Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 34

Kern-Technik

Spinlocks, die nicht spinnen, und Interrupt-Service-Routinen, die im Prozesskontext laufen: Das Realtime-Preemption-Patch stellt die Welt des Kernels scheinbar auf den Kopf. Eva-Katharina Kunst, Jürgen Quade

Virtualisierung auf der einen und Echtzeitverhalten auf der anderen Seite. Das sind die beiden großen Entwicklungsfelder am Linux-Kernel, die zurzeit spürbare Verbesserungen für den Nutzer mit sich bringen. Mit Kernel 2.6.21 hat Linus Torvalds nicht nur das in der Kern-Technik-Folge 31 (siehe **[1]**) beschriebene Dyntick-Patch übernommen, sondern gleich das dort in Aussicht gestellte Tickless-Patch mit dazu.

Doch damit nicht genug. In der Queue der Entwickler Thomas Gleixner und Ingo Molnar stehen weitere Änderungen, die Stück für Stück den Weg in den Mainstream-Kernel finden sollen. Entwickler von Systemen mit hohen Anforderungen an das Zeitverhalten nutzen das entwickelte Patchset »PREEMPT_RT« (**[2]**, **[3]**, **[4]**, **[5]**) bereits seit einiger Zeit als Latenzzeit-Killer und um das Timing berechenbarer zu gestalten. Der Trick: Nicht unterbrechbare Bereiche werden unterbrechbar. Die nicht unterbrechbaren Bereiche existieren in mehreren Formen: als Interrupt-Service-Routinen, als Soft-IRQs und als durch Spinlocks geschützte Codesequenzen.

Alles ein Thread

Die Preempt-Modifikationen stellen damit das bisherige Unterbrechungsverhalten auf den Kopf, mit weitreichenden Konsequenzen für die Kernelprogrammierer. Denn mit einem Mal scheint der Kernel Interrupt-Service-Routinen zu schedulen und Soft-IRQs schlafen zu legen.

Doch das ist in Wahrheit nur babylonische Sprachverwirrung. Denn technisch gesehen stellt der Linux-Kernel weiterhin die vier Unterbrechungsebenen zur Verfügung, wie sie im **Kasten „Unterbrechungsmodell“** dargestellt sind. Nur dass die Entwickler den Code aus den Interrupt-Service-Routinen in hoch priorisierte Threads, die ISR-Threads, verschoben haben. Der zugehörige Code wird damit nicht mehr auf der Interrupt-, sondern auf der Kernel-Ebene abgearbeitet und der dort ablaufende Code ist bekanntlich unterbrechbar. Die richtige ISR hat nur noch die Aufgabe, den zugehörigen ISR-Thread zu aktivieren.

Genauso sieht es mit den Soft-IRQs aus. Den hierzu gehörenden Code haben die Kernelentwickler ebenfalls in eigene Threads verlagert (Soft-IRQ-Threads), die der Scheduler aufgrund ihrer hohen Priorität allerdings bevorzugt zur Abarbeitung auswählt. Im Fall des Falles unterbrechen aber wichtigere Rechenprozesse einen Soft-IRQ-Thread. In Anlehnung an das alte Unix-Motto „Alles ist eine Datei“ heißt es jetzt also zusätzlich „Alles ist ein Thread“.

Auch den Spinlocks rücken die Entwickler zu Leibe. Diese führen unter Umständen zu einer Interrupt-Sperre, sodass hoch priore Rechenprozesse mit engen zeitlichen Anforderungen bisher chancenlos waren. Wenn aber alles ein Thread ist und die zugehörigen Codesequenzen, beispielsweise die ehemaligen Interrupt-Service-Routinen, schlafen können, dann bleibt für Spinlocks nichts weiter als ein kleiner Performance-Vorteil (kein aufwändiger Prozesswechsel bei einem kurzen, gesperrten kritischen Abschnitt) übrig. Den tauschen die Entwickler gerne gegen verkürzte Latenzzeiten ein und verwandeln konsequenterweise Spinlocks im RT_PREEMPT-Patch in Mutexe.

