an, dass der `jiffies`-Systemtimer zum Zeitpunkt der Umstellung den Wert 10000 besitzt. Im ersten Element der zweiten Grobkategorie finden sich in einer verketteten Liste Timer, die zu den Zeitpunkten 10001, 10015, 10015 und 10254 ablaufen. Diese werden auf die Array-Positionen 1, 15 und 254 der ersten Grobkategorie verteilt, wobei an Position 15 eine verkettete Liste aus zwei Zeigern eingerichtet wird – schließlich laufen beide zum gleichen Zeitpunkt ab. Nachdem das Umkopieren erledigt ist, wird die Indexposition der zweiten Grobkategorie um 1 erhöht.

Der Zyklus beginnt danach wieder von vorne: Die Timer der ersten Grobkategorie werden der Reihe nach abgearbeitet, bis Indexposition 255 erreicht ist. Alle Timer im zweiten Array-Elements der zweiten Grobkategorie werden verwendet, um die erste Grobkategorie wieder aufzufüllen. Wenn die Indexposition der *zweiten* Grobkategorie 63 erreicht hat (die Grobkategorien besitzen ab der zweiten Ordnung schließlich nur mehr 64 Einträge), wird der Inhalt des ersten Elements der *dritten* Grobkategorie verwendet, um den Datenbestand der zweiten Grobkategorie zu erneuern. Wenn schließlich der Index der dritten Kategorie seinen Maximalwert erreicht hat, werden Daten aus der vierten Kategorie geholt; ebenso überträgt sich die Vorgehensweise auf den Transfer zwischen fünfter und vierter Kategorie.

Um die abgelaufenen Timer ermitteln zu können, muss der Kernel keine riesige Liste mit Timern durchlaufen, sondern kann sich auf die Überprüfung *einer* Array-Position der ersten Grobkategorie beschränken. Da diese meistens leer oder mit nur einem einzigen Timer belegt ist, lässt sich dies schnell durchführen. Auch das gelegentliche Umkopieren von Timern aus den Grobkategorien höherer Ordnung verbraucht nur wenig Rechenzeit, da es durch Zeigermanipulationen effizient durchgeführt werden kann (der Kern braucht keine Speicherblöcke zu kopieren, sondern nur Zeiger mit neuen Werten zu belegen, wie es bei allen Standard-Listenfunktionen üblich ist).

## Datenstrukturen

Der Inhalt der Grobkategorien wird durch zwei einfache Datenstrukturen erzeugt, die sich nur minimal voneinander unterscheiden:

```
typedef struct tvec_s {
    struct list_head vec[TVN_SIZE];
} tvec_t;
kernel/timer.c

typedef struct tvec_root_s {
    struct list_head vec[TVR_SIZE];
} tvec_root_t;
```

Während `timer_vec_root` der ersten Grobkategorie entspricht, repräsentiert `timer_vec` die Grobkategorien höherer Ordnung. Beide Strukturen unterscheiden sich lediglich hinsichtlich der Größe der Array-Elemente, die für die erste Kategorie `TVR_SIZE` beträgt, das auf 256 definiert ist. Alle anderen Kategorien verwenden `TVN_SIZE` Einträge, standardmäßig 64 Stück.

Jeder Prozessor des Systems verfügt über eigene Datenstrukturen zur Verwaltung der Timer, die auf ihm ablaufen. Als Wurzelement wird eine CPU-spezifische Instanz folgender Datenstruktur verwendet:

```
struct tvec_t_base_s {
    unsigned long timer_jiffies;
    tvec_root_t tv1;
    tvec_t tv2;
    tvec_t tv3;
    tvec_t tv4;
    tvec_t tv5;
} ____cacheline_aligned_in_smp;
kernel/timer.c
```

Die Elemente `tv1` bis `tv5` stellen die einzelnen Grobkategorien dar, ihre Funktion sollte aus obiger Beschreibung hervorgehen. Interessant ist das Element `timer_jiffies`: Es hält den (in Jiffies gemessenen) Zeitpunkt fest, bis zu dem alle Timer der Struktur ausgeführt wurden. Besitzt die Variable beispielsweise den Wert 10500, weiß der Kern, dass alle Timer bis zum Jiffies-Wert 10499 abgearbeitet wurden. Normalerweise ist `timer_jiffies` gleich groß oder um 1 kleiner als `jiffies`; die Differenz kann aber (unter sehr hoher Last) auch etwas größer werden, wenn der Kern einige Zeit nicht zur Abarbeitung der Timer kommt.

