Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 113

As Time Goes By

Unterschiedliche Variablen, Datentypen, Einheiten, Bezugspunkte und fiese Zeithüpfer erschweren das Leben eines Kernel- oder Treiberprogrammierers, der Zeiten messen, vergleichen, berechnen oder auch nur verschlafen möchte. Eva-Katharina Kunst, Jürgen Quade

Die Autoren

Eva-Katharina Kunst ist schon seit den Anfängen von Linux Fan von Open Source. **Jürgen Quade**, Professor an der Hochschule Niederrhein, bietet auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux an.

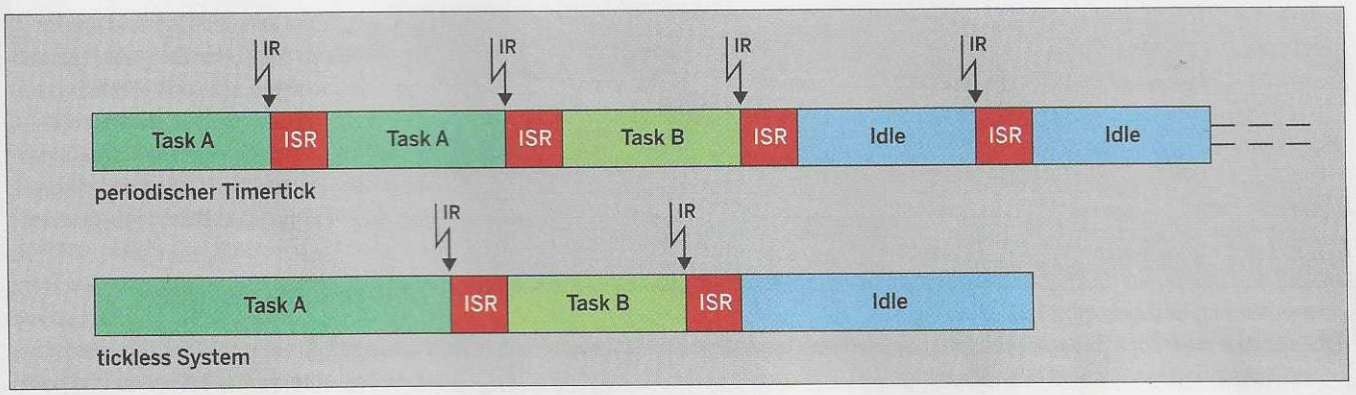
Die Zeitverwaltung im Linux-Kernel unterliegt einer ständigen Evolution. Während früher Timer-Ticks gezählt und in der Variablen `jiffies` abgelegt wurden, verarbeitet Linux heute über die Datenstruktur `ktime` vorwiegend Zeitspannen in Nanosekunden.

Die auf einem periodischen Taktzähler basierende Zeitverwaltung wurde also schon vor einigen Jahren durch ein modernes, bedarfsorientiertes Interrupt-System ersetzt. Bei nur geringfügig höherem Rechenaufwand arbeitet der Kernel damit heute Zeitaufträge erheblich genauer ab als früher, in den meisten Anwendungsfällen bei gleichzeitig signifikanten Energieeinsparungen.

Tickt nicht richtig

Damit tickt Linux zwar auch heute noch, aber eben nicht mehr periodisch. Das Konstrukt nennt sich Tickless-System 1. Mit jedem Interrupt respektive bei jedem Zeitauftrag kalkuliert Linux den Zeitpunkt, zu dem die nächste Unterbrechung sinnvoll ist, und programmiert den Timer-Baustein entsprechend neu. Dadurch schläft das System erheblich länger an einem Stück 2 und spart den Strom für unnötige Unterbrechungen – eine Grundvoraussetzung für den Einsatz in mobilen Geräten.

Aus Gründen der Kompatibilität gibt es `jiffies` auch heute noch im Linux-



1 Moderne Linux-Systeme arbeiten tickless und arbeiten auf diese Weise Zeitaufträge erheblich genauer ab, als das früher der Fall war.

