Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 31

# Kern-Technik

Mit der Integration der hochauflösenden High-Resolution-Timer lässt Linus Torvalds die schleichende Unterwanderung der traditionellen, auf Jiffies basierenden Zeitverwaltung zu. Die Auswirkungen auf den Linux-Kernel sind gewaltig. Eva-Katharina Kunst, Jürgen Quade


**Im Jahr 1905** stellte Albert Einstein seine spezielle Relativitätstheorie in den „Annalen der Physik“ der Öffentlichkeit vor und beendete damit die Jahrtausende alte Vorstellung, die Zeit verginge für alle gleich. Ein wundersames Phänomen: Zeit ist nicht absolut, sondern relativ und hängt vom Betrachter ab, und erstaunlicherweise stehen Zeit und Raum auch noch in direkter Beziehung zueinander.

Auch der Linux-Kernel baute jahrelang auf eine interne, konstant laufende Zeitbasis. Doch mit der Übernahme des High Resolution Timer Code von Thomas Gleixner und Ingo Molnar – kurz Hrtimer genannt – hat die Relativitätstheorie (wenn auch nicht die von Albert Einstein) auch in den Linux-Kernel Einzug gehalten. Der neue Code bildet nicht nur die Basis für wirklich hoch zeitauflösende Timer, sondern auch für ein beeindruckend hochgenaues Zeitverhalten (High Precision Timer).

## Böser Admin

Ein Problem bei der Zeitzählung im Linux-System ist der ungeduldige Admin, der in Unkenntnis von »ntp« und »adjtimex« mit der Funktion »settimeofday()« über »date« oder »rdate« kleinere oder sogar größere Korrekturen an der aktuellen Systemzeit vornimmt. Dass die Systemuhr bei diesem Vorgehen einen Hüpfer macht, vertragen nicht alle Systemkomponenten gut: Bei einer abrupten Sommerzeitumstellung wird beispielsweise aus einer Sekunde Wartezeit plötzlich mehr als ein Stündchen.

Mit dem neuen Timercode – seit Kernel 2.6.16 im Standardkernel – geht es besser (zur alten Lösung siehe **[1]**). Jetzt kann der Programmierer angeben, ob eine Funktion nach einer angegebenen Zeitdauer (also relativ) aufgerufen werden soll oder zu einem bestimmten Zeitpunkt (absolut). Dazu muss er lediglich ein Hrtimer-Objekt reservieren und bei der Initialisierung dieses Objekt einer von zwei Zeitquellen zuweisen.

## Tick für Tick

Die Zeitquelle »CLOCK_MONOTONIC« zählt unbeirrt Timer-Ticks im System. Auf einem Linux-PC mit neuerem Kernel