## Implementierung der Timerverarbeitung

Die Abarbeitung aller Timer wird in `do_timer` angestoßen, indem die Funktion `run_local_timers` aufgerufen wird. Diese beschränkt sich darauf, mit `raise_softirq(TIMER_SOFTIRQ)` den SoftIRQ zur Timer-Verwaltung zu aktivieren, der bei nächster Gelegenheit ausgeführt wird.<sup>15</sup> Als Handlerfunktion des SoftIRQs wird `run_timer_softirq` verwendet, die die CPU-spezifische Instanz von `tvec_t_base_s` auswählt und `__run_timers` aufruft.

`__run_timers` implementiert den weiter oben beschriebenen Algorithmus. Allerdings findet sich in den gezeigten Datenstrukturen nirgends die dringend benötigte Indexposition für die einzelnen Grobkategorien! Der Kern benötigt dazu keine explizite Variable, da alle notwendigen Informationen im `timer_jiffies`-Mitglied von `base` enthalten sind. Dazu werden zunächst folgende Makros definiert:

<sup>15</sup> Da SoftIRQs nicht unmittelbar bearbeitet werden, kann es vorkommen, dass der Kern einige Jiffies lang keine Timerverarbeitung durchführt. Timer können also gelegentlich etwas zu spät ausgelöst, aber nie zu früh aktiviert werden.

```
kernel/timer.c  #define TVN_BITS 6
                #define TVR_BITS 8
                #define TVN_SIZE (1 << TVN_BITS)
                #define TVR_SIZE (1 << TVR_BITS)
                #define TVN_MASK (TVN_SIZE - 1)
                #define TVR_MASK (TVR_SIZE - 1)
                #define INDEX(N) (base->timer_jiffies >> (TVR_BITS + N * TVN_BITS)) & TVN_MASK
```

Die Indexposition der ersten Grobkategorie kann berechnet werden, indem der Wert von `base->timer_jiffies` mit `TVR_MASK` markiert wird:<sup>16</sup>

```
int index = base->timer_jiffies & TVR_MASK;
```

Allgemein kann folgendes Makro verwendet werden, um die aktuelle Indexposition in der Grobkategorie N zu berechnen:

```
#define INDEX(N) (base->timer_jiffies >> (TVR_BITS + N * TVN_BITS)) & TVN_MASK
```

Ungläubige können sich auch hier leicht mit einem kurzen Perl-Skript schnell von der Korrektheit der Bitoperationen überzeugen.

Die Implementierung vollzieht genau das, was weiter oben beschrieben wurde. Dazu wird folgender Code verwendet:

```
kernel/timer.c  while (time_after_eq(jiffies, base->timer_jiffies)) {
                struct list_head work_list = LIST_HEAD_INIT(work_list);
                struct list_head *head = &work_list;
                int index = base->timer_jiffies & TVR_MASK;
```

Falls der Kern in der Vergangenheit einige Timer verschlafen hat, wird dies nachgeholt, indem alle Zeiger abgearbeitet werden, die zwischen dem letzten Ausführungszeitpunkt (`base->timer_jiffies`) und der aktuellen Zeit (`jiffies`) abgelaufen sind.

```
kernel/timer.c  if (!index &&
                (!cascade(base, &base->tv2, INDEX(0))) &&
                 (!cascade(base, &base->tv3, INDEX(1))) &&
                 !cascade(base, &base->tv4, INDEX(2)))
                cascade(base, &base->tv5, INDEX(3));
```

Die Funktion `cascade` wird verwendet, um die Timerlisten mit Timern aus höheren Grobkategorien auszufüllen, wobei wir hier nicht detailliert auf ihre Implementierung eingehen wollen (sie hält sich aber an den weiter oben beschriebenen Mechanismus).

```
kernel/timer.c  ++base->timer_jiffies;
                list_splice_init(base->tv1.vec + index, &work_list);
```

Alle Timer, die sich in der ersten Grobkategorie an der passenden Position für den jeweiligen `timer_jiffies`-Wert befinden (der für den nächsten Durchlauf um 1 erhöht wird), werden in eine temporäre Liste umkopiert und dadurch aus den Originaldatenstrukturen gelöscht.