Kernel, die allerdings nicht mehr gezählt, sondern berechnet werden. Dazu greift Linux auf den präzisen Timestamp-Counter (TSC) moderner CPUs zurück. Er zählt mit der Taktfrequenz des Prozessors, die heute im Gigahertzbereich liegt. Die Zeit liegt in Form der seit dem Booten vergangenen Nanosekunden vor.

Linux bietet, wie in der Tabelle Funktionen zum Umgang mit Zeit (Auswahl) aufgeführt, eine Reihe von Konvertierungsfunktionen an, um zwischen unterschiedlichen Datentypen zu konvertieren. Diese Vielfalt entstammt übrigens sowohl der Historie als auch der Funktionalität.

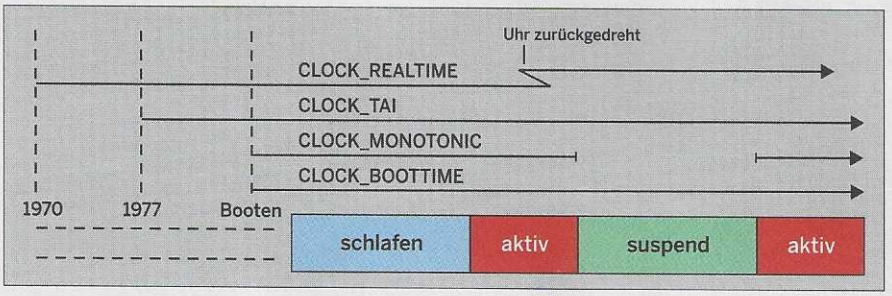
Beim Jiffies-Zähler beispielsweise handelt es sich um eine vorzeichenlose Ganzzahl – es gibt also keine negativen Werte und folglich auch keine Zeit vor dem Booten. Ktime dagegen speichert Nanosekunden als vorzeichenbehafteten 64-Bit-Wert. Hiermit kann man rund 290 Jahre in die Vergangenheit und in die Zukunft schauen. Wem das nicht genügt, der greift auf die moderne `struct timespec64` zurück. Mit 292 Milliarden Jahren nach vorn und hinten lassen sich sogar Ereignisse adressieren, die vor der Entstehung des Universums liegen 3.

Allerdings gestaltet sich das Rechnen mit Zeiten alles andere als trivial, insbesondere bei der Darstellung als `struct timespec64`, und ist damit fehleranfällig. Das Arbeiten mit Restzeiten und dem Überlauf von Variablen ist einfach nicht jedermanns Sache.

Zeiterfassung

Vor der Zeitrechnung und dem Zeitvergleich steht aber die Zeiterfassung. Die

Funktionen zum Umgang mit Zeit (Auswahl)
Initialisierung
`ktime_t ktime_set(const s64 secs, const unsigned long nsecs)`
Rechenfunktionen
`ktime_sub(lhs, rhs)`
`ktime_add(lhs, rhs)`
`ktime_add_unsafe(lhs, rhs)`
`ktime_add_ns(kt, nsval)`
`ktime_sub_ns(kt, nsval)`
`s64 ktime_divns(const ktime_t kt, s64 div)`
`s64 ktime_us_delta(const ktime_t later, const ktime_t earlier)`
`s64 ktime_ms_delta(const ktime_t later, const ktime_t earlier)`
`ktime_t ktime_add_us(const ktime_t kt, const u64 usec)`
`ktime_t ktime_add_ms(const ktime_t kt, const u64 msec)`
`ktime_t ktime_sub_us(const ktime_t kt, const u64 usec)`
`ktime_t ktime_sub_ms(const ktime_t kt, const u64 msec)`
`ktime_t ktime_add_safe(const ktime_t lhs, const ktime_t rhs);`
Konvertierung
`ktime_t timespec_to_ktime(struct timespec ts)`
`ktime_t timespec64_to_ktime(struct timespec64 ts)`
`ktime_t timeval_to_ktime(struct timeval tv)`
`ktime_to_timespec(kt)`
`ktime_to_timespec64(kt)`
`ktime_to_timeval(kt)`
`s64 ktime_to_ns(const ktime_t kt)`
`s64 ktime_to_us(const ktime_t kt)`
`s64 ktime_to_ms(const ktime_t kt)`
`bool ktime_to_timespec_cond(const ktime_t kt, struct timespec *ts)`
`bool ktime_to_timespec64_cond(const ktime_t kt, struct timespec64 *ts)`
`ktime_t ns_to_ktime(u64 ns)`
Vergleiche
`int ktime_compare(const ktime_t cmp1, const ktime_t cmp2)`
`bool ktime_after(const ktime_t cmp1, const ktime_t cmp2)`
`bool ktime_before(const ktime_t cmp1, const ktime_t cmp2)`



