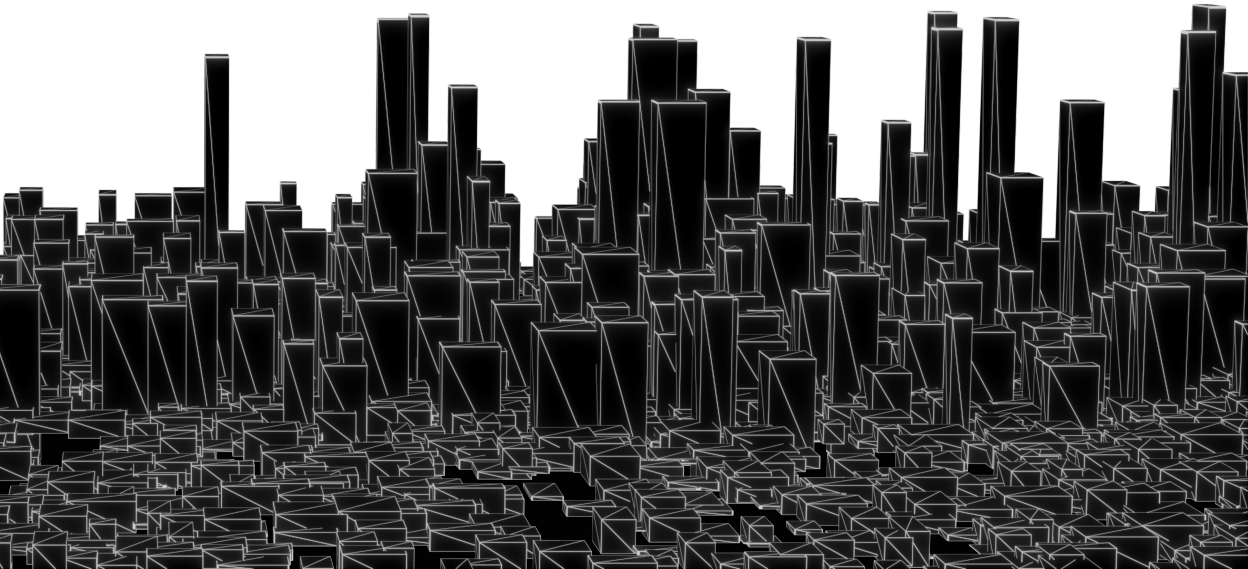
Hagen Paul Pfeifer

Linux-Systemanalyse

Von High-Level-Architekturanalysen zu Low-Level-Code-Optimierungen

1. Auflage

München – 2025



Kapitel 9

Zeit und Timer

Viele Aspekte des Lebens und der Wirklichkeit werden durch die Zeit strukturiert. Durch sie erhalten Abläufe eine Ordnung, Kausalitäten werden ermöglicht und ein Rahmen geschaffen, in dem Ereignisse verortet und koordiniert werden. Ihr Verständnis gilt in wissenschaftlichen, philosophischen und praktischen Kontexten als zentral, da sie als übergeordneter Bezugsrahmen dient.

Auch in der Informatik spielt die Zeit eine grundlegende Rolle – man denke etwa an den Taktsignalgeber der CPU, den Prozessortakt – den Herzschlag eines Computersystems. Zeitpunkte und Zeitdifferenzen bilden die Grundlage für eine Vielzahl von Operationen und Steuermechanismen in der Informationsverarbeitung. Insbesondere Timer werden eingesetzt, um Aktionen gezielt und zu definierten Zeitpunkten auszulösen. Sie sind unverzichtbar für periodische Aufgaben mit festen Intervallen und leisten einen wesentlichen Beitrag zur Organisation, Synchronisation und zeitlichen Steuerung von Systemprozessen.

Bereits in vorhergehenden Kapiteln wurden Zeit- und Timeraspekte an verschiedenen Stellen behandelt. Dieses Kapitel widmet sich nun gezielt der Analyse von Timern innerhalb von Applikationen, aber auch im Kernel. Timer gehören – neben „Daten-Interrupts“ – zu den Ereignissen, die eine Anwendung wieder in den ausführbaren Zustand versetzen können und haben somit erheblichen Einfluss auf das Laufzeitverhalten eines Systems[1].

[1] In aller Regel blockieren Anwendungen in datengetriebenen Systemaufrufen wie `read()`, etwa beim Warten auf Benutzereingaben oder Netzwerkpaketen, oder bis ein Timeout eintritt. Alternativ kann ein Task auch ohne blockierende Operationen kontinuierlich rechnen, beispielsweise in HPC, Batch, Machine-Learning-Trainingsjobs oder Video-Encode Workloads, dies stellt jedoch eine seltenere Ausnahme dar. Über dieses grundsätzliche Prozessverhalten lässt sich bei Gelegenheit durchaus einmal nachdenken.

9.1. Einführung

Im Linux-Kernel kommt traditionell das Konzept der *Jiffies* zum Einsatz, um Zeiteinheiten zu repräsentieren. Diese Jiffies basieren technisch auf einer nicht-negativen, ganzzahligen Zählervariable (unsigned long). Diese Variable wird durch Timerinterrupts 100, 250 oder 1000 pro Sekunde inkrementiert[2]. Hierbei erfolgt eine Erhöhung des Zählers, der die Anzahl der Timerinterrupts seit des Bootvorgangs darstellt.

Die grundlegende Idee ist, dass das gesamte System durch den Timerinterrupt regelmäßig unterbrochen wird. Dieser Interrupt tritt alle 10 ms, 4 ms oder sogar jede Millisekunde auf, abhängig von der konkreten Kernel-Konfiguration. Die Jiffies dienen somit als eine Art Zeitgeber, der es dem Betriebssystem ermöglicht, die vergangene Zeit zu verfolgen und zeitbasierte Aktionen durchzuführen. Dieses Konzept wird in vielen Teilen des Kernels verwendet, um Zeitmessungen und Zeitsteuerungen zu ermöglichen.

Die Variable zählt also schlicht die Unterbrechungen, welche für den Timer Interrupt ausgelöst wurden. Über Hilfsfunktionen kann der Jiffiswert in Millisekunden und andere Maßeinheiten konvertiert werden. Es ist ersichtlich das dieser Zähler grundsätzlich nicht an die „Wanduhrzeit“[3] gebunden ist oder gar repräsentiert. Oft ist für die Nutzer die Uhrzeit auch irrelevant. Beispielsweise wenn der Prozessscheduler erfasst wie lange der Prozess bereits ausgeführt wurde, ist lediglich die zeitliche Differenzinformation notwendig – mehr nicht. Ab der Prozess beispielweise am Tag oder in der Nacht gestartet wurde, ist für die Entscheidungen welcher der Prozess-Scheduler trifft, nicht notwendig. Jiffies sind also vielmals eine adäquate Basis für viele Entscheidungen im Kernel.