**Listing 1: »hrtimer.c«**

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <linux/ktime.h>
04
05 struct hrtimer hrt_monotonic, hrt_realtime;
06
07 static int print_jiffies( struct hrtimer *hrt )
08 {
09    printk("%ld jiffies ( %p )\n", jiffies,
   hrt );
10    return HRTIMER_NORESTART;
11 }
12
13 static int __init mod_init(void)
14 {
15    struct timespec tp;
16    ktime_t tim;
17
18    hrtimer_init( &hrt_monotonic, CLOCK_
   MONOTONIC, HRTIMER_REL );
19    hrtimer_init( &hrt_realtime, CLOCK_
   REALTIME, HRTIMER_ABS );
20    hrt_monotonic.function = print_jiffies;
21    hrt_realtime.function = print_jiffies;
22    tim = ktime_set( 20, 0 );
23    hrtimer_start( &hrt_monotonic, tim,
   HRTIMER_REL );
24    tim = ktime_add( ktime_get_real(), tim );
25    hrtimer_start( &hrt_realtime, tim,
   HRTIMER_ABS );
26    printk("Timer activated at %ld jiffies\n",
   jiffies );
27    return 0;
28 }
29
30 static void __exit mod_exit(void)
31 {
32    hrtimer_cancel( &hrt_monotonic );
33    hrtimer_cancel( &hrt_realtime );
34 }
35
36 module_init( mod_init );
37 module_exit( mod_exit );
38
39 MODULE_LICENSE("GPL");
```

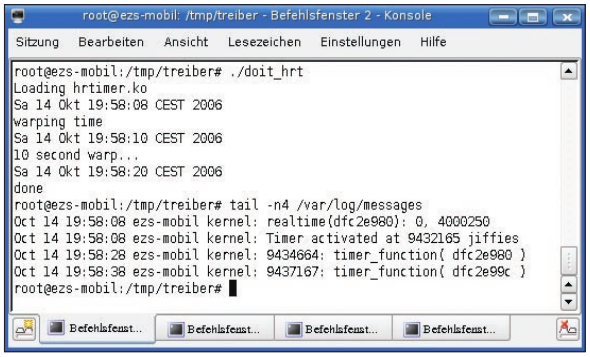


**Abbildung 1: Die Zeitquelle Monotonic berücksichtigt eventuelle Zeitsprünge nicht, die Zeitquelle Realtime hingegen schon.**

kommen pro Minute 250 Zählerpunkte zusammen. Tick für Tick wächst der Zählerstand, selbst wenn der Admin die Uhr kaltblütig zurückdreht. Auf solche Missetaten reagiert die Zeitquelle »CLOCK_REALTIME« aber empfindlich. Beim Vorstellen der Systemuhr warten zugeordnete Timerfunktionen länger. Wird die Uhr zurückgedreht, verkürzt sich die Wartezeit entsprechend.

Der Code in Listing 1 demonstriert dieses unterschiedliche Verhalten und zeigt damit zugleich die Verwendung der Hrtimer. Wie bereits die Namen der Variablen»hrt_monotonic« und »hrt_realtime« vermuten lassen, werden diese beiden Hrtimer-Objekte bei der Initialisierung mit Hilfe der Funktion »hrtimer_init()« den korrespondierenden Zeitquellen »CLOCK_MONOTONIC« und »CLOCK_REALTIME« zugeordnet (Zeilen 18 und 19).

Die Funktion »hrtimer_start()« aktiviert in den Zeilen 23 und 25 beide Timer so, dass sie nach 20 Sekunden die Timerfunktion »print_jiffies()« aufrufen. Der Code lässt sich mit dem Makefile aus Listing 2 übersetzen und mit »insmod hrtimer.ko« in den Kernel laden. Im Syslog ist dann zu sehen, dass die Funktion »print_jiffies()« wie erwartet nach exakt 20 Sekunden zweimal abläuft.

Manipuliert man jedoch während dieser 20 Sekunden die Systemzeit, sieht das Ergebnis anders aus. Abbildung 1 zeigt den Fall, dass der Admin die Systemzeit während des Wartens um 10 Sekunden vorstellt. Der Timer »hrt_realtime« bekommt diese Modifikation mit und verkürzt die Wartezeit um 10 Sekunden. Danach läuft die Funktion »print_jiffies()« ab, die ihre Ausgabe gemäß der im Logfile protokollierten Zeit (»19:58:28«) zum richtigen Zeitpunkt macht.