2 Zeitgeber legen den Bezugspunkt und das Verhalten eines Zeitzählers fest.

ist komplizierter, als man spontan denken mag, weil sie von vielen Aspekten abhängt 4, unter anderem auch vom Ort. Es macht ja einen Unterschied von fünf oder sechs Stunden aus, ob man in Berlin oder in New York nach der Uhrzeit fragt. Das Oder ergibt sich durch diverse Umstellungen von Sommer- auf Normalzeit.

Damit ergibt sich ein weiteres, großes Problem: Wer in der Nacht der Umstellung von Normal- auf Sommerzeit versucht, genau um 2:15 Uhr auf die Uhr zu sehen, der scheitert: Da die Zeit bei der Umstellung einen Sprung macht, gibt es manche Uhrzeiten einfach nicht. Das volle Ausmaß solcher Probleme führt die Lektüre des Wikipedia-Eintrags „Dynamische Zeit" vor Augen.

Die technische Lösung für derartige Komplikationen liegt in der Definition von passenden Zeitgebern. Linux unterstützt gleich eine Reihe davon (Tabelle Diverse Zeitgeber).

An der Uhr gedreht

Der Zeitgeber `CLOCK_MONOTONIC` zählt ab dem Booten immer aufwärts. Dabei ist es ihm egal, ob es einen Wechsel von Sommer- auf Normalzeit oder umgekehrt gibt. `CLOCK_REALTIME` dagegen liefert seinen aktualisierten Zählerstand ab der Unix-Epoche (1.1.1970) aus. Um ortsunabhängig zu sein, handelt es sich grundsätzlich um eine UTC-Zeit, also die Weltzeit. Hüpft die Zeit, weil jemand an der Uhr dreht, so hüpft der Zeitgeber gleich mit. Die Varianten mit `COARSE` am Ende sind zu Lasten der Genauigkeit auf Geschwindigkeit getrimmt. Sie liefern einfach den letzten Zählerstand aus, ohne ihn noch einmal explizit zu aktualisieren.

Diverse Zeitgeber

`CLOCK_MONOTONIC`
`CLOCK_REALTIME`
`CLOCK_MONOTONIC_RAW`
`CLOCK_MONOTONIC_COARSE`
`CLOCK_REALTIMTE_COARSE`
`CLOCK_BOOTTIME`
`CLOCK_TAI`