Die Uhrzeit wurde dann, aufbauend auf den Jiffies als „Building Block“ aufgebaut. Architekturspezifische Mechanismen sorgen dafür, dass eine höhere Auflösung als die Timer Tick Auflösung vorhanden war. Auch das Network Time Protokoll (NTP), ein Standard zur Synchronisation der Uhrzeit über ein Netzwerk, basierte im Kern auf Jiffies als Grundbaustein.

> **Hinweis 9.1** Bei zeitbasierten Messungen wird grob zwischen zwei Arten unterschieden: der *CPU-Zeit* und der *Systemzeit* (engl. *Wall-Clock Time*). Die *CPU-Zeit* steigt bei paralleler Ausführung proportional zur Anzahl der verwendeten Kerne und summiert

[2] Der Periodische Tick wurde seit Beginn des Linux Kernels 100 mal pro Sekunde geupdated. In der Entwicklung des 2.5 Kernels wurde der Wert für einige Architekturen auf 1000 Hz geändert und letztendlich auf 250 Hz korrigiert.

[3] Uhrzeit, weltweit standardisiert in Zeitzonen und unterteilt Stunden, Minuten und Sekunden. Also die reale, fortlaufende Zeit für Menschen auf der Armbanduhr.

sich entsprechend schneller auf. Im Gegensatz dazu verläuft die *Systemzeit* stets linear zur realen Zeit. Für valide Messergebnisse ist die Wahl der geeigneten Zeitreferenz also entscheidend.

Seit Beginn des Linux Kernels sind also Jiffies das dominierende Zeitsystem. Timer wurden mithilfe eines Timerwheel implementiert. Ein Timerwheel ist eine spezielle Datenstruktur, die oft in der Implementierung von Kernel Timern verwendet wird – nicht nur im Linux-Kernel. Es ist eine zyklische Liste von Timer-Einträgen, die in Abschnitten unterteilt ist, die als „Slots“ oder „Bucket“ bezeichnet werden. Jedes Slot repräsentiert einen bestimmten Zeitbereich. Timer-Einträge werden in die entsprechenden Slots basierend auf ihrem Ablaufzeitpunkt eingeordnet. Wenn ein Timer für eine bestimmte Zeitdauer eingestellt wird, wird er in den entsprechenden Slot im Timerwheel eingefügt. Bei jedem Timer-Interrupt wird der Timerwheel weiterbewegt, und die Timer-Einträge, deren Ablaufzeitpunkt erreicht ist, werden aktiviert. Jiffies dienen als Grundlage für die Zeiterfassung und ermöglichen es dem Kernel, zeitbasierte Operationen effizient zu planen und auszuführen.

9.2. High Resolution Timer

Die Implementierung eines Zeitsystems mittels Jiffies und Timerwheels zeichnet sich durch ihre Effizienz aus. Dieser Mechanismus findet ebenfalls in anderen Betriebssystemen Anwendung, beispielsweise in FreeBSD. Seit den Anfängen der Entwicklung des Linux-Kernels ist dieses Zeitsystem unverändert geblieben und wurde nicht durch eine grundlegend „bessere“ Alternative ersetzt.

Im Laufe der Jahre wurden auf der Linux Kernel Mailing List (LKML) zahlreiche neue Ideen und grundlegende Überarbeitungen vorgestellt. Keiner dieser Vorschläge konnte sich jedoch durchsetzen. Erst mit dem Entwicklungszyklus der Linux-Version 2.6, gegen Ende des Jahres 2005, wurde ein weiteres Zeitsubsystem in den Kernel integriert. Die Hauptprotagonisten dieser Entwicklung waren Thomas Gleixner und Ingo Molnar. Diese Entwicklungen sind unter der Bezeichnung „High Resolution Timer“ (HRTimer) bekannt geworden[4].

Die grundlegenden Unterschiede zum bestehenden Zeitsubsystem, sind folgende Eigenschaften:

- Red Black Tree basierte Datenstruktur für die Verwaltung von Timer. Es kommt

[4] Laut den Kommentaren von Thomas Gleixner und Ingo Molnar basieren die Ideen und Arbeiten auf zahlreichen früheren Entwicklungen, die es nicht in den offiziellen Kernel geschafft haben.

keine Timerwheels basierte Datenstruktur zum Einsatz.

- Es wird konsequent auf die Nutzung von 64-Bit-Werten gesetzt, um eine höhere Präzision und eine exakte Zeitmessung zu ermöglichen. HRTimer sind für Auflösungen unterhalb einer Millisekunde ausgelegt.
- Nutzung hochauflösender Hardware Timer.

Im Linux-Kernel werden zwei unterschiedliche Zeitframeworks bereitgestellt: Zum einen der klassische Kernel Timer, der historisch gewachsen ist, und zum anderen die High Resolution Timer, kurz HRTimer genannt. Der Kernel Timer stützt sich auf Jiffies als Zeitbasis, während die HRTimer auf hochauflösende Zeitquellen zurückgreifen. Mit der Implementierung der HRTimer ging eine umfassende Überarbeitung der darunterliegenden Infrastruktur einher. Diese Überarbeitung hat eine wartungs- und erweiterungsfähige Timerinfrastruktur geschaffen. Im nachfolgenden Abschnitt wird dies detaillierter erörtert.

Falls Sie sich für die Implementierung des Kernel Timers oder des HRTimers interessieren finden Sie diese in den Kernelquellen unter `kernel/timer.c`, respektive `kernel/hrtimer.c`.

9.3. Clock Sources und Clock Events

Clock Sources sind Zeitquellen, die den aktuellen Zeitstand im System liefern. Sie sind grundlegend für Funktionen wie Zeitstempel und Messung der Zeit. Während des Bootvorgangs werden die vorhandenen Zeitquellen initialisiert. Oft stehen mehr als eine Zeitquelle zur Verfügung, der Kernel selektiert die passenste unter Bewertung der folgenden Eigenschaften: Granularität, Leselatenz, Per-CPU Eigenschaften und Stabilität . Durch Bootparameter und über das sysfs kann die Zeitquelle modifiziert werden.

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm

$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

Auflistung 9.1: Verfügbare und Tatsächliche Clock Source

In Auflistung 9.1 wird die Ausgabe der verfügbaren Zeitquellen für ein x86-64 System dargestellt.

- **High Precision Event Timer (HPET)** ist eine Uhr und Timer, die auf x86-64 Plattformen verfügbar ist. Er hat eine gute Granularität, typischerweise von 10 MHz

oder 20 MHz, was eine Genauigkeit von bis zu 50 Nanosekunden ermöglicht. Die Clock ist nicht so schnell zu lesen wie der TSC und ist daher eine Fallbacklösung auf x86-64 Systemen wenn die TSC instabil ist.

- **Time Stamp Counter (TSC)** ist ein Taktgeber, der bei x86-Architekturen im Prozessor Core implementiert ist. Er zählt die Anzahl der Zyklen seit dem Reset des Prozessors mit der Frequenz des „Reference Cycle“ – nicht der CPU Core Clock Cycle. Auf einem Intel i9-13900K beträgt die Auflösung 3 GHz. Im Zugriff ist der TSC sehr Latenzarm da es sich um ein Register handelt. Je nach Prozessormodel aber auch ein wenig schlechter als ein Cache-Miss. Die Stabilität und Zuverlässigkeit varriert je nach Prozessormodell.
- **ACPI Power Management Timer (ACPI MP)** ist eine einfache Uhr und Timer. Er ist in der Regel genauso langsam zu lesen wie der HPET, aber mit einer geringeren Granularität im im MHz Bereich.

