Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 27

Kern-Technik

Trotz Taktveränderungen durch Speedstep oder Quiet'n'Cool im Kernel exakte Zeiten messen? Diese Folge der Kern-Technik stellt Kernelfunktionen für den richtigen Umgang mit der Zeit vor. Eva-Katharina Kunst, Jürgen Quade

Wie bei gesunden Menschen schlägt auch im Inneren jedes Betriebssystems das Herz regelmäßig den Takt. Dazu löst eine Timer-Hardware periodisch einen Interrupt aus, der unter anderem den Scheduler aktiviert. Fällt dieser Interrupt aus, bleibt das Betriebssystem schließlich ganz stehen. Ein hoher Systemtakt hat Vorteile: Je höher er ausfällt, desto responsiver ist das System.

Andererseits bedeutet ein höherer Systemtakt auch mehr Overhead – was auf schnellen Prozessoren allerdings prozentual gesehen vernachlässigbar ist. Im Kernel 2.4 pochte das Herz noch mit gemächlichen 100 Hz; bei Kernel 2.6 erhöhte Torvalds den Takt auf 1000 pro Sekunde. Je eine Unterbrechung pro Millisekunde führte auf einigen Systemen jedoch zu Problemen. Jetzt läuft der Kernel daher standardmäßig mit 250 Hz, also 4 Millisekunden Abstand.

Linux zählt seinen Takt in der globalen Variablen »jiffies«, die regelmäßige Leser der Kern-Technik-Serie schon kennen gelernt haben. Ein Lesezugriff gibt bereits einen ersten Anhaltspunkt über die Zeitrechnung des Kernels. Doch ist Vorsicht bei der Auswertung geboten: Beim Vergleichen von Zeitwerten kommt es leicht zu Fehlern. Da die Variable Jiffies vom Typ Unsigned Long ist, können auf einem 32-Bit-System maximal 4 Millionen Takte gezählt werden.

Bei einem Systemtakt von 1000 Hz kommt es also etwa alle 49 Tage zu einem Überlauf. In den Vergleich von Zeitstempeln vor und nach dem Überlauf mit den bekannten Operatoren > und < schleichen sich daher sehr leicht Fehler ein.

Makros für Zeitstempel

Falls die Zeitstempel nur ein paar Tage auseinander liegen, funktionieren die in Tabelle 1 aufgeführten Makros »time_after()«, »time_after_eq()«, »time_before()« und »time_before_eq()«. Der Trick: Statt zweier Zeitstempel vom Typ Unsigned Long vergleichen die Makros vorzeichenbehaftete Werte vom Typ Long. Abbildung 1 zeigt, dass damit auch bei einem Zählerüberlauf zwischen den Zeitstempeln das richtige Ergebnis herauskommt. Torvalds zwingt übrigens die Kernelprogrammierer zu sauberer Arbeit: Die Variable Jiffies ist bei Systemstart so vorbelegt, dass sie bereits nach 5 Minuten überläuft, siehe »INITIAL_JIFFIES« in »include/linux/jiffies.h«.

Natürlich lassen sich damit auch Zeitstempel miteinander vergleichen, die weiter auseinander liegen, allerdings nur über die Variable »jiffies64«. Auf einem 32-Bit-System ist der Zugriff auf eine 64-Bit-Variable ein kritischer Abschnitt, den ein Sequence-Lock schützt. Aus diesem Grund sollte der Zugriff nur über die Funktion »get_jiffies_64()« erfolgen. Auf 64-Bit-Hardware bilden Präprozessor und Compiler die Funktion auf den direkten und damit schnelleren Variablenzugriff ab.

Die Genauigkeit, mit der sich Zeiten mit Jiffies bestimmen lassen, liegt je nach gewähltem Systemtakt zwischen 1 und 10 Millisekunden. Zu ungenau für die gestellte Aufgabe? Kein Problem für Linux, denn es aktualisiert in der Kernelvariablen »xtime« die Zeit bei jedem Timertick mit bis zu Nanosekunden-Genauigkeit. Xtime zählt die Sekunden und Nanosekunden, die seit dem 1.1.1970 verstrichen sind. Da die Speicherung der Zeitinformation mehr als 32 Bit erfordert, führt der Zugriff aber zu einem kritischen Abschnitt.