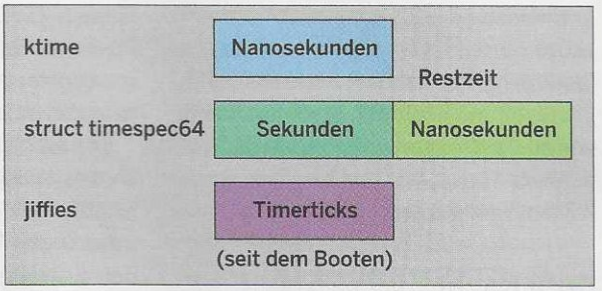
Kernel-Funktionen zur Zeiterfassung

`u64 ktime_get_ns(void)`
`u64 ktime_get_boottime_ns(void)`
`u64 ktime_get_real_ns(void)`
`u64 ktime_get_clocktai_ns(void)`
`u64 ktime_get_raw_ns(void)`
`void ktime_get_ts64(struct timespec64 *)`
`void ktime_get_boottime_ts64(struct timespec64 *)`
`void ktime_get_real_ts64(struct timespec64 *)`
`void ktime_get_clocktai_ts64(struct timespec64 *)`
`void ktime_get_raw_ts64(struct timespec64 *)`
`time64_t ktime_get_seconds(void)`
`time64_t ktime_get_boottime_seconds(void)`
`time64_t ktime_get_real_seconds(void)`
`time64_t ktime_get_clocktai_seconds(void)`
`time64_t ktime_get_raw_seconds(void)`
`ktime_t ktime_get_coarse(void)`
`ktime_t ktime_get_coarse_boottime(void)`
`ktime_t ktime_get_coarse_real(void)`
`ktime_t ktime_get_coarse_clocktai(void)`
`u64 ktime_get_coarse_ns(void)`
`u64 ktime_get_coarse_boottime_ns(void)`
`u64 ktime_get_coarse_real_ns(void)`
`u64 ktime_get_coarse_clocktai_ns(void)`
`void ktime_get_coarse_ts64(struct timespec64 *)`
`void ktime_get_coarse_boottime_ts64(struct timespec64 *)`
`void ktime_get_coarse_real_ts64(struct timespec64 *)`
`void ktime_get_coarse_clocktai_ts64(struct timespec64 *)`
`u64 ktime_get_mono_fast_ns(void)`
`u64 ktime_get_raw_fast_ns(void)`
`u64 ktime_get_boot_fast_ns(void)`
`u64 ktime_get_real_fast_ns(void)`

Zeitgeschwindigkeit

Bisher haben wir noch nicht thematisiert, dass Zeit unterschiedlich schnell vergeht. Hier greifen weniger die Arbeit Albert Einsteins und seine Relativitätstheorie, sondern schlicht schnöde Unterschiede in der Hardware. Technisch ist das Pro-

blem dadurch gelöst, dass sich die Frequenz (Zeitgeschwindigkeit) anpassen lässt – eine der Aufgaben des Network Time Protocols (NTP). Der Zeitgeber `CLOCK_MONOTONIC_RAW` liefert eine stets aufwärts zählende Hardware-Zeit, die im Unterschied zu `CLOCK_MONTONIC` unbeeinflusst von irgendwelchen Geschwindigkeitsanpassungen durch NTP oder Ähnlichem bleibt.