Hinweis 9.2 Der Time Stamp Counter (TSC) wurde historisch mit der konstanten Rate der Core-Frequenz inkrementiert. Seit der Einführung des Dynamic Voltage and Frequency Scaling (DVFS) - einer Technik zur Reduzierung des dynamischen Stromverbrauchs durch Anpassung von Spannung und Frequenz der CPU - ist diese Frequenz nicht mehr stabil. Dies führte zu einer reduzierten Eignung des TSC als hochauflösende Zeitquelle. Zur Beibehaltung einer konstanten Rate über P-State hinweg, wurde mit der Einführung des Pentium 4 das Feature „Constant TSC“ eingeführt. Bei den C-States erfolgte die Inkrementierung des TSC bis C1/Auto Halt konstant. Für eine konstante Inkrementierung auch in tieferen Schlafphasen wurde das Feature „Invariant TSC“ entwickelt. Es gewährleistet, dass der Zähler auch in höheren Schlafphasen inkrementiert wird. Also vollständig unabhägig von C-, P- oder T- States. Ob ein Prozessor Invariant TSC unterstützt, kann auschließlich über die beiden CPU Feature Flags `constant_tsc` und `nonstop_tsc` in `/proc/cpuinfo` ermittelt werden – beide Flags müssen enthalten sein. Nonstop deutet es bereits an: der TSC ist auch konstant bei höheren C-State Schlafzuständen. Die TSC-Ticks zählen also die verstrichene Zeit und nicht die Anzahl der CPU-Taktzyklen.

In Auflistung 9.2 eine Applikation um die *Reference Cycle Frequenz* auf einem System zu bestimmen:

```
1 int main(void)
2 {
3   unsigned long long elapsed_cycles;
4   unsigned long long start_tsc, end_tsc;
5   struct timespec start_time, end_time;
6   double elapsed_time, cpu_clock_speed;
7
```

```
  clock_gettime(CLOCK_MONOTONIC, &start_time);
  start_tsc = __rdtsc();

  sleep(10);

  end_tsc = __rdtsc();
  clock_gettime(CLOCK_MONOTONIC, &end_time);

  elapsed_time = (end_time.tv_sec - start_time.tv_sec);
  elapsed_time += (end_time.tv_nsec - start_time.tv_nsec) / 100000000;

  elapsed_cycles = end_tsc - start_tsc;
  cpu_clock_speed = elapsed_cycles / elapsed_time;

  printf("Elapsed Time: %f seconds\n", elapsed_time);
  printf("Elapsed Cycles: %llu\n", elapsed_cycles);
  printf("Estimated CPU Clock Speed: %lf GHz\n", cpu_clock_speed / 1000000000.0);

  return 0;
}
```

Auflistung 9.2: Berechnung der CPU-Taktfrequenz durch Auswertung von Taktzyklen und Zeitdifferenz während eines definierten Schlafintervalls.

Die Ausgabe auf einem Intel i9-13900K System ist in Auflistung 9.3 abgebildet. Zu erkennen: der Referenztakt beträgt auf diesen System 2.995 GHz. Also einen Multiplikator von 30 bei einem *Motherboard Master Clock Frequenz* von 100 MHz. Diese Daten können alternativ via `dmidecode -t processor` ausgelesen werden. *External Clock* und *Current Speed* sind die relevanten Einträge. Den Multiplikator errechnet man aus der Division von *Current Speed* und *External Clock*. Ein weiterer Weg, mit der Kenntnis im Hinterkopf das die Kernel Initialisierung in der Bootphase grundlegende Entscheidungen legt: `tsc: Detected 2995.200 MHz TSC`, via `journalctl -b | grep -i tsc -`.

```
Elapsed Time: 10.000000 seconds
Elapsed Cycles: 29951153940
Estimated CPU Clock Speed: 2.995115 GHz
```

Auflistung 9.3: Ermittelte Ausführungsdauer, Taktzyklen und daraus abgeleitete CPU-Taktfrequenz nach einer definierten Pause.

Neben den passiven Zeitquellen, den Clock Sources, bilden die Clock Events die zweite notwendige Grundlage für die Implementierung eines Zeitsystems. Während Clock Sources in der Regel einfache Zählregister darstellen, bieten Clock Events die Möglichkeit, Timer zu implementieren – also Elemente, die nach einer definierten Zeit eine Aktion auslösen. In Auflistung 9.4 auf der nächsten Seite ist das gewählte Clock Event Device die `lapic-deadline` Quelle. Es handelt sich dabei um einen in jeden CPU-Core lokal integrierten Timer des Advanced Programmable Interrupt Controllers (LAPIC), der mit dem Kernquarz-Taktgeber (Core Crystal Clock) getaktet wird. Dieser Taktgeber arbeitet mit einer Frequenz von 38,4 MHz, was einer maximalen Granularität von 26,04 Nanosekunden entspricht.

Der Zusatz *deadline* verrät das die Quelle an den lokalen TSC gebunden wird und damit letztendlich eine höhere Granularität erreicht.

```
$ cat /sys/devices/system/clockevents/clockevent0/current_device
lapic-deadline
```

Auflistung 9.4: Clock Event Quelle

Über das `proc` Filesystem kann die konkrete Anzahl der Core-lokalen Interrupts seit dem Bootvorgang abgefragt werden: `cat /proc/interrupts | grep LOC -`. Wie in Auflistung 9.5 dargestellt, gibt ein einfaches Python-Script die pro Sekunde auftretenden Timer-Interrupts aus. Das Script kann beliebig erweitert werden, um statistische oder Core-spezifische Informationen auszuwerten.

```
import time

def read_loc_interrupts() -> int:
    with open('/proc/interrupts', 'r') as file:
        for line in file:
            if 'LOC' not in line: continue
            parts = line.split()
            return sum(int(value) for value in parts[1:] if value.isdigit())
    return 0

irq_prev = read_loc_interrupts()
while True:
    time.sleep(1)
    irqs = read_loc_interrupts()
    print(irqs - irq_prev)
    irq_prev = irqs
```

Auflistung 9.5: Python Script zur Aufwertung des IRQ Timer: Kontinuierliches Auslesen und Differenzbildung der lokalen Interrupts aus */proc/interrupts* zur Überwachung pro Sekunde.

In Auflistung 9.6 wird die beispielhafte Ausführung des Scripts dargestellt, das auf einem standardmäßigen, Gnome-basierten System ausgeführt wurde. Auf diesem System fallen 700 Timer Interrupts pro Sekunde auf allen Cores an. Ab Zeile 10 ist eine erhebliche Erhöhung der Anzahl der Timer-Interrupts zu beobachten. Die künstlich verursachte Ursache dafür ist, dass mit dem Surfen auf heise.de mittels Chromium begonnen wurde.

```
$ python3 irq-timer-counter.py
657
729
644
627
696
844
629
722
1320  # start surfing heise.de
5560
4142
5856
```

```
2494
5255
3918
```

Auflistung 9.6: Ausgabe des Python Scripts zur Aufwertung des IRQ Timer mit beobachtbaren Schwankungen der lokalen Timer-Interrupts.