Am besten eignet sich für das Auslesen des Zeitwerts die Funktion »current_kernel_time()«, sie gibt die aktuelle Zeit in einer »struct timespec« zurück:

```
struct timespec {
   time_t tv_sec;
   long tv_nsec;
}
```

Wer stattdessen lieber mit einer »struct timeval« arbeitet, ruft die Funktion »do_gettimeofday()« auf:

```
struct timeval {
   time_t tv_sec;
   suseconds_t tv_usec;
}
```

Die Verwendung der Funktionen demonstriert Listing 1.

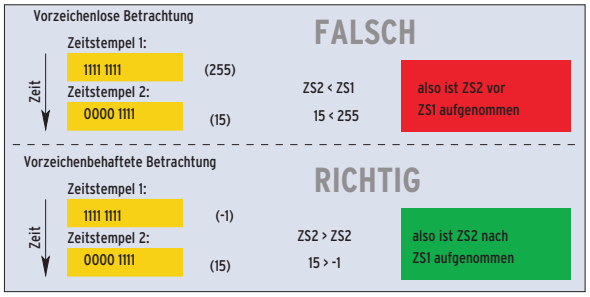

Abbildung 1: Zeitstempel vorzeichenbehaftet zu betrachten ermöglicht den korrekten Zeitvergleich (hier vereinfacht auf 8-Bit-Zeitstempel).

Obwohl »xtime« nur mit jedem Timertick neu gesetzt wird, besitzt die Variable hohe Genauigkeit. Anstatt sie bei 250 Hz Systemtakt blind jedes Mal um 4 Millisekunden zu erhöhen, misst der Kernel die zwischen den beiden Timerticks tatsächlich vergangene Zeit hochgenau. Um diesen Wert erhöht er dann »xtime«. Für die hochgenaue Messung greift der Kernel auf Hardware zurück. Moderne Prozessoren besitzen oft einen Taktzyklenzähler, den Time Stamp Counter (TSC). Mit jedem Taktsignal inkrementiert die CPU diesen Zähler. Bei 1 GHz Systemtakt erreicht der TSC eine Auflösung von Nanosekunden.

Auf diesen Zähler kann der Programmierer natürlich direkt zugreifen, sogar aus dem Userspace (siehe Kasten „Zeit im Userspace"). Beispielcode dafür – lauffähig auf einer x86-Plattform – zeigt Listing 2. Während im Userspace Assembler-Code nötig ist, hält der Kernel die Makros »rdtsc()«, »rdtscl()«, »rdtscll()« und »get_cycles()« bereit.

Da es bei der hohen Zählfrequenz schnell zu Überläufen kommt, ist der TSC auf den meisten Plattformen als 64-Bit-Zähler implementiert. Das Makro »rdtsc« übergibt in zwei Variablen einmal das höher- und einmal das niederwertige 32-Bit-Wort; »rdtscll« belegt die übergebene 64-Bit-Variable direkt mit dem Zählerwert, »rdtscl« organisiert nur das niederwertige 32-Bit-Wort.

Das Makro »get_cycles()« ist übrigens die plattformneutrale Variante, die je nach TSC einen Wert von 32- oder 64-Bit Breite zurückgibt. In Listing 3 ist der Zugriff auf den TSC vom Kernel aus zu erkennen. Doch auch mit dem TSC gibt es Probleme:

- Nicht alle Prozessoren bieten einen Taktzyklenzähler.
- Um aus dem erfassten Zählwert einen Zeitwert zu berechnen, ist die Kenntnis der Taktfrequenz notwendig.
- Auf modernen Prozessoren ist die Taktfrequenz nicht konstant.
- Ein 32-Bit-TSC läuft bei Gigahertz-Takt in wenigen Sekunden über.
- Die Verarbeitung eines 64 Bit breiten Zeitwerts ist auf einem 32-Bit-System nicht in einem Zugriff möglich.