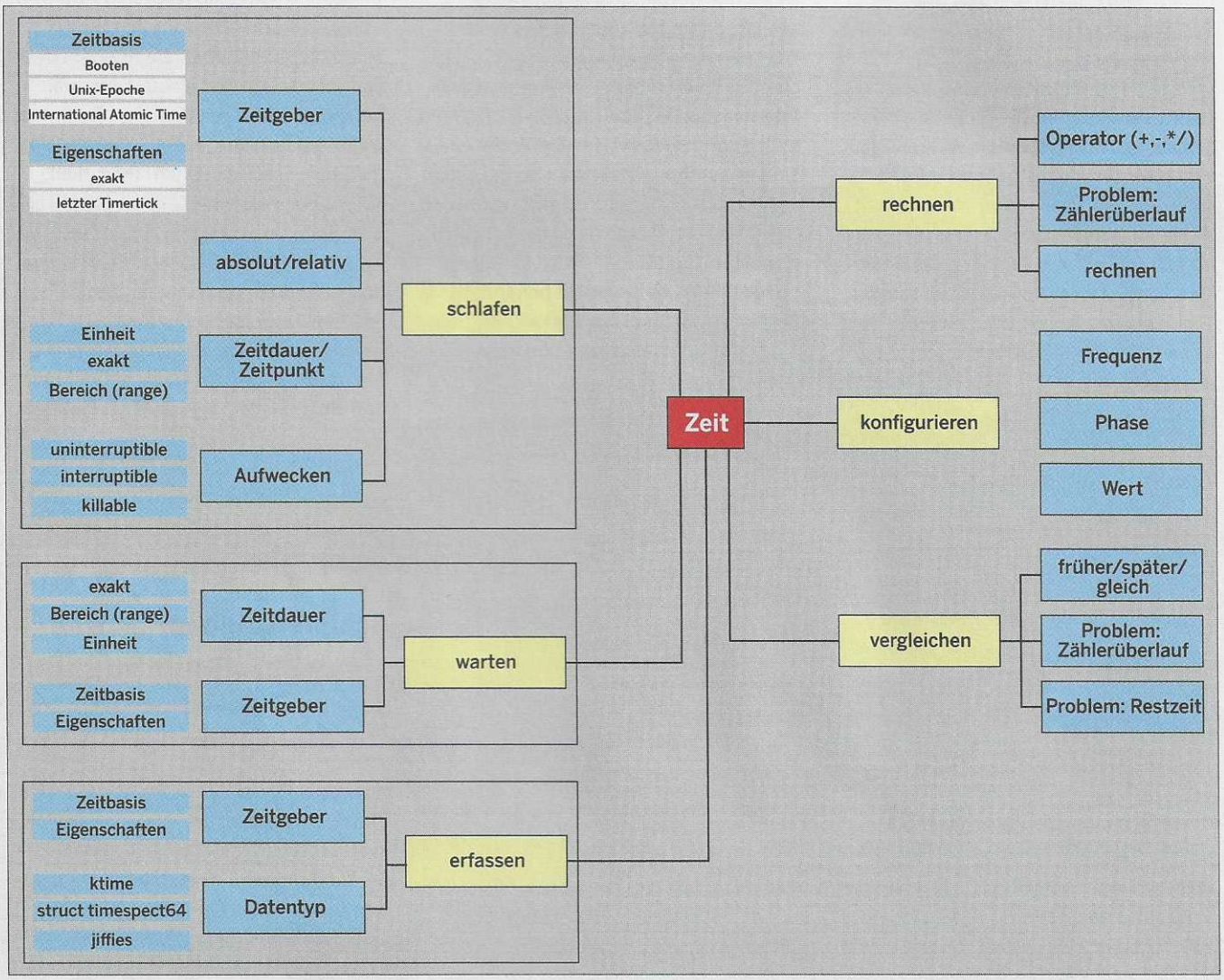
3 **Zentrale Kernel-Datentypen für das Speichern von Zeit.**

Internationale Atomzeit

Bei `CLOCK_BOOTTIME` handelt es sich um eine Verwandte von `CLOCK_MONOTONIC`, nur dass dieser Zeitgeber auch dann weiterzählt, wenn das System sich im Zustand *suspended* befindet, also etwa der Notebook-Deckel zugeklappt ist. Bleibt als Letztes noch `CLOCK_TAI`. TAI steht für Temps Atomique International. Ähnlich wie `CLOCK_MONOTONIC` zählt der Zeitgeber stetig vorwärts, wenngleich die Zeitbasis nicht 1970 ist, sondern 1977. Allerdings lässt sich TAI auch durch hin und wieder eingefügte Schaltsekunden nicht aus dem Tritt bringen. Im Unterschied zu UTC kommen also keinerlei Zeitsprünge vor.

Nomen est omen

Für die Zeiterfassung bietet Linux einen Reigen unterschiedlicher Funktionen an (Tabelle Kernel-Funktionen zur Zeiterfassung). Häufig lässt sich vom Namen ableiten, welchen Zeitgeber sie verwenden und als welchen Datentyp sie den Zählerstand zurückliefern. Ohne besondere Angabe verwenden Zeitfunktionen `CLOCK_MONOTONIC`. Das gilt auch für `ktime_t ktime_get(void)`, das die Zeit als Datentyp `ktime` zurückgibt. Auch



4 **Die Mindmap zum Thema Zeitverwaltung unter Linux verdeutlicht: Zeit ist eine komplexe Angelegenheit.**

`ktime_get_ts64()` liefert den Zählerstand seit dem Booten, allerdings als `struct timespec64`, also als eine Zeit, die eine Angabe in Sekunden und die Restzeit in Nanosekunden macht. Die Variante `ktime_get_real_ts()` gibt einen Zählerwert in Form eines einfachen `struct timespec` zurück, der auf `CLOCK_REALTIME` basiert.