Clock Event `lapic-deadline` ist ein CPU Core lokaler Timer. Sollte die CPU in einen C-State Schlafzustand sein, kann ein Aufwachen nicht getriggert werden. Der Kernel stellt dies sicher und nutzt bei einem Schlafzustand auf eine Core unabhängigen Unterbrecher, den Prozessor globalen HPET. Siehe dazu `.../clockevents/broadcast/current_device` und weitere Ausführungen auf auf Seite 605.

Auf ARM oder RISC-V basierten Archituren unterscheiden sich die Clock und Event Sourcen vollumfänglich. Auf einem Raspberry PI beispielsweise ist Clock Source der `arch_sys_counter` und Clock Event der `arch_sys_timer`. Alternative Zeitquellen oder Timer werden nicht bereitgestellt. Als Grundlage dient auf ARM basierten Systemen der sogenannte Generic Timer. Details sind im Referenzdokument zu entnehmen[43].

Bei der Ausführung des Scripts auf einem Vanilla Raspbian ohne grafische Benutzeroberfläche[5] zeigt sich ein deutlich entspannteres Bild. Dies wird in Auflistung 9.7 deutlich.

```
$ python3 irq-timer-counter.py
16
15
17
14
16
14
16
15
13
```

Auflistung 9.7: Stabile Ausgabe des Python Scripts zur Aufwertung des IRQ Timer auf einem Raspberry Pi bei geringer Systemlast mit konstant niedrigen Interrupt-Werten.

Bei der Ausführung auf einem Vier-Kern-System werden weniger als 20 Timer-Interrupts pro Sekunde ausgelöst. Unter der stark vereinfachten Annahme einer Gleichverteilung entspräche dies einem Timer-Event alle 200 ms pro Kern.

> **Hinweis 9.3** Nicht jeder Kernel-Timer löst zwangsläufig einen Interrupt aus. Zur Reduktion der Systemlast durch Interrupts wird vom Betriebssystem angestrebt, mehrere Timer möglichst synchron ablaufen zu lassen und deren Ereignisse gebündelt zu verarbeiten. Dieses Verfahren wird als *Timer Coalescing* bezeichnet. Darüber hinaus stellen

[5] Das Python3-Script muss so angepasst werden, dass der Suchstring `LOC` durch `arch_timer` ersetzt wird

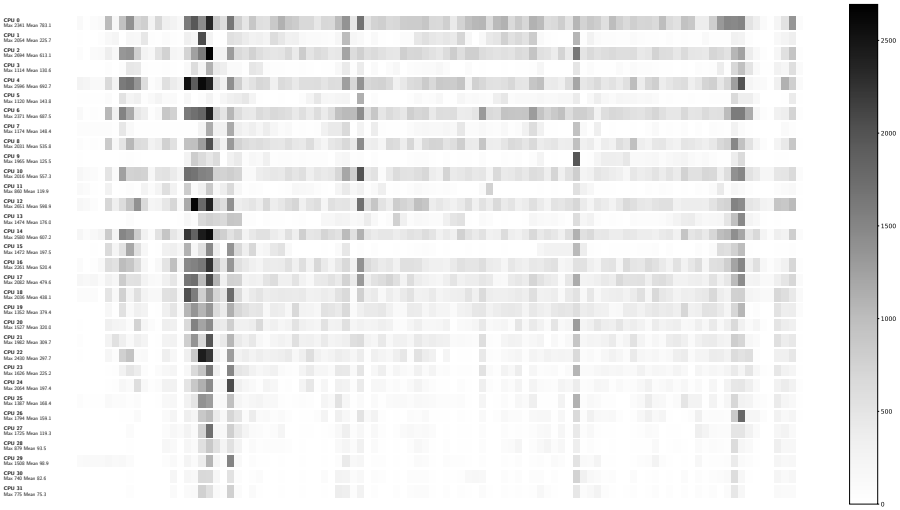


Abbildung 9.1.: Timer basierte CPU Wakeups über die Zeit; CPU separiert auf einem 32-Kern System mit anfänglich höherer Timer-Last.

unterschiedliche Prozessorarchitekturen spezifische Mechanismen für den Umgang mit Timern bereit. Die ARM-Architektur bietet beispielsweise die Instruktion `Wait for Event` (WFE), mit der der Prozessor in einen energiesparenden Schlafzustand versetzt wird, bis ein Ereignis eintritt. Timer-Ereignisse können dabei signalisiert werden, ohne einen dedizierten Interrupt auszulösen. Nach dem Aufwachen aus dem WFE-Zustand kann der Prozessor den Status der Timer-Ereignisse direkt über spezielle Register abfragen. Ein separater Interrupt-Mechanismus ist hierfür nicht zwingend erforderlich.

9.4. Tickless

Für das Laufzeitverhalten ist es wichtig zu verstehen, wie der Kernel mit Timer-Interrupts im Leerlauf umgeht. Traditionell war hier der periodische Ticking-Modus üblich,

aktiviert über die Kerneloption CONFIG_HZ_PERIODIC. In diesem Modus führt der Kernel ständig zyklische Timer-Interrupts aus – unabhängig davon, ob ein Core gerade arbeitet oder im Leerlauf steckt. Dadurch wird regelmäßig geplanter Code ausgeführt, doch auch auf inaktiven Cores pochen die Interrupts weiter – was zu unnötiger Energie- und Ressourcenbeanspruchung führt

Wird kein Task zur Ausführung bereitgehalten und spielt das Power-Management eine wesentliche Rolle, stellt die kontinuierliche Auslösung von Interrupts eine vermeidbare Systembelastung dar. Durch Aktivierung der Option CONFIG_NO_HZ_IDLE wird auf Systemen im *Idle*-Zustand die Erzeugung regelmäßiger Timer-Interrupts unterbunden. Der Timer wird so konfiguriert, dass ein Interrupt ausschließlich bei tatsächlichem Ablauf eines Timers ausgelöst wird. Dieses Verfahren wird als *Dyntick Idle* bezeichnet. Konkret: sind im Kernel- oder Userspace keine unmittelbar anstehenden Timer-Events vorhanden, erfolgt keine Programmierung eines periodischen Timers. Stattdessen wird ein One-Shot-Timer auf den Zeitpunkt des nächsten fälligen Timers gesetzt. Der periodische 100 Hz-, 250 Hz- oder 1 000 Hz-Timer-Tick entfällt somit. Anstelle eines festen Takts wird nur noch ein einzelner Timer programmiert, der beim nächsten fälligen Ereignis ausgelöst wird.