Anschließend müssen nur noch die einzelnen Handlerrountinen ausgeführt werden:

```
kernel/timer.c  repeat:
                if (!list_empty(head)) {
                    void (*fn)(unsigned long);
```

<sup>16</sup> Leser, die mit den notwendigen Bitoperationen nicht vertraut sind, finden in Anhang C („Anmerkungen zu C“) einige Hinweise dazu. Das gewünschte Verhalten der Rechnung kann man sich aber leicht durch ein kleines Perl-Programm klarmachen:

```
$TVR_MASK = (1 << 8) - 1;
foreach $count (1..100) { print "count, idx: $count " . ($count & $TVR_MASK) . "\n"; };
```

```

        unsigned long data;

        timer = list_entry(head->next,struct timer_list,entry);
        fn = timer->function;
        data = timer->data;

        list_del(&timer->entry);
        timer->base = NULL;
        fn(data);
        goto repeat;
    }
}

```

### 11.5.6 Aktivierung neuer Timer

Bei der Installation neuer Timer muss unterschieden werden, ob sie vom Kernel selbst oder von Applikationen aus dem Userspace benötigt werden. Zunächst soll der Mechanismus für Kerntimer besprochen werden, da Usertimer darauf aufbauen.

`add_timer` wird verwendet, um eine fertig ausgefüllte Instanz von `timer_list` in die eben beschriebenen Strukturen einzufügen. Nach Überprüfung einiger Sicherheitsbedingungen (beispielsweise darf derselbe Timer nicht zweimal eingefügt werden) wird die Arbeit an die Funktion `internal_add_timer` delegiert, deren Aufgabe darin besteht, den neuen Timer an die richtige Stelle der Datenstrukturen einzuordnen.

Der Kernel muss zuerst berechnen, nach wie vielen Ticks der Timeout des neuen Timers erreicht wird, da in der Datenstruktur neuer Treiber ein Absolutwert für den Timeout angegeben wird. Um eventuell versäumte Aufrufe der Timerbearbeitung zu kompensieren, wird `expires - base->timer_jiffies` zur Berechnung des Offsets verwendet.

Anhand dieses Werts können Grobkategorie und Zielposition innerhalb der Kategorie herausgefunden werden; der neue Timer muss nur mehr in die verkettete Liste eingefügt werden. Da er an den *Schluss* der Liste gesetzt wird, die Abarbeitung in `run_timer_list` aber vom Beginn aus erfolgt, wird ein first in, first out-Mechanismus realisiert.

### 11.5.7 Implementierung der timerbezogenen Systemaufrufe

Ausgangspunkt der Systemaufrufe `alarm` und `timer` sind wie üblich die beiden Funktionen `sys_alarm` und `sys_setitimer`. `alarm` installiert Timer des Typs `ITIMER_REAL` (Echtzeittimer), während `setitimer` neben der Installation von Echtzeit Timern auch für virtuelle und Profiling-Timer verwendet werden kann. `do_setitimer` unterscheidet zwischen diesen drei Typen.

Bei der Installation von Echtzeit Timern muss zuerst mittels `del_timer_sync` ein eventuell bereits vorhandener Timer des Prozesses aus der Timerliste entfernt werden; die Installation eines Timers „überschreibt“ deshalb die bisher gültigen Werte. Anschließend werden die Elemente `it_real_value` und `it_real_sync` der Taskstruktur mit den neuen Werten für Ablaufzeit und Intervall versehen, die aus dem Userspace übergeben wurden. Mit `add_timer` wird der neue Timer in die allgemeinen Datenstrukturen des Kerns eingefügt. Nun bleibt nur noch die Frage, wie der Kern die Periodizität des Timers erreicht.