Zusätzlich gibt es noch Varianten, die sich besonders schnell einsetzen lassen, und vor allem in allen Kontexten, insbesondere im Interrupt-Kontext. Zu ihnen zählen neben `u64 ktime_get_mono_fast_ns(void)` und `u64 ktime_get_raw_fast_ns(void)` auch `u64 ktime_get_boot_fast_ns(void)` sowie `u64 ktime_get_real_fast_ns(void)`.

Däumchen drehen

Neben dem Erfassen der Zeit und dem Rechnen damit ist noch das Pausieren beziehungsweise das Schlafenlegen für den (Kernel-)Programmierer relevant. Dabei gilt es, das aktive Warten vom passiven Schlafen zu unterscheiden. Während das aktive Warten weitestgehend unnütz CPU-Leistung verbrennt und damit nur einige Mikrosekunden dauern sollte, kann man durch Schlafenlegen auch längere Zeitspannen überbrücken. Während des Schlafens arbeitet die CPU anderen Code ab oder spart Strom, indem sie sich selbst in einen Schlafzustand versetzt. Für das aktive Warten finden sich in der Kernel-Werkzeugkiste Funktionen wie `ndelay()`, `udelay()` und `mdelay()`. Die Verwendung von `mdelay()` ist schon grenzwertig, denn ein Thread-Wechsel benötigt signifikant weniger Zeit.

Gute Nacht

Schlafen ist komplizierter als Warten. Zum einen müssen die Konditionen definiert sein, unter denen das System schläft. Das geschieht über die Wahl des geeigneten Zeitgebers. Dazu kommen noch Probleme mit dem Aufwecken: Während Wartezeiten im Millisekundenbereich überschaubar sind und kein System wirklich lahmlegen, können sich Jobs auf Jahre hinaus schlafen legen – eine unschöne Angelegenheit.

Daher baut ein halbwegs professioneller Kernel-Programmierer eine Möglichkeit ein, einen Job vorzeitig aus seinen Träumen zu reißen. Für ein solches asynchrones Wecken bietet Linux gleich zwei Alternativen: Die Interruptible-Funktionen reißt es grundsätzlich bei Eintreffen eines Signals aus ihren Träumen, also bei einem Aufweckversuch. Demgegenüber werden die Killable-Varianten nur dann aktiviert, wenn der zugehörige Thread das Signal nicht abfängt, es also nicht behandelt. Ein nicht abgefangenes Signal führt nämlich standardmäßig zum Abbruch des Threads. Die übrigen Varianten sind Uninterruptible, wachen also tatsächlich nur dann auf, wenn die Schlafenszeit abgelaufen ist.

Erlaubt der Programmierer das asynchrone Wecken durch Verwendung der Killable- oder Interruptible-Variante, muss er den Rückgabewert der Funktionen auswerten und eine passende Reaktion definieren. Im einfachsten Fall gibt er einen Fehlercode zurück, oder er sorgt für den Abbruch des Threads (Listing 1).

Die Tabelle Warten und Schlafen führt entsprechende Funktionen auf. Im einfachsten Fall verwendet der Programmierer `msleep()` oder, noch professioneller, `msleep_interruptible()`. Letzteres gibt bei vorzeitigem Wecken die Zeit zurück, die der Thread eigentlich noch hätte schlafen sollen. Diese Funktionen basieren übrigens auf `CLOCK_MONOTONIC`.

Die Funktionen `schedule_hrtimer_timeout()` und `schedule_hrtimer_nanosleep()` erlauben die Auswahl des Zeitgebers. Darüber hinaus kann man bei diesen Varianten über den Parameter `mode` (Typ `hrtimer_mode`) entweder eine minimale Schlafdauer (`HRTIMER_MODE_REL`) angeben oder einen frühesten Aufweckzeitpunkt (`HRTIMER_MODE_ABS`).

Es versteht sich quasi von selbst, dass die Funktionen des passiven Schlafens nicht im Interrupt-Kontext verwendet werden können, also nicht innerhalb einer Interrupt-Service-Routine oder eines Soft-IRQs: Diese Codesequenzen haben keinen Kontext, also keinen Task-Kontrollblock. Daher bekommen sie nicht über den Scheduler Rechenzeit zugeteilt und lassen sich damit generell nicht schlafen legen.