Sobald die CPU aus dem Idle-Zustand erwacht – etwa durch den Ablauf des One-Shot-Timers oder das Auftreten eines Tasks (z.b. getriggert durch ein Netzwerk IRQ) –, wird der periodische Timer durch den Kernel erneut aktiviert und der normale Taktbetrieb wieder aufgenommen. Dieses Verfahren bleibt aktiv, bis erneut Idle-Bedingungen erkannt werden.

Zur weiteren Reduzierung der Interrupt-Anzahl über reine Idle-Perioden hinaus wurde die Option CONFIG_NO_HZ_FULL eingeführt. In diesem Modus wird der Clock Tick nicht nur im Idle-Zustand deaktiviert, sondern auch dann, wenn lediglich ein einzelner Prozess zur Ausführung bereitsteht. Ziel ist es, regelmäßige Interrupts zu vermeiden, sofern kein weiterer Task verfügbar ist.

> **Hinweis 9.4** Auch wenn CONFIG_NO_HZ_FULL zunächst vorteilhaft erscheint, sind mit der Aktivierung mehrere Nachteile verbunden. Daher sollte sie ausschließlich in begründeten Ausnahmefällen, etwa in harten Echtzeitsystemen, eingesetzt werden. Durch die Aktivierung entstehen zusätzliche Kosten bei Übergängen zwischen User- und Kernelmodus. Jeder dieser Übergänge wird komplexer, da betroffene Subsysteme wie RCU benachrichtigt werden müssen. Diese erhöhte Komplexität führt zu einer messbaren und spürbaren Zunahme der Latenz.

Timer-basierte Interrupts bleiben jedoch weiterhin essenziell für zentrale Kernelmechanismen wie Scheduling-Fairness und System-Bookkeeping. Um sicherzustellen, dass auch

dauerhaft laufende Userspace-Applikationen – wie etwa `main() { do { } while(42);}` – regelmäßig unterbrochen werden, muss eine periodische Rückkehr in den Kernel erfolgen und dieser wird durch regelmäßige Timer-Interrupts erzwungen.

Die Entscheidung zur Deaktivierung des Clock Ticks wird vom *CPU Idle Governor* getroffen. Dieser Komponente wird das notwendige Wissen, beispielsweise über den Ablaufzeitpunkt des nächsten Timers, zur Verfügung gestellt, um eine fundierte Entscheidung zu ermöglichen.

Ein Problem entsteht, wenn CPU-Cores in tiefe Schlafzustände wie C3 oder C6 überführt werden. In diesen Zuständen wird je nach Featureset der lokale LAPIC-Timer deaktiviert[6], wodurch keine Timer-Ereignisse mehr empfangen werden können. Um dennoch sicherzustellen, dass diese Cores bei einem anstehenden Ereignis geweckt werden, wird das Timer Broadcast Framework eingesetzt. In diesem Fall übernimmt ein aktiver Core oder ein globaler Timer – etwa ein auf One-Shot-Modus konfigurierter HPET – die Aufgabe, betroffene Cores durch das Senden eines Broadcast-Interrupts (IPI) zu aktivieren. Auf diese Weise können auch Cores im tiefen C-State zuverlässig geweckt werden.

Im regulären Betrieb ist dieses Framework meist weniger bedeutend, da lokale Timer in der Regel ausreichen. In Analyse- und Debugging-Szenarien des Kernels, bei der Entwicklung energieeffizienter Power-Management-Konzepte oder in hochpräzisen, latenzkritischen Umgebungen ist das Broadcast-Verfahren jedoch wichtig und die wirkweise sollte für potentiell notwendige Analysen verstanden sein – insbesondere in HPC-Umgebungen oder Echtzeitsystemen.

In Auflistung 9.8 ist die Kernelkonfiguration eines Vanilla Debian Kernels aufgelistet. Der Standard-Debian Kernel unterstützt also den Tickless Mode: `CONFIG_NO_HZ_FULL`. Der reguläre Timertick liegt bei 250 Hz.

```
$ cat /boot/config-$(uname -r) | grep HZ
CONFIG_NO_HZ_COMMON=y
# CONFIG_HZ_PERIODIC is not set
# CONFIG_NO_HZ_IDLE is not set
CONFIG_NO_HZ_FULL=y
# CONFIG_NO_HZ is not set
# CONFIG_HZ_100 is not set
CONFIG_HZ_250=y
# CONFIG_HZ_300 is not set
# CONFIG_HZ_1000 is not set
CONFIG_HZ=250
```

Auflistung 9.8: Auszug der Kernel-Konfiguration mit Fokus auf Zeiteinstellungen und aktivierte HZ-Optionen.

[6] Modernere CPU Cores halten den lokalen APIC oft am Leben, zumindest die Komponenten welche für das Timerhandling zuständig sind – sprich der LAPIC-Timer funktioniert auch im tiefen C-States. Stichwort ist hier *ARAT* (“Always Running APIC Timer”). Wenn dies zutrifft sind „Broadcast-Verrenkungen“ nicht notwendig.

Die Option `CONFIG_NO_HZ_COMMON` kann ignoriert werden: diese schaltet lediglich die Basiskomponenten frei, welcher ein Tickless Kernel technisch ermöglicht - ob der Tickless Mode tatsächlich aktiv ist, kann und wird über Boot-Parameter gesteuert werden.

In Auflistung 9.9 ist das Verhalten des Timer Ticks mittels eines Scripts analysiert worden. Als Basisevent wurde für diese Analyse `timer:tick_stop` verwendet, sowie der zugehörige Aufruf der Kernel-Funktion `tick_sched_timer` extrahiert. In der nachfolgenden aufgezeigten Analyse wird das Verhalten eines Tickless System ein wenig „spürbar“.

```
            Time  CPU           Event Success Dependency Kernel-Function
11060.262656178   19       TICK_STOP       1          0 None
11060.262748987   20       TICK_STOP       1          0 None
11060.262822935   21       TICK_STOP       1          0 None
11060.262882346   22       TICK_STOP       1          0 None
11060.262944061   23       TICK_STOP       1          0 None
11060.263034752   24       TICK_STOP       1          0 None
11060.263174075    2    EXPIRE_ENTRY      -1       None tick_sched_timer
11060.263174239    8    EXPIRE_ENTRY      -1       None tick_sched_timer
11060.263182312   26       TICK_STOP       1          0 None
11060.263241470   27       TICK_STOP       1          0 None
11060.263465690   30       TICK_STOP       1          0 None
11060.265324779    8    EXPIRE_ENTRY      -1       None watchdog_timer_fn
11060.267187041    0       TICK_STOP       1          0 None
11060.267187678    2    EXPIRE_ENTRY      -1       None process_timeout
11060.267189072   31    EXPIRE_ENTRY      -1       None tick_sched_timer
11060.267195190    2       TICK_STOP       1          0 None
[...]
```

Auflistung 9.9: Auflistung von Kernel-Ereignissen mit Fokus auf *TICK_STOP* und *EXPIRE_ENTRY* Aktivitäten über mehrere CPUs.