Insbesondere die je nach Last wechselnde Taktfrequenz ist ein großes Problem. Sei es Power Now, Quiet'n'Cool oder Speedstep: Wenn sich während der Zeitmessung die Taktfrequenz ändert, ist eine Differenz-Zeitmessung nicht mit einer einfachen Differenz-Bildung getan.

Time-Shift

Doch auch in dieser Situation bietet Linux einen Ausweg. Schließlich ist es der Kernel selbst, der die CPU anweist einen Gang runter oder auch rauf zu schalten. So kann der Programmierer über den Notifier-Mechanismus vor und nach jeder Taktfrequenzänderung eine eigene Funktion aufrufen lassen. Sie bekommt die zuletzt eingestellte und die neue Taktfrequenz mitgeteilt, sodass die Berechnung der verstrichenen Zeit nicht mehr übermäßig schwer fällt. In Listing

Tabelle 1: Kernelfunktionen zur Zeitmessung

Typ	Kurzbeschreibung
int time_after(unsigned long a, unsigned long b)	Zeitvergleich
int time_after_eq(unsigned long a, unsigned long b)	Zeitvergleich
int time_before(unsigned long a, unsigned long b)	Zeitvergleich
int time_before_eq(unsigned long a, unsigned long b)	Zeitvergleich
u64 get_jiffies_64(void)	Anzahl Timerinterrupts seit Systemstart als 64-Bit-Zähler
struct timespec current_kernel_time(void)	Sekunden und Nanosekunden seit 1. 1. 1970
void do_gettimeofday(struct timeval *tv)	Sekunden und Mikrosekunden seit 1. 1. 1970
rdtsc(unsigned long low, unsigned long high)	Time Stamp Counter
rdtscl(unsigned long low)	Time Stamp Counter (niederwertiges Wort)
rdtscll(unsigned long long val)	Time Stamp Counter (64-Bit-Wort)
cycles_t get_cycles(void)	Time Stamp Counter (Hardware-unabhängig)
void ndelay(unsigned long nanoseconds)	Verzögerung in Nanosekunden
void udelay(unsigned long useconds)	Verzögerung in Mikrosekunden (maximal 20000)
void mdelay(unsigned long mseconds)	Verzögerung in Millisekunden
long schedule_timeout_interruptible(long timeout)	Unterbrechbar schlafen legen
long schedule_timeout_uninterruptible(long timeout)	Ununterbrechbar schlafen legen
void add_timer(struct timer_list *)	Timer-Tasklet zur späteren Ausführung
int queue_delayed_work(struct workqueue_struct *wq, struct work_struct *work, unsigned long delay)	Funktion in einer Workqueue abarbeiten lassen
schedule_delayed_work(struct work_struct *work, unsigned long delay)	Funktion in der Event-Workqueue abarbeiten lassen

Listing 1: Genaue Zeit im Kernel

```
01 struct timespec spec_since_1970;
02 struct timeval val_since_1970;
03 ...
04 spec_since_1970 = current_kernel_time();
05 do_gettimeofday(&val_since_1970);
```

Listing 2: Time Stamp Counter vom Userspace

```
01 static inline unsigned long long int rdtsc()
02 {
03     unsigned long long int x;
04     __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
05     return x;
06 }
```

Listing 3: Time Stamp Counter im Kernel

```
01 unsigned long low, high;
02 cycles_t tscall;
03 ...
04 if( cpu_has_tsc )
05    rdtsc( low, high );
06 tscall=get_cycles();
```

4 ist die Realisierung eines solchen Taktfrequenz-Notifiers zu sehen.

Als Erstes definiert man ein Objekt, das die Adresse der Notifer-Callback-Funktion aufnimmt (Zeile 31). Der Aufruf von »cpufreq_register_notifier()« macht die Funktion der Zeitverwaltung des Kernels bekannt. Dieser fügt abhängig von der gewählten Parametrisierung die Callback-Funktion in eine von zwei Listen ein: Lautet der zweite Aufrufparameter »CPUFREQ_POLICY_NOTIFIER«, dann ruft der Linux-Kernel die Callback-Funktion jedes Mal auf, wenn sich die Policy der Frequenzumschaltung ändert. In Listing 4 kommt stattdessen »CPUFREQ_TRANSITION_NOTIFIER« zum Einsatz. Damit ruft der Kernel die Notifier-Funktion vor und nach der Taktfrequenzänderung auf.