Wie wir in den Beispielen am Anfang diese Kapitels gesehen haben, wird bei Ablauf eines dynamischen Timers keine Handler-Routine im Userspace ausgeführt, sondern lediglich ein Signal erzeugt, was zum Aufruf eines Signalhandlers und damit indirekt zum Aufruf einer Callback-Funktion führt.

Der Kernel hingegen verwendet für alle Userspace-Echtzeit-Timer eine Callback-Funktion: `it_real_fn`. Sie ist für zwei Dinge verantwortlich:

- Das Signal `SIGALRM` wird an den Prozess geschickt, der den Timer installiert hat.
- Um den Timer periodisch zu machen (wenn das `it_real_incr`-Feld der Taskstruktur ungleich 0 ist), wird der Timer neu installiert, wozu eine entsprechend erhöhte `expires`-Zeit verwendet wird.

Da der `alarm`-Systemaufruf periodische Timer mit Periodenlänge 0 erzeugt, werden sie (wie erwartet) nur ein einziges Mal ausgeführt.

Die Timertypen `ITIMER_REAL` und `ITIMER_VIRTUAL` benötigen keinen internen Kernel-Timer, sondern können durch einfache Manipulation der Taskstruktur des Prozesses installiert werden: Es müssen lediglich `it_virt_value` und `it_virt_incr` bzw. `it_prof_value` und `it_prof_incr` mit den Daten der Funktionsparameter gefüllt werden. Die Abarbeitung dieser Timertypen erfolgt direkt im Timer-Interrupt.

### 11.5.8 Verwaltung der Prozesszeiten

`update_one_process` wird verwendet, um die prozess-spezifischen Zeitelemente zu verwalten. Sie wird bei jedem Timer-Interrupt direkt aufgerufen.<sup>17</sup>

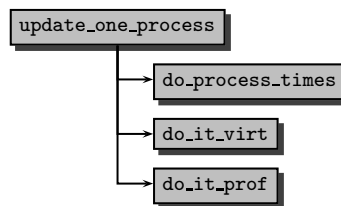


Abbildung 11.10: Codeflussdiagramm für `update_one_process`

Wie das Codeflussdiagramm in Abbildung 11.10 zeigt, müssen drei Dinge erledigt werden:

- `do_process_times` bringt die Werte für die verbrauchte User- und Systemrechenzeit in der Taskstruktur (`utime` und `stime`) auf den aktuellen Stand. Außerdem wird das Signal `SIGXCPU` im Sekundenrhythmus verschickt, wenn der Prozess seine durch `Rlimit` auferlegten Rechenzeitgrenzen überschritten hat.
- `do_it_virt` dekrementiert den Wert des `it_virt_value`-Elements in der Taskstruktur um die Anzahl der im Benutzermodus verstrichenen Ticks. Wenn der Zähler 0 erreicht hat, wird das Signal `SIGVTALRM` verschickt, um den Prozess über den Timeout zu informieren.
- `do_it_prof` dekrementiert das Element `it_prof_value` der Taskstruktur um 1. Da der Timer-Interrupt unabhängig vom Kernel- oder Benutzermodus auftritt, wird durch diese Vorgehensweise die Laufzeit des Prozesses in beiden Modi erfasst, wie es beim Profiling erwünscht ist. Wenn der Zähler auf 0 gefallen ist, wird das Signal `SIGPROF` zur Meldung des Timeouts versandt.

Die beiden letzten Aktionen machen explizite Kerntimer zur Implementierung von virtuellen und Profiling-Timern überflüssig.

<sup>17</sup> Auf Mehrprozessorsystemen wird die Funktion nicht aus dem globalen Timer-Interrupt-Handler des Systems heraus aufgerufen; vielmehr wird eine Variante des Timer-Interrupts verwendet, die per CPU aufgerufen wird. Auf IA-32-Systemen wird dies als IO-APIC-Timer bezeichnet; andere Architekturen verwenden andere Bezeichnungen, worauf wir aber nicht detaillierter eingehen wollen.