In der Visualisierung in Abbildung 9.2 auf der nächsten Seite wurden diese Daten grafisch dargestellt. Eine horizontale Hilfslinie markiert die konfigurierte Timer-Tick-Frequenz von 250 Hz, entsprechend einem Intervall von 4 ms. Unterhalb dieses Wertes tritt kein `tick_sched_timer`-Differenzwert auf - das ist leicht erkennbar.

Vertikale Linien kennzeichnen Timer-Tick-Deaktivierungen, während Kreise den Ablauf des `tick_sched_timer`-Timers repräsentieren. In Phasen hoher Auslastung – exemplarisch dargestellt im linken oberen Teil des Bildes für CPU0 – wird der Timer Tick nur selten deaktiviert. Deutlich erkennbar ist, dass der regelmäßige 250 Hz-Timer-Tick den Scheduling-Prozess antreibt.

Auf weniger ausgelasteten CPUs, wie in diesem Fall CPU3, wird der Timer Tick seltener deaktiviert, aber auch seltener ausgelöst, wodurch die Darstellung vergleichsweise „leer“ erscheint.

Für CPU1 zeigt sich ein interessanteres Verhalten: Trotz einer mittleren Anzahl an `tick_sched_timer`-Abläufen ist ausreichend Spielraum vorhanden, um den Timer Tick dennoch deaktivieren zu können.

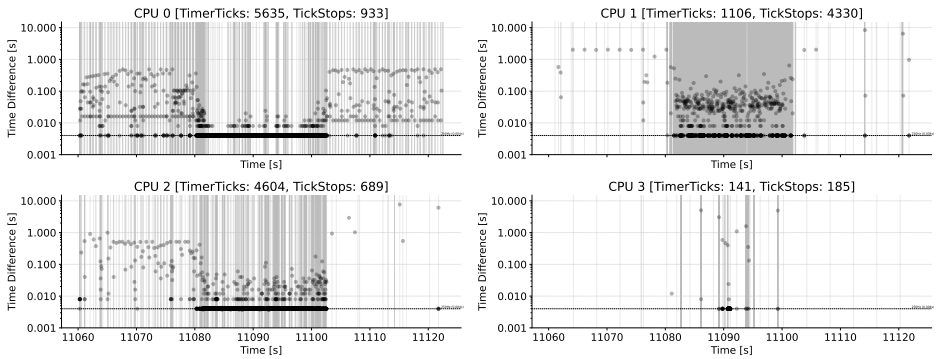


Abbildung 9.2.: Visualisierung Scheduler Timer Tick und Verhalten in Tickless Systemen.

9.5. Timer-Lebenszyklus

Für die Analyse von Kernel und Userspace initierten Timer ist ein Basiswissen über den Lebenszyklus der Timer notwendig. Dies wird in diesen Abschnitt vermittelt.

Grundsätzlich kann man die Timer Lebenszyklen in zwei Phasen unterteilen. Der Aufbau und Startphase sowie die Ausführungs- und Abbauphase. In jeder Phase und für ältere Kernel- sowie hochauflösende HRTimer stehen eine Reihe von Funktionen bereit. Im Nachfolgeneden werden nur die Hauptfunktionen des Standard Lebenszyklus für beide Timer Ausprägungen erläutert:

- **Aufbau- und Startphase**
  - Timer deklarieren, Strukturen allokieren
  - Timer Initialisieren: `timer_setup` oder `hrtimer_init`
  - Timer starten: `mod_timer` oder `hrtimer_start`
- **Ausführungs- und Abbauphase**
  - Callback wird aufgerufen („expire“) (optional, Timer werden teilweise vor Ablauf beendet, denken sie an das letzte Argument von *select(2)*, `struct timeval *timeout`)
  - Timer wird neu gestartet: `mod_timer`, `hrtimer_forward_now` (optional)
  - Timer aufräumen: `del_timer_sync`, `hrtimer_cancel`

Hinweis 9.5 Wenn ein Timer abläuft, also *expired* und dieser umfangreichere Berechnungen im Kernel durchführen muss, wird dies nicht zwangsläufig im Interrupt Kontext des Timer Interrupts ausgeführt. Timer werden konkret im Kontext eines SoftIRQ ausgeführt. Also einem Kerneltask welcher speziell für die Bottom-Half Bearbeitung eines Interruptes erschaffen wurde. Für schnelle Bearbeitung, wie das setzen des *need-resched* Flags kann dies aber durchaus im IRQ Kontext durchgeführt werden.

9.6. Timer-Analyse

In den letzten Abschnitten wurden die Grundlagen für ein tieferes Verständnis der Linux-Kernel-Timer gelegt – nun wird der Fokus auf die praktische Anwendung gerichtet. Mithilfe von perf werden die timerrelevanten Events anhand eines konkreten Beispiels analysiert.

Damit das Beispiel verständlich bleibt, reduzieren wir die Aufnahme auf das Wesentliche. Der ausgeführte perf record mitsamt des Workloads ist in Auflistung 9.10 ersichtlich. Die relevanten Optionen sind folgende:

- Der Workload sowie die Aufzeichnung findet ausschließlich auf Core 8 statt. Eine Aufzeichnung auf allen Cores, Option `-a`, würde die aufgezeichnete Datenmenge deutlich erhöhen. Core 8 wurde gewählt da der Scheduler bei gegebner Last diesesn Core wenig Task zuweist. Es ist wichtig zu beachten, dass eine reine Aufzeichnung im Prozessmodus nicht möglich ist, da Timerinterrupts sowie SoftIRQ im eigenen Kontext ausgeführt werden.
- Es wird nicht das Shell Build-In Kommando sleep aufgerufen, sondern die Applikation sleep. Dies ist wichtig, da nur über diesen Weg eine CPU Affinität vergeben werden kann.
- Der Schlafzeitraum wird auf 10 ms beschränkt um die aufgezeichnete Datenmenge zu limitieren
- Es werden alle Timer sowie IRQ Events aufgezeichnet. Dabei ist anzumerken das IRQ events neben den reinen Hardware IRQ Events auch SoftIRQ beinhalten.

```
$ perf record -C 8 -e 'timer:*,irq:*' -- taskset -c 8 /bin/sleep 0.01
```

Auflistung 9.10: Erfassung von Timer- und Interrupt-Ereignissen auf CPU-Kern *8* während eines kurzen *sleep*-Aufrufs mittels *perf record*.