Der Timer »hrt_monotonic« dagegen ignoriert den Zeithüpfer, wartet genau 20 Sekunden und ruft dann die Funktion auf. Absolut betrachtet kommt dieser Aufruf, wie im Logfile zu sehen, 10 Sekunden zu spät (»19:58:38«). Die exakte Wartezeit ist auch aus dem ebenfalls mit ausgegebenen Jiffie-Wert abzulesen. Während der Monotonic-Timer die erwarteten 5000 Jiffies (»9432165 + 5000«)

mehr anzeigt, hat der Realtime-Timer nur 2500 Jiffies lang gewartet.

Die vom Hrtimer aufgerufene Funktion wird im Kontext des Timer-Softinterrupt abgearbeitet, also im Interrupt-Kontext. Damit steht nur ein eingeschränkter Funktionsumfang bereit. Bei umfangreicheren Aufgaben würde die Timerfunktion also typischerweise eine Workqueue oder einen Kernelthread aufwecken, um die Arbeit zu verrichten.

Ein weiterer neuralgischer Punkt findet sich in der Initialisierung des Timers »hrt_realtime« (Listing 1, Zeile 19): Ein Timer, der der Zeitquelle »CLOCK_REALTIME« zugeordnet werden soll, muss den Startzeitpunkt der Timerfunktion als Absolut-Wert erhalten. Darauf ist bereits bei der Initialisierung über die Funktion »hrtimer_init()« zu achten. Wenn hier nicht das Flag »HRTIMER_ABS« gesetzt ist, ordnet der Kernel den Timer stillschweigend der Zeitquelle »CLOCK_MONOTONIC« zu.

Wer das kleine Experiment selbst nachvollziehen möchte, findet in Listing 3 eine kurze Applikation, die mit Superuser-Rechten ausgestattet die Systemzeit vorstellt. Der Aufruf von »make warp_time« übersetzt das Programm. Es versteht sich von selbst, dass nicht nur die Zeit hüpft, wenn man diese Software auf Produktivsystemen ausprobiert.

Das kurze Skript aus Listing 4, in jenem Verzeichnis ausgeführt, in dem sich sowohl das Kernelmodul »hrtimer.ko« als auch das Zeitsetzungsprogramm »warp_time« befinden, demonstriert das Verhalten, das in Abbildung 1 zu sehen ist. Wichtig ist es, nach Abschluss des Experiments die Uhrzeit wieder zurückzusetzen, entweder mit »warp_time« und einem negativen Parameter oder beispielsweise »ntpdate de.pool.ntp.org«.

## Moderne Technik

Die Entwickler des Hrtimer-Code kommen aus dem Umfeld des Realtime-Computing. Kein Wunder also, dass einfaches Locking, optimierte Datenstrukturen und übersichtliche Abläufe zum Einsatz kommen. Neben einem Red-Black-Tree (siehe [2]), der aktivierte Timer effizient einsortiert, wird der neue Zeitdatentyp »ktime_t« eingeführt. Er basiert nicht mehr auf Timerticks, den bekannten Jiffies, sondern auf Nanosekunden.

## Bitparade

64-Bit-Systeme handhaben den neuen Zeittyp anders als 32-Bit-Systeme, aber für den Programmierer transparent. Auf 64-Bit-Systemen, für die die Variable optimiert ist, kommt eine skalare Version zum Einsatz: Zeitwerte werden als 64-Bit-Nanosekunden abgelegt. Auf 32-Bit-Maschinen ist der 64-Bit-Wert dagegen in zwei 32-Bit-Wörter eingeteilt (Union-Variante). Das eine Wort speichert die Sekunden, das andere die Nanosekunden ab – übrigens in normalisierter Form: Der Nanosekundenanteil darf niemals größer als 1 Sekunde werden.

Die beiden 32-Bit-Wörter sind zudem auf einer Little-Endian- anders im Speicher abgelegt als auf einer Big-Endian-

**Listing 2: Makefile**