Diese tiefgreifende Modifikation in der Basistechnologie (Spinlock zu Mutex – aktives zu passivem Warten) ist für den Programmierer unsichtbar. Der Compiler tauscht dank ausgeklügelter Makros die Namen der Datentypen und der verwendeten Funktionen gegen Mutex-Typen und -Funktionen aus. So wird aus dem Datentyp »spinlock_t« ein »struct rt_mutex«, und die Funktionen »spin_lock()« und »spin_unlock()« firmieren danach unter »rt_mutex_lock()« und »rt_mutex_unlock()«.

Namensverwirrung

Dass das Kind nicht mehr den richtigen Namen trägt, hat wohl damit zu tun, dass Linus Torvalds Veränderungen am Kernel bekanntlich nur in kleinen Häppchen akzeptiert. Der Austausch der Datentypen und Funktionsnamen wäre zweifelsohne ein dicker Brocken. Dennoch: Wo »rt-mutex« drin ist, sollte auch

»rt-mutex« dranstehen! Alles andere führt über kurz oder lang nur zu Missverständnissen und zu unvorhersehbaren Fehlern.

Die Umwandlung von Mutexen zu Spinlocks führt auf Mehrprozessorsystemen zu einem Nebeneffekt, auf den der Programmierer achtgeben sollte: Während ein Spinlock sicherstellt, dass eine zusammenhängende Codesequenz komplett auf einer CPU abläuft, können beim Einsatz eines Mutex mehrere Prozessoren an der Abarbeitung des Programmabschnitts beteiligt sein. Wer Per-CPU-Variablen in seinem Code einsetzt, sollte also gut aufpassen. Nur wer die Preemption über die in [6] erläuterten Funktionen ausschaltet, schließt zusätzliche Race Conditions aus.

Auch wenn normale Kernelentwickler und Treiberprogrammierer wohl kaum einen Einsatzbereich dafür haben, lässt das PREEMPT_RT-Patch auch weiterhin die klassischen Interrupt-Service-Routinen zu. Um das alte Verhalten zu aktivieren, setzen Programmierer beim Aufruf von »request_irq()« das Flag »IRQF_NODELAY«. Damit läuft die ISR direkt beim Auftreten des Interrupts im Interrupt-Kontext ab.

Sind in einem solchen Fall kritische Abschnitte zu schützen, muss das gute alte Spinlock wieder herhalten, das jetzt unter dem Datentyp Raw-Spinlock (»raw_spinlock_t«) firmiert. In Verbindung mit den bekannten Spinlock-Funktionen werden beim Kompilieren daraus durch die erwähnten Makros echte Spinlocks.

Doch noch einmal zurück zum Realtime-Mutex, das nämlich noch mit einer weiteren Eigenschaft aufwarten kann: Es kennt und unterstützt die so genannte Prioritätsvererbung.

Prioritäts-Inversion und -Vererbung

Als Prioritätsinversion wird eine Situation bezeichnet, in der ein Job mit mittlerer Priorität die Abarbeitung eines Jobs mit hoher Priorität verzögert (siehe Abbildung 2a). Der hoch priore Job (A) kommt typischerweise deshalb nicht zum Zug, weil er einen durch Semaphor oder Mutex geschützten kritischen Abschnitt betreten möchte, den ein Rechenprozess mit niedriger Priorität (C) gerade

Unterbrechungsmodell

Codesequenzen im Linux-Kernel können auf vier unterschiedlichen Ebenen ablaufen (siehe Abbildung 1). Die Abarbeitungsebene entscheidet darüber, welche Funktionen des Kernels zum Einsatz kommen und welche (anderen) Codesequenzen sie unterbrechen dürfen. Die sich durch die Unterbrechung ergebenden kritischen Abschnitte sind durch entsprechende Methoden zu schützen.

Grundsätzlich gilt, dass Code, der auf einer niedrigeren Ebene lauffähig wird, höhere Priorität hat (vertikale Unterbrechbarkeit). Die Abarbeitung des Code auf einer darüberliegenden Ebene wird also zugunsten des lauffähig gewordenen Code unterbrochen. Eine Interrupt Service Routine (ISR) unterbricht einen Systemcall, der auf Kernel-Ebene arbeitet.