In Auflistung 9.11 wird die beispielhafte Ausgabe dargestellt. Da sich sehr viel Detailwissen hinter den jeweiligen Zeilen verbirgt, wird in den nächsten Absätzen jede relevante Zeile einzeln erläutert, Es ist zu beachten, dass die Ausgabe mit `--reltime` aufgerufen wurde. Der Timestamp in der zweiten Spalte ist damit relativ zum Beginn der Aufnahme. Des Weiteren wurde die Ausgabe modifiziert, um eine kompaktere Darstellung zu ermöglichen.

```
$ perf script --reltime
  sleep 0.000000:         hrtimer_init: hrtimer=0x00 clockid=CLOCK_REALTIME mode=0x1
  sleep 0.000000:        hrtimer_start: hrtimer=0x00 function=hrtimer_wakeup
                                          expires=6.455427759 softexpires=6.455377759
swapper 0.002627:       hrtimer_cancel: hrtimer=0x01
swapper 0.002629: hrtimer_expire_entry: hrtimer=0x01 now=6.448003690
                                          function=tick_sched_timer
swapper 0.002632:    irq:softirq_raise: vec=9 [action=RCU]
swapper 0.002642:    irq:softirq_raise: vec=7 [action=SCHED]
swapper 0.002644:  hrtimer_expire_exit: hrtimer=0x01
swapper 0.002645:        hrtimer_start: hrtimer=0x01 function=tick_sched_timer
                                          expires=6.452000000 softexpires=6.452000000
swapper 0.002648:    irq:softirq_entry: vec=7 [action=SCHED]
swapper 0.002661:     irq:softirq_exit: vec=7 [action=SCHED]
swapper 0.002662:    irq:softirq_entry: vec=9 [action=RCU]
swapper 0.002665:     irq:softirq_exit: vec=9 [action=RCU]
swapper 0.006683:       hrtimer_cancel: hrtimer=0x01
swapper 0.006687: hrtimer_expire_entry: hrtimer=0x01 now=6.452057161
                                          function=tick_sched_timer
swapper 0.006690:    irq:softirq_raise: vec=9 [action=RCU]
swapper 0.006700:    irq:softirq_raise: vec=7 [action=SCHED]
swapper 0.006701:  hrtimer_expire_exit: hrtimer=0x01
swapper 0.006702:        hrtimer_start: hrtimer=0x01 function=tick_sched_timer
                                          expires=6.456000000 softexpires=6.456000000
swapper 0.006706:    irq:softirq_entry: vec=7 [action=SCHED]
swapper 0.006716:     irq:softirq_exit: vec=7 [action=SCHED]
swapper 0.006717:    irq:softirq_entry: vec=9 [action=RCU]
swapper 0.006720:     irq:softirq_exit: vec=9 [action=RCU]
swapper 0.010101:       hrtimer_cancel: hrtimer=0x00
swapper 0.010102: hrtimer_expire_entry: hrtimer=0x00 now=6.455478395
                                          function=hrtimer_wakeup
swapper 0.010103:  hrtimer_expire_exit: hrtimer=0x00
```

Auflistung 9.11: Ausgabe von *perf script* mit relativer Zeit zeigt Ablauf von High-Resolution Timern und zugehörigen Softirqs während kurzer *sleep*-Phase.

- **Zeile 2:** Unmittelbar nach dem Start der Applikation `sleep` (0.000000) wird der High Resolution Timer initialisiert. Bei aktiviertem Systemcall Event Tracing wäre ersichtlich, dass sleep auf den Systemaufruf `sys_nanosleep` basiert. Die Implementierung von `sys_nanosleep` wiederum nutzt HRTimer, anstelle von Kernel Timern. Dies wird durch die Initialisierung des Timers über die Funktionen `hrtimer_init` und `hrtimer_start` ersichtlich. Das erste Argument von `hrtimer_init` ist die eindeutige Kennung, das „Handle“ des Timers, hier dargestellt als `0x00`. Dieser Wert ist ein 64-Bit-Wert, hier aus Gründen der Darstellung gekürzt. Die ID stellt einen einzigartigen Identifikator für den Timer dar. Weiterhin wird ersichtlich, dass als ClockID `CLOCK_REALTIME` verwendet wurde. Der Mode ist mit `0x1` festgelegt, was

den Relativen Modus definiert. Neben diesem existiert der Absolute Modus, und es gibt zudem die Möglichkeit, den Timer an eine CPU verschiedenartig zu pinnen[7].

- **Zeile 3-4:** Startet den Timer 0x00 unmittelbar. Als Callback Function wird `hrtimer_wakeup` aufgerufen. Diese Funktion ist die Standardfunktion, welche beim ablaufen eines Timers den jeweilige Task wieder Ablauffähig schaltet und den Scheduler informiert. Das Argument `expires` gibt an wann der Timer spätestens ausgeführt werden kann. `softexpires` gibt an wann der Timer frühstmöglichst expiren kann[8]. Die beiden Werte stehen im direkten Zusammenhang mit den sogenannten Slack welche auf auf Seite 614 ausführlich erklärt wird.
- **Zeile 5:** Im Kontext des swapper/idle Tasks beendet sich ein Timer mit der ID `0x01`. Die Initialisierung und start hat vor der Aufzeichnung begonnen, es sind daher keine Informationen bekannt welcher Task diesen Timer gestartet hat.
- **Zeile 6-7:** Timer `0x01` ist abgelaufen, das Timeout also erreicht. Sofern das passiert wird der registrierte Callback, siehe `hrtimer_start`, zwischen den Funktionen `hrtimer_expire_entry` und `hrtimer_expire_exit` ausgeführt. Als Argument wird die aktuelle Zeit als `now` übergeben, dies ist Hilfreich da die verwendete Uhr von perf oft eine andere Zeitquelle hat. Über das Argument now kann damit die Differenz zwischen angedachten und tatsächlichen Timerausführung berechnet werden. Als Callback wird die Funktion `tick_sched_timer` aufgerufen. Es ist am Namen bereits ersichtlich das es sich bei dem Timer wir der ID `0x01` um den regulären Timer Tick handeln wird.
- **Zeile 8-9:** Die Funktion `tick_sched_timer` markiert zwei SoftIRQ zur Ausführung, einmal für die RCU und das andere mal für die Scheduling Verarbeitung. Dies waren also zwei Aktionen, welche unter anderem von dem zyklischen Tick instanziert wurden.
- **Zeile 10:** Der Callback ist beendet, dies wird durch das Ende umklammernde expire Funktion angezeigt.
- **Zeile 11-12:** Direkt am Anschluss des Ablauf des zyklischen Timer Ticks wird ein neuer gestartet. Dieser liegt 4 ms in der Zukunft. Die 4 ms entsprechen exakt 1/250 Hz – genau dem `CONFIG_HZ` Wert. Anders als beim Userspace initierten Timer beträgt die Differenz zwischen expires und softexpires genau 0. Es soll also kein Slack verwendet werden – verständlich.

---

7 Für eine detailliertere Auflistung der verschiedenen Modi ist die Datei `hrtimer.h` zu konsolidieren

8 POSIX legt fest das ein Timer unter keine Umständen vorab ablaufen darf. Eine verspätete Ausführung – auch wenn sie noch so klein sein kann – liegt in der Natur der Dinge

- **Zeile 13-16:** Es kommen die in Zeile 8 und 9 markierten SoftIRQs zur Ausführung. Der RCU sowie der Scheduler bezogene SoftIRQ.
- **Zeile 17-28:** Nach 4,018 ms des letzen Events und 4,039 ms seit Re-Start vom Timer `0x01` kommt dieser zyklische Timer wieder zur Ausführung. Ab hier wiederholt sich die Ausführung
- **Zeile 29-32:** Läutet das Ende der Applikation ein. Nach 10.101 ms wird nachdem der Timer in Zeile 3 gestartet wurde, läuft dieser nun ab. In Zeit 30 kommt der Callback zur Ausführung. Dieser weckt den Prozess sleep und returniert damit aus dem Syscall `sys_nanosleep`.