```
01 ifneq ($(KERNELRELEASE),)
02 obj-m := hrtimer.o
03
04 else
05 KDIR := /lib/modules/$(shell uname -r)/build
06 PWD     := $(shell pwd)
07
08 default:
09     $(MAKE) -C $(KDIR) M=$(PWD) modules
10 endif
```

**Listing 3: Zeitsprung »warp_time.c«**

```
01 #include <stdio.h>
02 #include <sys/time.h>
03
04 int main( int argc, char **argv )
05 {
06    struct timeval tv;
07    unsigned long ttw=10;
08
09    if( argc==2 )
10       ttw = strtoul( argv[1], NULL, 0 );
11    printf("%d second warp...\n", ttw);
12    gettimeofday( &tv, NULL );
13    tv.tv_sec += ttw;
14    if( settimeofday( &tv, NULL ) ) {
15       perror( "settimeofday" );
16    }
17    return 0;
18 }
```

**Tabelle 1: Funktionsübersicht »ktime_t«**

| Funktionsname | Beschreibung |
|---|---|
| ktime_t ktime_set( const long secs, const unsigned long nsecs ); | Setzt eine »ktime_t«-Variable auf Basis von »secs« und »nsecs«. Die Funktion gibt den »ktime_t«-Wert zurück. |
| ktime_t ktime_sub(const ktime_t lhs, const ktime_t rhs); | Subtrahiert »rhs« von »lhs« und gibt das Ergebnis zurück. |
| ktime_t timespec_to_ktime(struct timespec ts); | Konvertiert den im »struct timespec«-Format übergegebenen Wert »ts« und gibt diesen im »ktime_t«-Format zurück. |
| ktime_t timeval_to_ktime(struct timeval tv); | Konvertiert den im »struct timeval«-Format übergebenen Wert »tv« und gibt ihn im »ktime_t«-Format zurück. |
| ktime_t ktime_add(const ktime_t add1, const ktime_t add2); | Addiert »add1« und »add2« und gibt das Ergebnis im »ktime_t«-Format zurück. |
| ktime_t ktime_add_ns(const ktime_t kt, u64 nsec); | Addiert den »ktime_t« »kt« zu den im Skalarformat übergebenen Nanosekunden »nsec« und gibt das Ergebnis als »ktime_t« zurück. |
| ktime_t timespec_to_ktime(const struct timespec ts); | Konvertiert den als »struct timespec« übergebenen Wert »ts« und gibt ihn als »ktime_t« zurück. |
| ktime_t timeval_to_ktime(const struct timeval tv); | Konvertiert den als »struct timeval« übergebenen Wert »tv« und gibt diesen als »ktime_t« zurück. |
| static inline struct timespec ktime_to_timespec(const ktime_t kt); | Gibt »kt« als Wert vom Typ »struct timespec« zurück. |
| static inline struct timeval ktime_to_timeval(const ktime_t kt); | Gibt »kt« als Wert vom Typ »struct timeval« zurück. |
| u64 ktime_to_ns(const ktime_t kt) | Gibt »kt« als Wert in Nanosekunden (skalar) zurück. |
| void ktime_get_ts( struct timespec *ts); | Legt in »ts« die Zeit der Zeitquelle »CLOCK_MONOTONIC« als Wert vom Typ »struct timespec« ab. |
| ktime_t ktime_get_real_ts( struct timespec *ts ); | Legt in »ts« die Zeit der Zeitquelle »CLOCK_REALTIME« als Wert vom Typ »struct timespec« ab. |
| ktime_t ktime_get_real(void); | Gibt die Zeit der Zeitquelle »CLOCK_REALTIIME« als Wert vom Typ »ktime_t« zurück. Für diese Funktion findet sich in der Headerdatei kein Prototyp. |

Maschine. Durch diese spezielle Art der Speicherung lassen sich die 32-Bit-Wörter auch wie ein 64-Bit-Wert verarbeiten. Funktionen und Makros, die den Umgang mit dem neuen, von der Prozessorarchitektur abhängigen »ktime_t«-Objekten erleichtern, führt Tabelle 1 auf. Listing 1 zeigt neben dem Umgang mit den Hrtimer-Objekten auch den Umgang mit dem neuen Zeitdatentyp.

Hrtimer als solche stehen aber nicht nur den In-Kernel-Usern zur Verfügung, also beispielsweise Gerätetreibern. Da »nanosleep()«, »getitimer()«, »setitimer()« und Posix-Timer an den neuen Hrtimer-Code gekoppelt sind, machen Applikationen von Hrtimer hinter den Kulissen bereits ausgiebig Gebrauch.

Tabelle 2 listet weitere Funktionen, die im Umgang mit High-Resolution-Timern nützlich sind. Alle Funktionen sind nur in Kernelmodulen mit GPL-Lizenz nutzbar. Erwähnenswert ist die Funktion »hrtimer_get_res()«, die die Auflösung des Timers zurückgibt. Wer etwa die Funktion »mod_init()« um die Zeilen

```
struct timespec tp;