Zu erwähnen bleibt noch, dass beim Entladen des Moduls das Objekt »nb« wieder aus dem Kernel entfernt werden muss (»cpufreq_unregister_notifier()«). Wer ein System mit variabler Taktfrequenz besitzt, kann den Quellcode aus Listing 4 mit dem Makefile (Listing 5) übersetzen und das erzeugte Modul per »insmod cpunotifier.ko« laden. Bei sehr unterschiedlichen Lastsituationen verändert der Kernel wie beschrieben den Takt. Das Notifier-Modul dokumentiert seine Aktivitäten im Syslog, wie Abbildung 2 zeigt.

Der Linux-Kernel bietet natürlich mehr als nur die Möglichkeit, genaue Zeiten zu erfassen und zu vergleichen. Häufig ist es notwendig, eine gewisse Zeit aktiv oder passiv zu warten. Aktives Warten ist an sich unerwünscht, schließlich kann der Prozessor nützlichere Dinge tun, als Warteschleifen drehen.

Warten in Schleifen

Wenn die Wartezeit kurz ist und sich das Umschalten (Scheduling) auf einen anderen Rechenprozess nicht lohnt oder wenn das Warten im Interruptkontext stattfindet, gibt es kaum Alternativen zur Warteschleife. Der Kernel bietet dafür die Funktionen »ndelay()«, »udelay()« und »mdelay()« an. Die erste Funktion bekommt die Wartezeit in Nano-, die zweite in Mikro- und die dritte in Millisekunden übergeben. Übrigens gibt sich der Compiler redlich Mühe festzustellen, ob »udelay()« mit mehr als 20000 parametrisiert wird. In diesem Fall erscheint beim Kompilieren die Meldung:

```
*** Warning: "__bad_udelay" ➥
/tmp/treiber/udelaytest.ko] undefined!
```

Da das Symbol »__bad_udelay« an keiner Stelle im Kernel definiert ist, scheitert das Laden des Moduls.

Noch ein zweiter Punkt: Da »udelay()« (wie auch »ndelay« und »mdelay«) mit einer Zählschleife arbeitet, lässt sich damit prinzipiell sehr genau warten. Unterbrechen aber wichtigere Aktionen die gerade aktive Codesequenz, verlängert sich die Wartezeit um deren Laufzeit.

Zeit im Userspace

Wenn es um Zeit und Zeitverwaltung auf Applikationsebene geht, bietet Linux eine Reihe von Funktionen. Das Auslesen der aktuellen Zeit läuft dabei meist über den Systemcall »gettimeofday()«. Alternativ kann der Programmierer – falls seine CPU das unterstützt – auch auf den Time Stamp Counter zugreifen. Dazu bindet er auf der x86-Architektur einfach die in Listing 2 dargestellte (Assembler-)Funktion in sein Programm ein. Allerdings muss er sicherstellen, dass sich während der Messung die Taktfrequenz nicht ändert.

Das Schlafenlegen eines Rechenprozesses erfolgt typischerweise über den Systemcall »sleep()«. Er bietet aber nur eine Auflösung im Sekundenbereich. Lange Zeit haben sich Programmierer daher mit »select()« beholfen, also mit einer Funktion, der man ein Timeout übergeben kann. Mittlerweile lassen sich Sub-Sekunden-Wartezeiten über »nanosleep()« realisieren. Diese Funktion wird innerhalb des Kernels direkt auf »schedule_timeout_interruptible()« abgebildet.

Um eine Funktion zu einem späteren, definierten Zeitpunkt abarbeiten zu lassen, gibt es zwei Interfaces: Itimer und Posix-Timer. Die über »setitimer()« und »getitimer()« zur Verfügung gestellten Rückwärtszähler schicken dem zugehörigen Prozess ein Signal, wenn der Wert null ist.

Intern ähnlich wie der Itimer implementiert, bieten die Posix-Timer dem Programmierer mit Funktionen wie »timer_create()« und »clock_settime()«) ein flexibles Interface. Die Benachrichtigung, dass ein Zähler abgelaufen ist, erfolgt wie bei den Itimern per Signal.

