Echtzeit-Linux mit dem RT-Preemption-Patch

IT-Klinger — Andreas Klinger
ak@it-klinger.de

22.07.2008

Der RT-Preemption-Patch integriert sich beinahe nahtlos in den Standard-Kernel und bietet Echtzeitfähigkeit. Dieser Artikel beschreibt neue Features des Kernels aus der Perspektive Echtzeit und welche Techniken ihn auf Echtzeit trimmen.

1 Standard-Scheduler kennt Echtzeit-Tasks

Der Default-Scheduler des Kernel wird als „Completely Fair Scheduler“ (CFS) bezeichnet. Hinter dieser Titulierung steckt der Ansatz, daß alle Aufgaben des Systems adäquate Rechenzeit zugewiesen bekommen. Im Grunde genommen ist dies ein Gegensatz zur Echtzeit, bei der wenige Aufgaben soviel Rechenzeit als notwendig bekommen und die verbleibende Rest-Rechenzeit unter den weniger zeitkritischen Tasks aufgeteilt wird.
Der Scheduler klassifiziert zeitkritische Funktionalitäten und weist ihnen die notwendigen Resourcen zu.
An oberster Stelle steht naturgemäß die Interruptbehandlung. Sofern nicht gerade Interrupts abgeschaltet sind, unterbrechen die Interrupt-Service-Routinen alle Tasks des laufenden Systems. Nach Beendigung dieser wird überprüft, ob eine hoch-priorisierte Aufgabe dazugekommen ist, welche schnellstmöglich abzuarbeiten ist.
SoftIRQ-Mechanismen werden sofort abgearbeitet und unterbrichen damit alle darunter liegenden Klassen. Sie werden bezogen auf eine CPU serialisiert abgearbeitet und müssen damit nicht gegen sich selber geschützt werden. Aus dieser Gruppe sind für den Entwickler Tasklets und Timer verwendbar. Die beiden vorgestellten Mechanismen sind zwar gut für zeitnahe Aufgaben geeignet, haben allerdings den Nachteil, daß sie nur in einem Kernel-Treiber Verwendung finden.

Im Userspace kommen an oberster Stelle die sogenannten RT-Tasks (RealTime-Tasks) [1]. Diese unterbrechen alle anderen „normalen“ Tasks und bekommen soviel Rechenzeit, wie sie benötigen. Dazu gibt es 99 Prioritätsstufen (1 - 99), welche

[1] Der Begriff Task bezieht sich sowohl auf Prozesse, als auch auf Threads in einer Multithreading-Anwendung