Die horizontale Preemption spezifiziert, ob Codesequenzen, die auf der gleichen Ebene ablaufen, sich gegenseitig unterbrechen können. Auf der ISR- und der Soft-IRQ-Ebene gibt es bei Linux typischerweise keine horizontale Unterbrechung. Werden mehrere konkurrierende Codesequenzen auf der Interrupt-Ebene oder der Soft-IRQ-Ebene lauffähig, findet konsequenterweise eine Sequenzialisierung im Sinne eines First Come First Serve (FCFS) statt: Erst wenn die gerade aktive Codesequenz beendet ist, arbeitet der Kernel die konkurrierende Codesequenz ab.

Auf der Kernel-Ebene hat Kernel 2.6 die Kernel-Preemption eingeführt. Dort abzuarbeitende Codesequenzen – Kernel-Threads und Systemcalls – sind priorisiert. Höher priore Jobs unterbrechen demnach niedrig priore Prozesse. Auf der Applikations-Ebene (Ebene 4) schließlich unterbrechen nicht nur die höher prioren Jobs die niedrig prioren, die gleich prioren Rechenprozesse geben nach dem Round-Robin-Prinzip (Zeitscheiben) die CPU nach einer endlichen Verarbeitungszeit ab. Diese Priorisierung arbeitet übrigens Ebenen-übergreifend, sodass ein hoch priorer Job auf Applikationsebene einen weniger wichtigen Job auf der Kernel-Ebene unterbrechen darf.

Lock-Typen

Entwickler müssen das Unterbrechungsmodell kennen, damit sie kritische Abschnitte korrekt schützen können. Der Kernel bietet hierzu diverse Methoden an, die den konkurrierenden Zugriff auf ein Betriebsmittel innerhalb einer Ebene, aber auch von unterschiedlichen Unterbrechungsebenen heraus absichern. Die bekanntesten Methoden sind das Semaphor, das im Linux-Kernel im Wesentlichen in der Variante Mutex eingesetzt wird, und schließlich das Spinlock. Auf Einprozessormaschinen lassen sich Spinlocks allerdings nicht einsetzen. Innerhalb des Kernels setzen Programmierer hier alternativ die Interruptsperre ein. Sie mussten sich um dieses Detail bisher nicht kümmern, das Kernel-Buildsystem hat für sie jeweils den richtigen Code eingesetzt.

Jedoch mussten sie sehr wohl bedenken, auf welcher Ebene die an einem zu schützenden kritischen Abschnitt beteiligten Codesequenzen ablaufen. Schützen beispielsweise eine Interrupt-Service-Routine und Code auf Kernel-Ebene einen kritischen Abschnitt, mussten die Programmierer das Spinlock mit einer Interrupt-Sperre kombinieren (»spin_lock_irq()«). Das einfache Spinlock, repräsentiert durch die Funktion »spin_lock()«, kann den gegenseitigen Ausschluss, also die sequenzielle Abarbeitung, nur dann garantieren, wenn die konkurrierenden Codesequenzen auf unterschiedlichen Prozessoren laufen. Die zusätzliche Interruptsperre schließt damit die Unterbrechung des Code auf Kernel-Ebene durch eine ISR aus, die auf dem gleichen Prozessor abläuft.

Übrigens: Man sollte sich bei der Wahl des Schutzmechanismus eines kritischen Abschnitts nicht allein vom Unterbrechungsmodell und damit von einer scheinbaren Parallelverarbeitung leiten lassen. Mehrprozessorsysteme, die eine echte Parallelverarbeitung bieten, sind heute Standard. Besser ist es also, grundsätzlich davon auszugehen, dass alle Routinen mehrfach parallel ablaufen.