Listing 4: »cpunotifier.c«

```
01 #include <linux/fs.h>
02 #include <linux/notifier.h>
03 #include <linux/cpufreq.h>
04
05 static int my_cpufreq_notifier( struct
   notifier_block *nb, unsigned long val, void
   *data )
06 {
07    struct cpufreq_freqs *freq = data;
08
09    switch( val ) {
10    case CPUFREQ_PRECHANGE:
11       printk("CPUFREQ_PRECHANGE ");
12       break;
13    case CPUFREQ_POSTCHANGE:
14       printk("CPUFREQ_POSTCHANGE ");
15       break;
16    case CPUFREQ_RESUMECHANGE:
17       printk("CPUFREQ_RESUMECHANGE ");
18       break;
19    case CPUFREQ_SUSPENDCHANGE:
20       printk("CPUFREQ_SUSPENDCHANGE ");
21       break;
22    default:
23       printk("unknown val\n");
24       break;
25    }
26    printk("cpu: %d, old: %d, new: %d\n",
   freq->cpu, freq->old, freq->new );
27    return 0;
28 }
29
30 static struct notifier_block nb={
31    .notifier_call = my_cpufreq_notifier,
32 };
33
34 static int __init mod_init(void)
35 {
36    if( cpufreq_register_notifier(&nb,CPUFREQ_
   TRANSITION_NOTIFIER) ) {
37       printk("cpufreq_register_notifer
   failed\n");
38       return -EIO;
39    }
40    printk("mod_init: cpufreq_get: %d\n",
   cpufreq_get(0));// CPU 0
41    return 0;
42 }
43
44 static void __exit mod_exit(void)
45 {
46    cpufreq_unregister_notifier( &nb, CPUFREQ_
   TRANSITION_NOTIFIER );
47 }
48
49 module_init( mod_init );
50 module_exit( mod_exit );
51 MODULE_LICENSE("GPL");
```

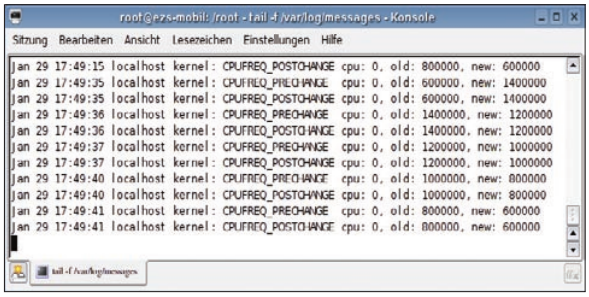

Abbildung 2: Im Syslog zu sehen: Bei jeder Änderung der Prozessortaktfrequenz wird die Notifier-Funktion vom Kernel zweimal aufgerufen, vorher und nachher.

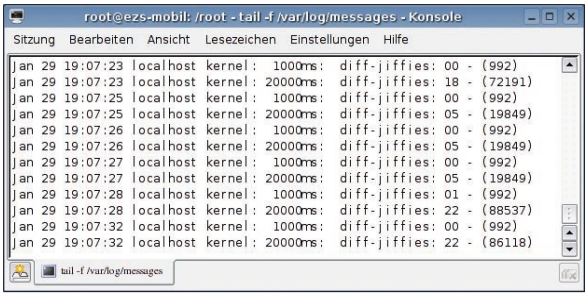

Abbildung 3: Nur bei kurzen Wartezeiten (1000 ms) erreicht »udelay()« eine hohe Genauigkeit. Bei hohen (20 000 ms) wird die Abweichung bereits recht groß.

Hochgenau lässt sich damit also nur in einer Interrupt Service Routine (ISR) warten, wenn andere Aktivitäten unterbunden sind.

Listing 6 verschafft ein Gefühl für die Tücken beim Verwenden von »udelay()«. Beim Laden des Moduls »udelaytest.ko« ruft das Beispiel »udelay()« einmal mit »1000« und einmal mit »20000« auf. Die real vergehende Zeit misst das Programm mit Hilfe des Time Stamp Counter. Das Ergebnis der Messung taucht wieder im Syslog auf (Abbildung 3).