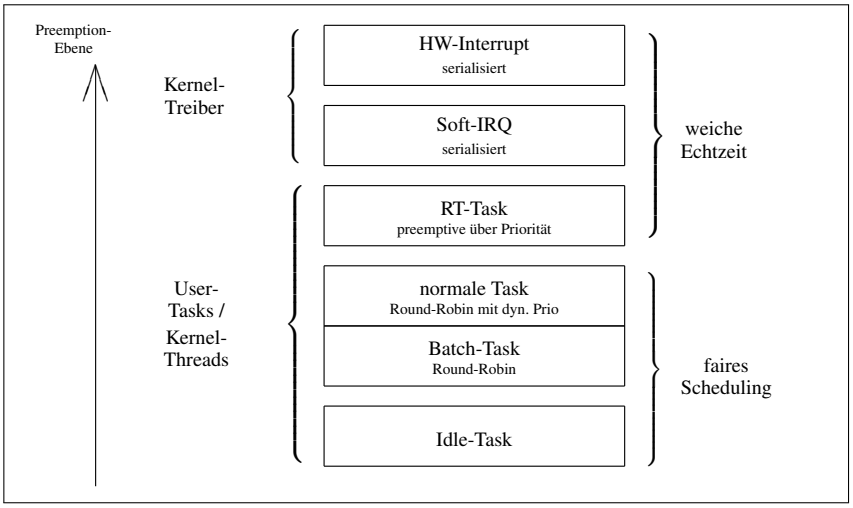

Abbildung 1: Preemption-Ebenen des Standard-Kernel

sich wiederum untereinander unterbrechen. Erst wenn alle RT-Tasks ihre Rechzeit abgeben, kommen normale Tasks zum Zug.
Normale Tasks werden „fair“ untereinander nach einem Round-Robin-Verfahren gescheduled. Jeder Task bekommt Rechenzeit und zwar entsprechend seiner Priorität; der mit der höheren Priorität bekommt mehr und der mit der geringeren weniger.
Für CPU-intensive Aufgaben die im Hintergrund ablaufen können ist die Klasse der Batch-Tasks vorgesehen. Diese bekommen nur dann Rechenzeit, wenn kein anderer Task rechnet, das System sozusagen idle wäre. Mit diesem Mechanismus lassen sich langwierige Aufgaben in den Hintergrund auslagern, ohne daß diese die Reaktionsfähigkeit des Systems negativ beeinträchtigen.
Die IDLE-Klasse wird kernelintern verwendet und wurde der Vollständigkeit wegen dargestellt.

2 hrtimer-Subsystem liefert zeitliche Auflösung

Das neue hrtimer-Subsystem ist eine fundamentale Überarbeitung der alten Timer-Behandlungen und eine essentielle Grundlage für den RT-Preemption-Patch. Die hrtimer vereinen alle Funktionalitäten im Zusammenhang mit zeitlichen Abläufen. Dazu gehören RTC, Datum/Uhrzeit-Behandlung und Zeitsynchronisierung.
Vor allem aber bieten sie ein Framework für den exakten zeitlichen Ablauf. Sind Zeitpunkte bekannt, zu denen ein Ereignis eintreten soll, dann wird der Timer-Interrupt auf dieses programmiert. Damit ist der Kernel-Entwickler nicht mehr an die maximal 1 Millisekunde genauen jiffies angewiesen. Er kann zeitliche Ereignisse mit einer numerischen Granularität von 1 Nanosekunde vorausplanen.
Verwaltet werden diese Ereignisse in einem zeitlich sortierten Red-Black-Tree (rb-

tree). Damit skaliert das Betriebssystem wesentlich besser und der konstante Overhead der vormals periodischen Timer-Behandlung wird auf ein notwendiges Minimum reduziert.
Damit das bestehende, auf einem periodischen Zeittrigger basierende System weiterhin funktioniert, werden die jiffies durch die hrtimer emuliert und die alten Funktionalitäten aufgerufen. Ziel ist es, das alte System komplett abzulösen und den Kernel zukünftig komplett ohne periodische Ticks zu gestalten.

3 Synchronisierung mit RT-Mutexen und RCU's

Auch im Hinblick auf die Synchronisierungsmechanismen wurde einiges verbessert. Dazu gehören vor allem die RT-Mutexe und die RCU's. Bei einem RT-Mutex kann nur ein Ausführungspfad einen kritischen Codeabschnitt betreten. Betritt ein anderer Ausführungspfad den gesicherten Codeabschnitt, wird dieser so lange blockiert, bis derjenige der den Mutex gelockt hat, diesen wieder freigibt.
Was nun, wenn der blockierende Task ein RT-Task ist und dadurch auf unbestimmte Zeit schlafen gelegt wird, wo er doch schnellstmöglich reagieren sollte?
Des Rätsels Lösung ist in der Echtzeitwelt bekannt und wurde auch im Linux-Kernel implementiert. Derjenige Task, welcher einen anderen mit höherer Priorität blockiert wird vorübergehend in seiner Priorität angehoben. Das geschieht genau so lange, bis er den kritischen Abschnitt wieder verlassen hat. Anschließend fällt er auf seine normale Priorität zurück. Durch diese Prioritätsvererbung wird erreicht, daß der niedrig priorisierte Task keinen höheren ausbremsen kann.

Im Gegensatz dazu können sich bei einem RCU (Read-Copy-Update) gleichzeitig viele lesende Tasks in einem kritischen Abschnitt befinden, ohne daß sich diese gegenseitig blockieren.
Ein schreibender Task blockiert auch nicht. Innerhalb des RCU's wird aufgezeichnet, welche Leser sich zum Zeitpunkt des Schreibens im kritischen Bereich befanden. Erst wenn alle betroffenen Leser den kritischen Abschnitt verlassen haben, darf der schreibende Task den nicht mehr benötigten Datenbereich freigeben. Damit wird erreicht, daß alle Leser entweder die alten oder die neuen Daten sehen. Eine Racing-Bedingung in Form von ungültigen intermediären Daten ist nicht möglich. Die Benachrichtigung des Schreibers erfolgt dabei entweder über eine Callback- oder Wartefunktion.
Vorraussetzung für die Verwendung von RCU's sind atomare Zeigeroperationen, worüber alle geschützten Datenbereiche referenziert werden.