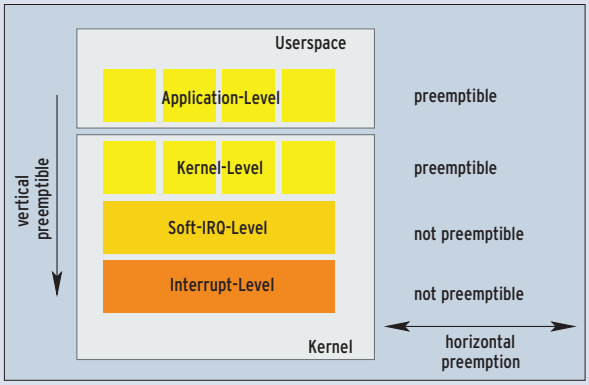


Abbildung 1: Das Unterbrechungsmodell des Kernels: Die Unterbrechbarkeit setzt sich aus horizontaler und vertikaler Preemption zusammen.

bearbeitet. Der an sich unbeteiligte Prozess mit mittlerer Priorität (B) hält dabei den niedrig prioren und somit auch den hoch prioren Job auf.

Zur Entschärfung der Prioritätsinversion setzen die Entwickler auf die so genannte Prioritätsvererbung. Sobald der hoch priore Job (C) die Semaphor- oder Mutex-Operation zum Betreten eines kritischen Abschnitts aufruft und das Semaphor/Mutex durch den niedrig prioren Rechenprozess (C) belegt ist, bekommt dieser die Priorität des hoch prioren Jobs vererbt (siehe Abbildung 2b). So ausgestattet lässt er sich nicht mehr durch den mittel priore Job (B) verdrängen. Gibt er das Semaphor/Mutex frei, nimmt er wieder seine ursprüngliche Priorität an und der hoch priore Job betritt – mit einer signifikant kürzeren Latenzzeit – den kritischen Abschnitt.

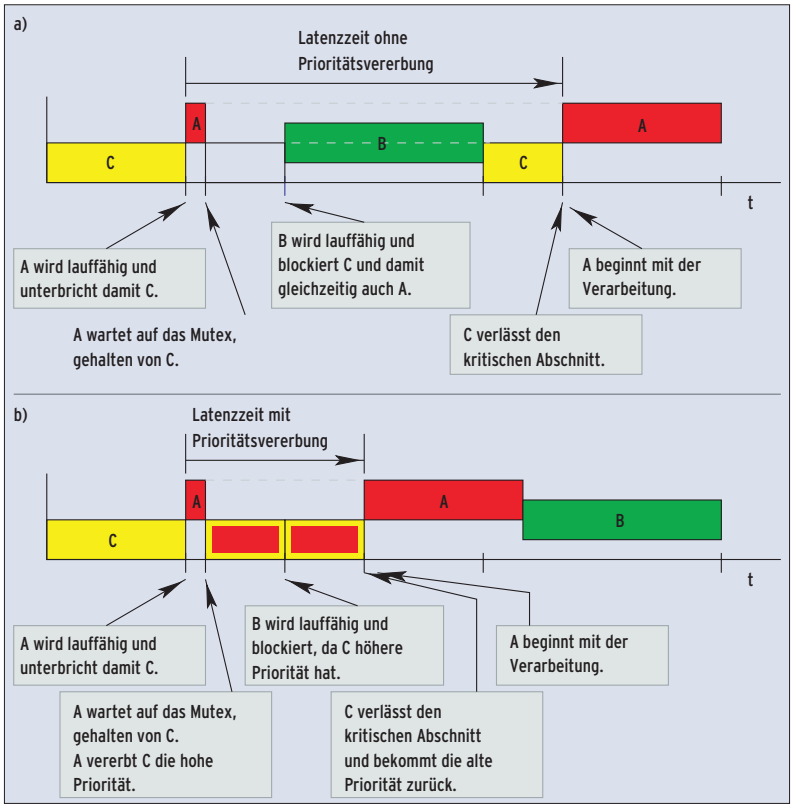


Abbildung 2: Prioritätsinversion und -vererbung: Belegung der CPU durch die drei Jobs A (rot, hohe Priorität), B (grün, mittlere Priorität) und C (gelb, niedrige Priorität).