Da die Wahrscheinlichkeit gering ist, bei einer kurzen Wartezeit unterbrochen zu werden, ist die tatsächliche Wartezeit bei kleinen Zeiten (1000 Mikrosekunden) sehr genau. Bei Wartezeiten von mehr als 4 Millisekunden kommt es aber durch den erwähnten Timertick-Interrupt zu ersten Verzögerungen, die sich insgesamt summieren. Im Beispiel wartet die Instanz statt der veranschlagten 20 Milliskunden bis zu 88 Millisekunden (22 Jiffies, Abbildung 3).

Das ohnehin empfehlenswertere passive Warten eliminiert diese zusätzlichen Verzögerungen. Dazu dienen die Funktionen »schedule_timeout_interruptible()« und »schedule_timeout_uninterruptible()«, denen der Programmierer die Wartezeit in Jiffies übergibt. Alternativ rechnen »msecs_to_jiffies()« und »usecs_to_jiffies()« Milli- oder Mikrosekunden-Angaben in Jiffies um. Da die zugehörige Instanz schlafen soll, stehen diese Funktionen nicht im Interrupt-Kontext zur Verfügung. Man kann sie also nicht in einer ISR oder einem Soft-IRQ (Tasklet, Timer) verwenden. Der Aufruf einer der beiden Funktionen garantiert, dass mindestens die angegebene Zeit geschlafen wird.

Timer oder Workqueue

Um nur eine Zeit abzuwarten und dann eine in sich geschlossene Aufgabe zu bearbeiten, ist es besser, mit »add_timer()« direkt auf Timer oder auf die bestehende Event-Workqueue (»queue_delayed_work()«, »schedule_delayed_work()«) zurückzugreifen. Timer-Funktionen laufen im Interrupt-, Workqueues im Prozess-Kontext ab. Timer starten relativ genau zum angegebenen Zeitpunkt; da sich hinter den Workqueues Kernelthreads verstecken, bestimmt der Scheduler deren genauen Abarbeitungszeitpunkt. Zum Aufsetzen eines Timer und die Übergabe einer Funktion an eine Workqueue siehe [1] oder [2].

Noch ein Hinweis: Die Kernelentwickler Thomas Gleixner und Ingo Molnar haben Linus Torvalds davon überzeugt, eine überarbeitete Version der Zeitverwaltung in Kernel 2.6.16 zu übernehmen. Damit halten auch neue High-Resolution Timer in den Kernel Einzug. Alles Wissenswerte dazu verrät die nächste Folge der Kern-Technik. *(ofr)* ■

Infos:

[1] Kunst, Quade, „Kern-Technik", Folge 4: Linux-Magazin 11/03, S. 96

[2] Quade, Kunst, „Linux-Treiber entwickeln": Dpunkt-Verlag, 1. Auflage, Juni 2004

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.

Listing 5: Makefile

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

Listing 6: »udelaytest.c«

```
01 #include <linux/fs.h>
02 #include <linux/delay.h>
03 #include <linux/cpufreq.h>
04
05 static int __init delaytest_init(void)
06 {
07    long old_jiffies;
08    cycles_t t1, t2;
09    int freq=cpufreq_get(0)/1000; // in msec
10
11    t1=get_cycles();
12    old_jiffies = jiffies;
13    udelay(1000); // Kurzes Warten
14    t2=get_cycles();
15    printk(" 1000ms:  diff-jiffies: %2.2ld -
   (%ld)\n",
16           jiffies-old_jiffies, (unsigned
   long)(t2-t1)/freq);
17    old_jiffies = jiffies;
18    t1 = t2;
19    udelay(20000); // Langes Warten
20    t2=get_cycles();
21    printk("20000ms:  diff-jiffies: %2.2ld -
   (%ld)\n",
22           jiffies-old_jiffies, (unsigned
   long)(t2-t1)/freq);
23    return -EIO; // Nur ein Test: Das erspart
   uns das Entladen.
24 }
25
26 module_init( delaytest_init );
27 MODULE_LICENSE("GPL");
```