4 RT-Preemption-Patch virtualisiert Interrupts

Die bis jetzt genannten Features sind bereits Bestandteil des Kernel-Mainstreams. Sie sind Vorraussetzung für den RT-Patch und erlauben bereits weiche Echtzeit mit Standard-Linux.

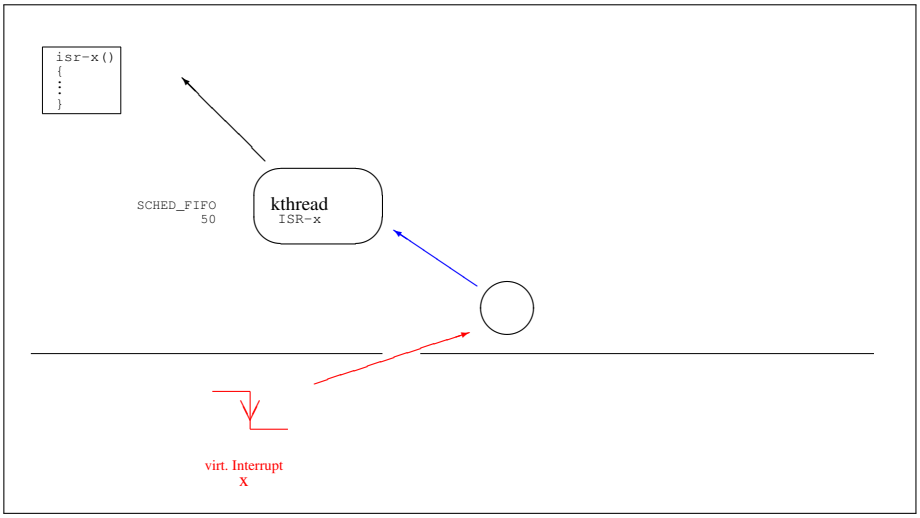

Abbildung 2: Interrupt-Service-Routinen werden durch Kernel-Threads virtualisiert

Mal andersherum betrachtet: Was fehlt zur Echtzeit eigentlich noch? Vor allem wird das Echtzeitverhalten von RT-Tasks durch HW-Interrupts und Sekundärreaktionen darauf behindert.

Der RT-Preemption-Patch trennt Interrupts vom Hardwarekontext so daß er nicht mehr unmittelbar die korrespondierende Interrupt-Service-Routine (ISR) aufruft. In einer zusätzlichen Funktion wird zwischen direkt aufzurufenden ISR und virtualisierten Interrupts unterschieden.
Beim virtualisierten Interrupt wird lediglich dessen Auftreten vermerkt. Der eigentliche Aufruf der ISR erfolgt durch einen dedizierten Kernel-Thread mit der RT-Priorität 50. Ist kein RT-Task mit einer höheren Priorität rechenbereit wird die ISR durch den Kernel-Thread aufgerufen. Ansonsten muß dieser warten, bis kein anderer RT-Task höherer Priorität mehr rechnen möchte. Wird umgekehrt während der Abarbeitung der ISR ein RT-Task höherer Priorität rechenbereit, dann unterbricht dieser den Kernel-Thread. Wie ersichtlich, werden damit ISR's unterbrechbar, was sie normalerweise nicht sind.

Ein Kernel-Modul kann ISR's registrieren, welche nicht virtualisiert, sondern direkt im HW-Kontext ausgeführt werden. Der Interrupt muß dazu mit dem Flag `IRQ_NODELAY` im `desc`-Array gekennzeichnet werden. Der Timer-Interrupt ist ein Beispiel für diesen Typus.