Praxis

Um in den Genuss der verkürzten Latenzzeiten zu kommen, benötigt der Entwickler die Quellen eines aktuellen Linux-Kernels (zum Beispiel 2.6.21) und das passende PREEMPT_RT-Patch. Der Quellcode zum Linux-Kernel findet sich auf [http://www.kernel.org], das passende PREEMPT_RT-Patch (zum Beispiel »patch-2.6.21-rt1«) unter [7]. Nach dem Entpacken des Kernels im Verzeichnis »/usr/src/«, geht es ans Patchen und Konfigurieren. Auf dem Ubuntu-System der Autoren übernehmen das die Kommandos in Listing 1.

Bei der Kernelkonfiguration muss die Option »Complete Preemption (Real-Time)« aktiviert sein (siehe Abbildung 3). Alternativ lässt sich mit der Option »Preemptible Kernel (Low-Latency Desktop)« die Verschiebung der ISRs und der Soft-IRQs in Threads getrennt per SYS-Filesystem (Dateien »/proc/sys/kernel/softirq_preemption« und »/proc/sys/kernel/hardirq_preemption«) zur Laufzeit des Kernels kontrollieren. Bleibt nur noch, den neuen Kernel zu übersetzen und zu installieren. Unter Ubuntu geht das folgendermaßen:

```
/usr/src/linux-2.6.21# make-kpkg --initrd➥
 buildpackage
/usr/src/linux-2.6.21# cd ..
/usr/src# dpkg -i linux-image-2.6.21-rt1_➥
2.6.21-rt1-10.00.Custom_i386.deb
```

Mit einem folgenden Reboot ist der Latenzzeit-Killer schließlich aktiviert. Das zeigt sich bei einem Blick auf die Taskliste (siehe Abbildung 4). Hier tauchen die Interrupt-Service-Threads ebenso auf wie die Soft-IRQ-Threads. Da es sich nur bei den Prioritäten von 40 bis 139 um Echtzeitprioritäten handelt, laufen diese Threads augenscheinlich mit der mittleren Priorität 90.

Über Prioritäten legt der Anwender fest, dass der Scheduler Rechenprozesse mit hohen Anforderungen an eine kurze Latenzzeit bevorzugt. Das Kommando »chrt« kann die Prioritäten einzelner Threads gezielt anpassen. Unglücklicherweise stimmen aber die mit »ps« angezeigten Prioritäten nicht genau mit denen überein, die das Kommando »chrt« erwartet. Es setzt nämlich die Prioritäten getrennt einmal für den Bereich der normalen und einmal für den Bereich der Realzeit-Prioritäten. Jeder Bereich beginnt wiederum mit Priorität 0. Priorität 90 bei »ps« entspricht Priorität 50 und Scheduling Mode »SCHED_FIFO« bei »chrt«.

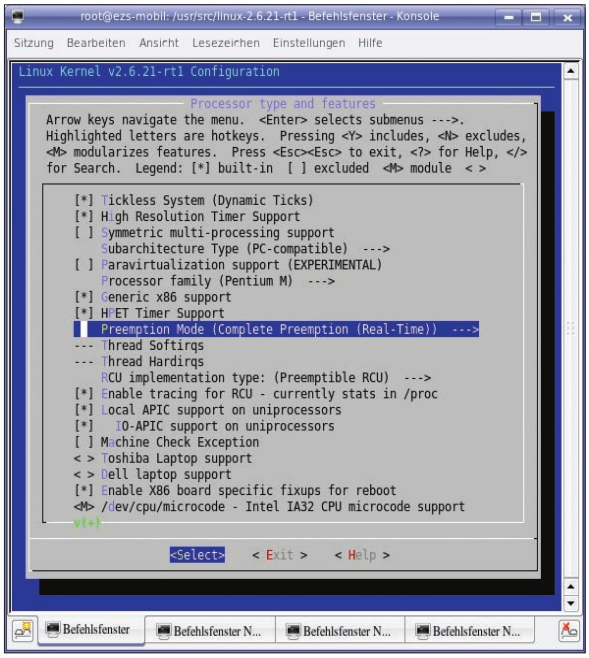
Wer die Priorität des in Abbildung 4 gezeigten Prozesses »IRQ-10« auf 91 anheben will, gibt diesen Befehl ein:

```
sudo chrt -vfp 51 1998
```