hrtimer_get_res( CLOCK_REALTIME, &tp );
printk("realtime(%p): %ld, %ld\n",
    &hrt_realtime, tp.tv_sec, tp.tv_nsec );
```

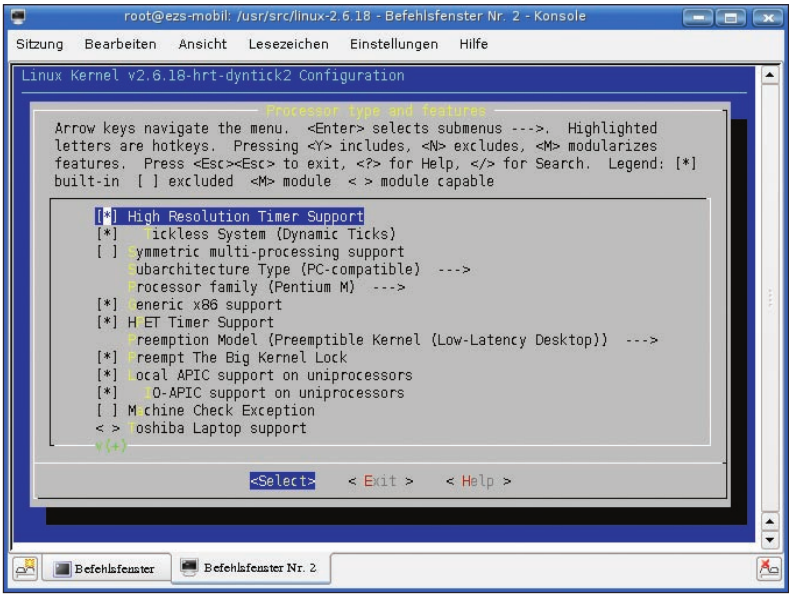


**Abbildung 2: Nur wer nach dem Patchen des Kernels den High-Resolution-Timer und das Dyntick-Patch aktiviert, kommt in den Genuss erheblich verkürzter Latenzzeiten.**

ergänzt, sieht im Syslog, dass auf einer PC-Plattform die Hrtimer mit einer Auflösung von nur 4 Millisekunden oder 250 Hz arbeiten. Dabei lässt der Name doch anderes vermuten. Mit dem Hrtimer-Code allein, wie er in den Kerneln 2.6.16 bis 2.6.19 implementiert ist, bietet der Kernel noch keine höhere Auflösung oder höhere Genauigkeit als die des Systemtakts. Wer seinen Kernel aber mit dem auf der Kernel-Mailingliste kursierenden Dyntick-Patch tunt (siehe [5]), darf sich auf ein Aha-Erlebnis einstellen: Statt mit Millisekunden hat er es jetzt mit Mikrosekunden zu tun (siehe Kasten „Dynamische Ticks“). ▶

**Tabelle 2: Hrtimer-Funktionsübersicht**

| Funktionsname | Beschreibung |
|---|---|
| extern void hrtimer_init(struct hrtimer *timer, clockid_t which_clock, enum hrtimer_mode mode); | Initialisiert den Timer »timer« und ordnet ihn der Zeitquelle »CLOCK_REALTIME« oder »CLOCK_MONOTONIC« zu. Ein »mode« von »HRTIMER_ABS« bedeutet, dass die Zeit absolut angegeben wird (für »CLOCK_REALTIME« ist diese Wahl zwingend). Falls »mode« »HRTIMER_REL« ist, wird die Zeit relativ angegeben. |
| extern int hrtimer_start(struct hrtimer *timer, ktime_t tim, const enum hrtimer_mode mode); | Aktiviert den Timer »timer« auf der aktuellen CPU, sodass im Fall von »mode=HRTIMER_REL« nach der Zeitdauer »tim« oder im Fall von »mode=HRTIMER_ABS« zum Zeitpunkt »tim« die Timerfunktion »timer->function« (im Interrupt-Kontext) startet. Gibt im Erfolgsfall 0 zurück und eine 1, falls »timer« bereits aktiviert war. |
| extern int hrtimer_cancel(struct hrtimer *timer); | Die Funktion deaktiviert »timer«. Falls zum Zeitpunkt des Aufrufs die »timer->function« abgearbeitet wird, wartet die Funktion bis zum Ende der Abarbeitung. Die Funktion gibt 0 zurück, falls »timer« inaktiv war, sonst 1 (»timer« ist abgearbeitet). |
| extern int hrtimer_try_to_cancel(struct hrtimer *timer); | Diese Funktion versucht »timer« zu stoppen. Sie gibt 0 zurück, falls »timer« inaktiv war, und 1, falls »timer->function« ausgeführt worden ist, und -1, falls gerade »timer->function« ausgeführt wird und nicht gestoppt werden kann. |
| extern ktime_t hrtimer_get_remaining(const struct hrtimer *timer); | Gibt die verbliebene Wartezeit bis zum Start der Callback-Funktion »timer->function« zurück. |
| extern int hrtimer_get_res(const clockid_t which_clock, struct timespec *tp); | Die Funktion speichert die Auflösung der Zeitquelle »which_clock« in »tp« ab. »which_clock« kann entweder den Wert »CLOCK_REALTIME« oder »CLOCK_MONOTONIC« haben. Die Funktion gibt grundsätzlich 0 zurück. |

**Listing 4: Skript »doit_hrt«**