Analog werden auch die SoftIRQ's virtualisiert, indem diese zum Ablaufzeitpunkt markiert, jedoch erst abgearbeitet werden, wenn der entsprechende Kernel-

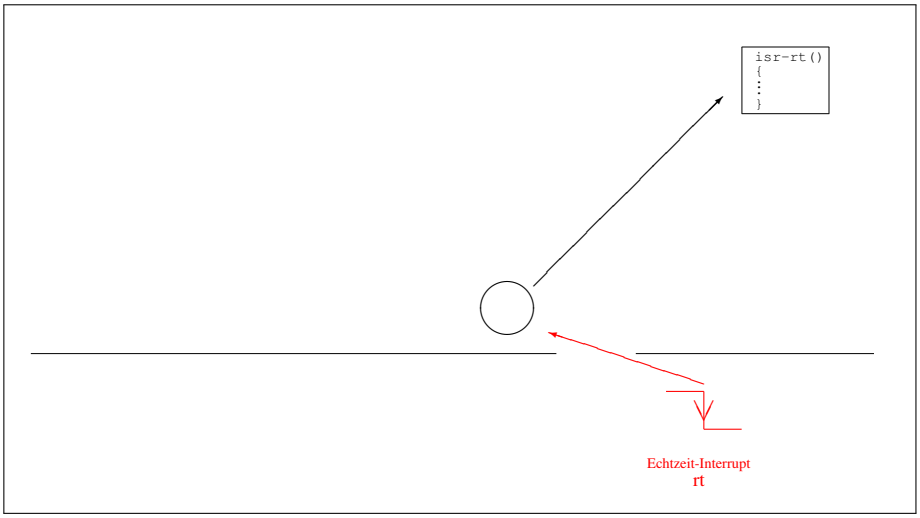

Abbildung 3: Echtzeitrelevante Interrupts werden nicht virtualisiert

Thread Rechenzeit zugewiesen bekommt.

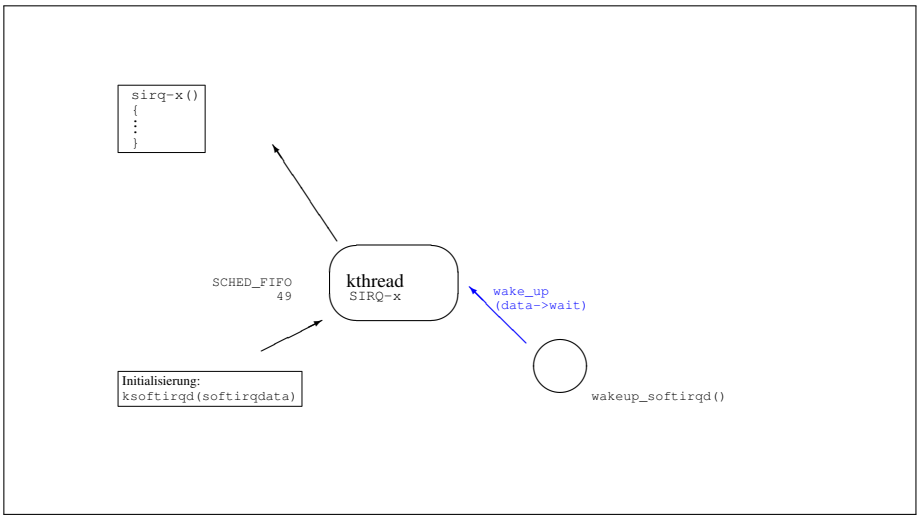

Abbildung 4: SoftIRQ's werden durch Kernel-Threads virtualisiert

Sowohl virtualisierte Interrupts, als auch SoftIRQ's werden jeweils von einem eigenen Kernel-Thread mit der Scheduling-Policy `SCHED_FIFO` und der Priorität 50 gescheduled. Damit können sich RT-Tasks mittels einer Priorität > 50 im System plazieren, wenn diese bevorzugt gescheduled werden sollen.