Warten und Schlafen

Warten
`void udelay(unsigned long usecs)`
`void ndelay(unsigned long nsecs)`
`void mdelay(unsigned long msecs)`
Schlafen
`void msleep(unsigned int msecs)`
`unsigned long msleep_interruptible(unsigned int msecs)`
`void usleep_range(unsigned long min, unsigned long max)`
`void ssleep(unsigned int seconds)`
`void fsleep(unsigned long usecs)`
`long hrtimer_nanosleep(ktime_t rqtp, const enum hrtimer_mode mode, const clockid_t clockid)`
`int schedule_hrtimeout_range(ktime_t *expires, u64 delta, const enum hrtimer_mode mode)`
`int schedule_hrtimeout_range_clock(ktime_t *expires, u64 delta, const enum hrtimer_mode mode, clockid_t clock_id)`
`int schedule_hrtimeout(ktime_t *expires, const enum hrtimer_mode mode)`

Schlafeffizienz

Wer glaubt, damit wäre das Thema Zeitgeber und Zeitverwaltung erschöpfend behandelt, der kennt den Linux-Kernel schlecht. Seit vielen Jahren steht neben dem Leistungsdiktat die Forderung nach Effizienz im Raum.

Daher gibt der moderne Programmierer durch Angabe einer Zeitspanne dem Kernel die Möglichkeit, den Aufweckzeitpunkt selbst optimal auszuwählen. Das vermeidet unnötige stromfressende zusätzliche Unterbrechungen. Bevor der Programmierer die entsprechende

Funktion `int schedule_hrtimeout_range(ktime_t *expires, u64 delta, const enum hrtimer_mode mode)` verwendet, muss er den Zustand des Tasks entweder auf `TASK_INTERRUPTIBLE` oder aber auf `TASK_UNINTERRUPTIBLE` setzen. Als Zeitgeber dient `CLOCK_MONOTONIC`. Eine Alternative bietet die Funktion `void usleep_range(unsigned long min, unsigned long max)`, bei der man die minimale sowie die maximale Schlafenszeit in Mikrosekunden angibt.

Die Vielzahl an Funktionen, Varianten und Parametrierungsmöglichkeiten überfordert Gelegenheitsprogrammierer unvermeidlich. Dieser Gruppe empfiehlt Linus Torvalds seit Kernel 5.8 die Funktion `void fsleep(unsigned long usecs)`, die einen Job warten lässt, falls die Verzögerung weniger als 10 Mikrosekunden dauert. Spezifiziert der Programmierer eine Verzögerung von 20 Millisekunden (also 20 000 Mikrosekunden), legt die Funktion den Thread für 20 bis 40 Millisekunden schlafen. Gibt der Programmierer mehr als 20 Millisekunden vor, rundet Linux den Verzögerungswert auf die nächstgrößere Millisekunde auf und schläft für die angegebene Dauer.

Zu guter Letzt bleibt zu erwähnen, dass sich die Klassiker unter den Schlaffunktionen, `schedule_timeout_killable()`, `schedule_timeout_interruptible()` und `schedule_timeout_uninterruptible()` nach wie vor im Kernel finden. Sie bekommen die Schlafenszeit in `jiffies` übergeben, als Zeitgeber fungiert `CLOCK_MONOTONIC`. Sollte man sie in neuem Kernel-Code noch verwenden? Die Antwort ist ein definitives Nein. *(jlu)*

Listing 1: Schlaf gut

```
timeout=8000000000; // Thread wartet fuer 8 Sekunden
while (timeout!=0) {
  set_current_state(TASK_INTERRUPTIBLE);
  timeout = schedule_hrtimeout_range_clock(timeout, 20000, HRTIMER_
MODE_REL, CLOCK_MONOTONIC);
  if (signal_pending(current)) {
    printk("early woke up by remaining %ld ns\n", timeout);
    retval = -ERESTARTSYS;
    break;
  }
}
```

Weitere Infos und interessante Links
www.lm-online.de/qr/44069