```
01 #/bin/bash
02 echo "Loading hrtimer.ko"
03 insmod hrtimer.ko
04 sleep 2
05 echo "warping time"
06 ./warp_time 10
07 echo "done"
```

**Listing 5: »linux/ktime.h«**

```
01 ...
02 typedef union {
03         s64     tv64;
04 #if BITS_PER_LONG != 64 && !defined(CONFIG_KTIME_
   SCALAR)
05         struct {
06 # ifdef __BIG_ENDIAN
07         s32     sec, nsec;
08 # else
09         s32     nsec, sec;
10 # endif
11         } tv;
12 #endif
13 } ktime_t;
14 ...
```

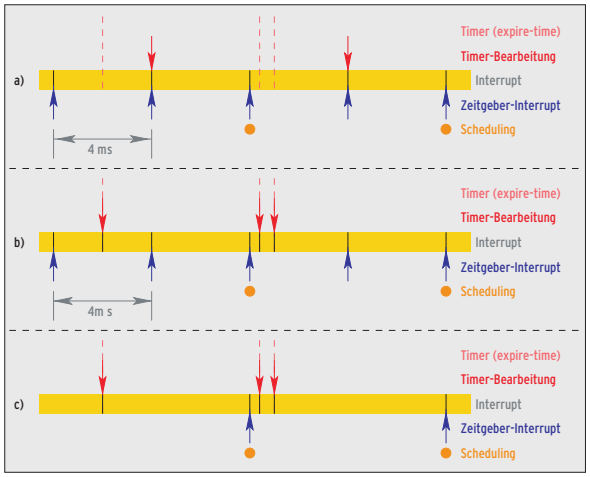


**Abbildung 3:** Die Zeitverwaltung im Linux-Kernel bis Version 2.6.19 basiert auf dem periodischen Zeitgeber-Interrupt ((a), blaue Pfeile). Kommende Versionen erzeugen zusätzlich dynamisch Interrupts, die die Timergenauigkeit erhöhen ((b), rote Pfeile). In einem nächsten Schritt sollen die dynamischen Interrupts den klassischen Zeitgeber-Interrupt ablösen (c).

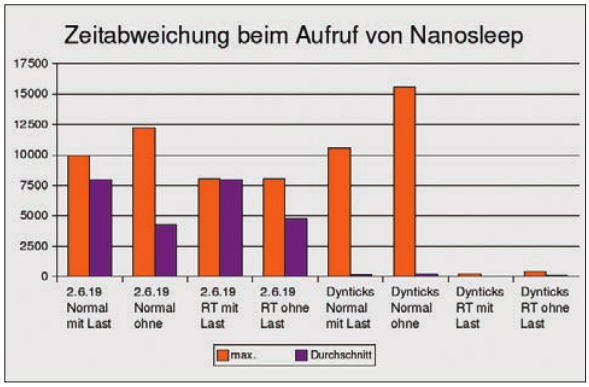


**Abbildung 4:** Ein mit den Dynticks gepatchter Hrtimer-Kernel zeigt Latenzzeiten, die im Durchschnitt deutlich unter 100 Mikrosekunden liegen (lila Balken). Die mit RT gekennzeichneten Echtzeitprozesse scheinen bei diesem Kernel grundsätzlich mit hoher Präzision zu reagieren.

Eigene Messungen bestätigen die in [3] veröffentlichten Zahlen. Abbildung 4 zeigt Abweichungen, die beim Aufruf der Funktion »nanosleep()« von der der Funktion übergebenen Soll-Schlafenszeit auftreten. Auf dem Testsystem – Pentium-M-Notebook mit 1,4 GHz – konnten die Autoren einen Fehler von durchschnittlich gut 5 Millisekunden (lila Balken) messen. Maximal waren es sogar 12,5 Millisekunden (orange).

Ein mit Hrtimer und Dynticks gepatchter Kernel spielt bei normalen Prozessen mit den Maximalzeiten in der gleichen Liga, die Durchschnittszeiten – relevant für weiche Echtzeitsysteme – sind mit unter 100 Mikrosekunden aber erheblich besser. Mehr noch: Versehen mit Realzeitpriorität ließ sich keine Zeitabweichung messen, die mehr als 500 Mikrosekunden betrug. Dabei war das Verhalten relativ unabhängig davon, ob das System unter Last stand oder nicht.

## Einer zu viel

Mit den Hrtimern hält ein zweites Zeitverwaltungssystem Einzug in den Linux-Kernel. Bislang ergänzen sich Low-Resolution- und Low-Precision- (das bisherige Timekeeping) sowie High-Resolution- und High-Precision-Zeitverwaltung. Eine gelungene Symbiose mit optimierter Aufgabenverteilung, so die Werbung der Kernelprogrammierer. Doch die Hrtimer ziehen mehr und mehr Aufgaben an sich. Ist vielleicht doch ein Zeitverwaltungssystem zu viel im Kernel?