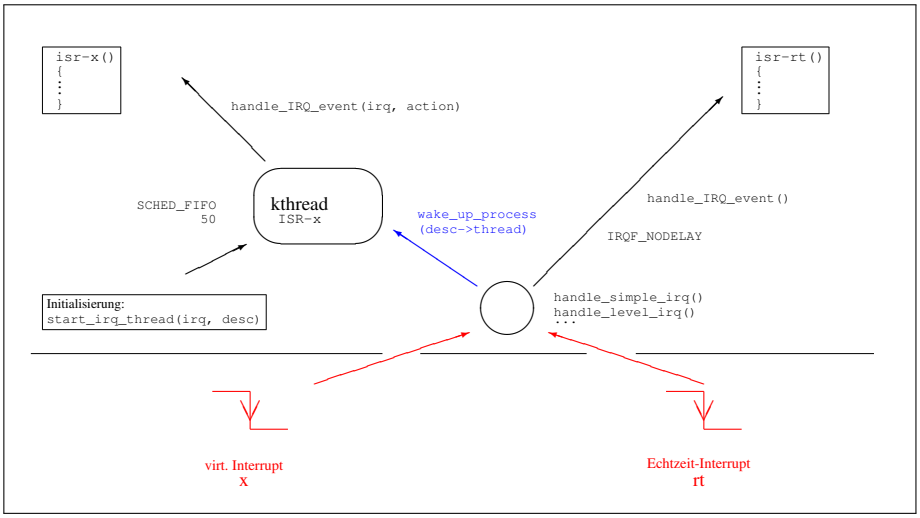

Abbildung 5: Detailansicht der Interrupt- und SoftIRQ-Virtualisierung

5 echtzeitfähiges Scheduling

Wie man aus dem Vergleich der Preemption-Klassen der beiden Abbildungen ablesen kann, liegen die RT-Tasks jetzt direkt unterhalb der nicht virtualisierten Interrupts und vor allen virtualisierten Interrupts und SoftIRQ's. Die Reaktion dieser Tasks kann nun beträchtlich zeitnaher als bisher erfolgen.
Der große Vorteil dieses Systems liegt darin, daß bestehende Linux-Systeme unverändert belassen werden können. Nur an jenen Systemteilen, an denen es notwendig ist, Echtzeitanforderungen zu erfüllen, werden die entsprechenden Anwendungsteile in RT-Tasks und die notwendigen ISR's in NoDelay-Interrupts verwandelt.
Neben den RT-Tasks im Userbereich ist der Entwickler auch in der Lage innerhalb des Kernels Module mit Echtzeitfähigkeiten auszustatten. Dazu legt er lediglich einen Kernel-Thread an und weist diesem eine RT-Scheduling-Policy zu.

6 Fazit

Mit dem vorgestellten RT-Preemption-Patch können gemeinsam mit den schon in den Kernel-Mainstream eingeflossenen Mechanismen Anwendungen mit harter Echtzeit realisiert werden. Auf modernen CPU's sind Antwortzeiten unterhalb der 100 Mikrosekunden-Latte kein unlösbares Problem mehr, wenn man nicht auf ein umfangreiches Betriebssystem wie Linux verzichten möchte.

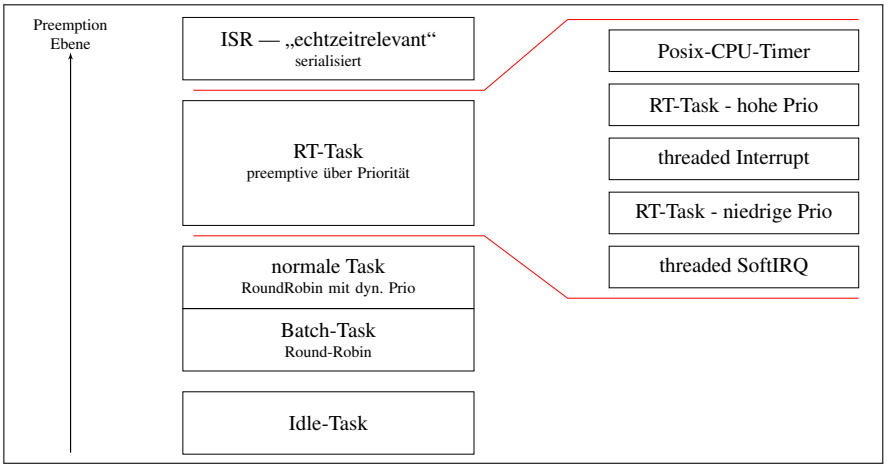

Abbildung 6: Preemption-Ebenen des Echtzeit-Linux-Kernels