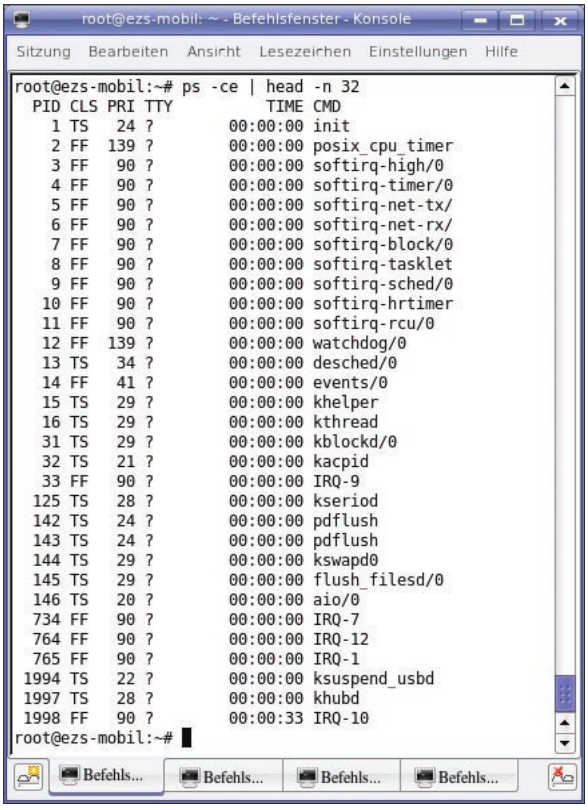
Das RT_PREEMPT-Patch bringt Linux einem richtigen Echtzeit-Betriebssystem einen erheblichen Schritt näher. Für viele Anwendungen sind damit die bisherigen Realtime-Erweiterungen wie RT-Linux und RTAI nicht mehr notwendig. Da auch auf dem Desktop dank Multimedia die Anforderungen an das Echtzeitverhalten zunehmend strenger werden und die Performance-Penalties des vorgestellten Patchsets sehr gering sind,

Listing 1: Realtime-Patch

```
01 /usr/src# wget http://www.kernel.org/pub/linux/
   kernel/v2.6/linux-2.6.21.tar.bz2
02 /usr/src# wget http://people.redhat.com/mingo/
   realtime-preempt/patch-2.6.21-rt1
03 /usr/src# tar xvfj linux-2.6.21.tar.bz2
04 /usr/src# cd linux-2.6.21
05 /usr/src/linux-2.6.21# patch -p1 <../patch-2.6.21-rt1
06 /usr/src/linux-2.6.21# cp /boot/config-`uname -r` .
07 /usr/src/linux-2.6.21# make menuconfig
```



▲ **Abbildung 3: Die richtige Kernelkonfiguration entscheidet über die Echtzeit-Eigenschaften. Der relevante Eintrag heißt »Complete Preemption«.**



▶ **Abbildung 4: Das Kommando »ps« verrät: ISRs und Soft-IRQs tauchen in der Taskliste auf. Die Spalte »PRI« zeigt die Priorität jedes Prozesses.**

spricht nicht viel gegen eine Aufnahme in den Mainline-Kernel. Linus Torvalds sollte allerdings für eine korrekte Namensgebung der verwendeten Synchronisationselemente sorgen – selbst wenn dies eine Textersetzung in vielen Codepassagen bedeuten würde. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“ (31): Linux-Magazin 1/07, S. 110

[2] Initiale Ankündigung der RT-Preemption auf der Kernel-Mailingliste: [http://kerneltrap.org/node/3995]

[3] Linux-Realtime-Wiki: [http://rt.wiki.kernel.org/index.php/Main_Page]

[4] Paul McKenney, „A realtime preemption overview“: [http://lwn.net/Articles/146861/]

[5] Paul McKenney, „SMP and embedded real time systems“: [http://www.linuxjournal.com/article/9361]

[6] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“ (26): Linux-Magazin 2/06, S. 102

[7] Realtime Preemption Patch: [http://people.redhat.com/mingo/realtime-preempt/]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel „Linux Treiber entwickeln“ haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.