Thomas Gleixner und Ingo Molnar merken zu dieser Frage an [4]: „Die Implementierung lässt Raum für weitere Entwicklungen, wie die eines völlig ohne periodischen Takt betriebenen Systems (bei dem der Scheduler die Zeitscheibe kontrolliert), Profiling mit variabler Frequenz und dem kompletten Entfernen der Jiffies (Kernel-interne Zeiteinheit, die auf dem periodischen Tick basiert) in der Zukunft." *(ofr)* ■

### Dynamische Ticks

Möglicherweise wird die Kernelversion 2.6.20 bereits die auf der Kernel-Mailingliste kursierenden Dyntick-Patches enthalten (siehe [5] und Abbildung 2), die zu spürbar kürzeren Latenzzeiten führen.

Technisch basiert dieses Wunder darauf, nicht mehr nur zu den periodisch kommenden Zeitgeber-Interrupts (Ticks) den Ablauf der Timer zu überprüfen, sondern im Vorhinein den nächsten Ablaufpunkt auszurechnen und einen Hardware-Timer dafür aufzuziehen. Abbildung 3 erläutert das Timekeeping der verschiedenen Linux-Kernel.

a) Die Zeitverwaltung bis Kernel 2.6.16 basiert auf einem periodischen Interrupt. Beim Auftreten dieses Zeitgeber-Interrupt ruft der Kernel zuvor abgelaufene Timerfunktionen auf und stößt – falls dies notwendig ist – das Scheduling an.

b) Das auf periodischen Timerticks basierende Zeitverwaltungssystem koexistiert mit den neuen Hrtimern, die mit Unterstützung der dynamische Ticks hochpräzise arbeiten. Die Dynticks sorgen dafür, dass immer dann Interrupts entstehen, wenn ein Timer abläuft. Sie sollen voraussichtlich ab Version 2.6.20 im Linux-Kernel integriert sein.

c) Für spätere Linux-Versionen ist es vorgesehen, den periodischen Zeitgeber-Interrupt ebenfalls durch die Dynticks zu ersetzen. Überflüssige Zeitgeber-Interrupts gehören dann der Vergangenheit an.

### Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 27: Linux-Magazin 04/06, S. 118

[2] Tim Schürmann, „Bunte Bäume": Linux-Magazin 12/06, S.70

[3] Gleixner, Niehaus, „Hrtimers and beyond: Transforming the Linux Time Subsystem. Proceedings of the Linux Symposium": Volume One, 2006, S. 333: [http://www.linuxsymposium.org/2006/linuxsymposium_procv1.pdf]

[4] Thomas Gleixner und Ingo Molnar, „High resolution timer design notes": [http://rt.wiki.kernel.org/index.php/High_resolution_timer_design_notes]

[5] Dyntick-Patch: [http://tglx.de/projects/hrtimers/2.6.18/patch-2.6.18-hrt-dyntick2.patch]

### Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Ihr Buch „Linux-Treiber entwickeln" ist seit Juni in zweiter Auflage im Handel.