Im Beispiel werden alle wesentlichen Funktionen für das Timermanagement beschrieben. Üblicherweise werden HRTimer eingesetzt, insbesondere wenn Userspace-Applikationen einen Timer nutzen, wie es beispielsweise bei nanosleep() der Fall ist. Im Kernel sind jedoch weiterhin die älteren Kernel Timer vorhanden. Die Tracepoints für diese sind nahezu identisch, die Unterscheidung ist durch den Namen möglich: `timer:hrtimer_init` versus `timer:timer_init`.

In ausführlichen Beispiel 9.11 auf Seite 609 wurden Timer-relevante sowie SoftIRQ Tracepoints dargestellt. In der Aufzeichnung sind keine Timer Interrupts vorhanden, da auf x86-64-Architekturen Timer Interrupts über IRQ-Vektoren umgesetzt werden. Timer IRQ Informationen sind somit über IRQ-Vector Tracepoints verfügbar sind. In Auflistung 9.12, Zeile 1, werden die korrespondierenden Entry- und Exit-Routinen des Timer IRQ Vectors aktiviert. Für Arm- und RISC-V-Architekturen wird der Timer als Standard-IRQ bereitgestellt, was in Zeile 10 bis 14 erkennbar ist.

```
1  $ perf record -e 'timer:*,irq:*,irq_vectors:local_timer_*' ...
2  [...]
3  irq_vectors:local_timer_entry: vector=236
4   hrtimer_cancel
5   hrtimer_expire_entry
6   hrtimer_expire_exit
7  irq_vectors:local_timer_exit: vector=236
8
9  $ perf record -e timer:\*,irq:\* ...
10 irq:irq_handler_entry: irq=13 name=arch_timer
11  timer:hrtimer_cancel
12  timer:hrtimer_expire_entry
13  timer:hrtimer_expire_exit
14 irq:irq_handler_exit: irq=13 ret=handled
```

Auflistung 9.12: Aufzeichnung Timer relevanter Tracepoints inklusive Time IRQ; Modifizierte Ausgabe

Nach der ausführlichen Beschreibung der wichtigsten timerrelevanten Tracepoints sollen nun die drei verbleibenden Timing Tracepoints erläutert werden. Diese lassen sich in zwei Gruppen unterteilen: einen Tracepoint für Informationen zum periodischen Timer Tick

und zwei Tracepoints für Interval Timer.

- **`tick_stop`**: wird aufgerufen, wenn der periodische Timer Tick angehalten wird oder werden sollte. Als erstes Argument wird der Parameter *success* dem Tracepoint übergeben. Ist dieser 1, so wurde der Timer Tick angehalten. Im Fall von `success=0` schlug dies Fehl. Als weiteres Argument gibt der Tracepoint die Ursache des Scheiterns aus.

  `dependency=SCHED` wird visualisiert wenn mehr als 1 ablauffähiger Prozess in der Runqueue zur Ausführung wartet und daher das abschalten des Timer Ticks nicht möglich ist.

  `dependency=POSIX_TIMER` es existieren laufende POSIX Timer welche ein abschalten verhindern.

  `dependency=PERF_EVENTS` es werden perf events aufgezeichnet, es entfällt ein Abschaltung.

  `dependency=CLOCK_UNSTABLE` die Clock Source ist instabil, sodass ein abschalten nicht möglich ist.

  `dependency=None` signalisiert keinen Fehlerfall, es wurde erfolgeich abgeschalten, was im success mit 1 enkodiert wurde.

- **`itimer_state`** wird aufgerufen sofern ein Interval Timer (*setitimer(2)*) gestartet oder beendet wurde. Im Gegensatz zu Kernel- und HRTimer sind keine dedizierte start oder cancel Tracepoints vorhanden. Aber: über das zweite Argument, `it_value` kann dies differenziert werden. Sollte der Wert 0 sind: cancel, in allen anderen Fällen wurde der Timer gestartet.

- **`itimer_expire`** ist der korrespondierende Tracepoint wenn der Timer abläuft. Es existiert keine Entry/Exit Klammer für itimer, kurz vor dem versenden des Signals an den korrespondierenen Prozess wird diese Funktion aufgerufen.

Als Abschluss dieses Abschnitts dient ein Beispiel, das den Umgang mit Timern veranschaulicht. Grundlage bildet ein Lastgenerator, der eine Reihe von Prozessen erzeugt, welche gezielt CPU-, I/O- und Netzwerklast erzeugen. Diese Applikationen laufen jeweils exakt 20 s. Parallel zur Ausführung wurden Timer-Events analysiert.

In Abbildung 9.3 auf der nächsten Seite sind alle abgelaufenen Timer in logarithmischer Skalierung dargestellt. Vor und nach dem Lasttest verläuft die Aktivität vergleichsweise ruhig[9].

[9] Dabei fällt auf, dass beispielsweise Kitty – ein Terminalemulator – kontinuierlich in festen Abständen Timer auslöst. Diese Muster bieten potenzielle Ansatzpunkte für Optimierungen. Es stellt sich die Frage, ob der jeweilige Timer tatsächlich regelmäßig ausgelöst werden muss. Solche wiederkehrenden, möglicherweise

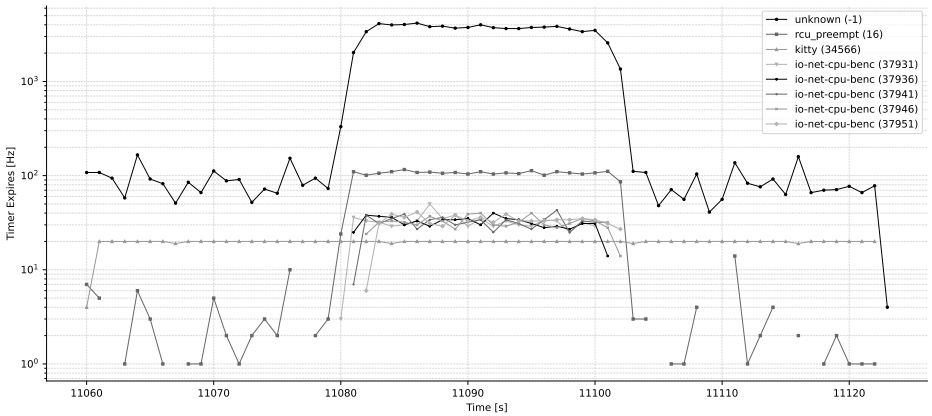


Abbildung 9.3.: Visualisierung Timer Events während Ausführung eines Netzwerk-Last-Generators; Aufgetragen auf logarithmische Skale.

Während der Testausführung steigt die Anzahl der Timer-Events deutlich an. Bei genauerer Analyse zeigt sich, dass einige dieser Timer direkt vom Lastgenerator gesetzt wurden. Der Großteil der Timer lässt sich jedoch keinem spezifischen Prozess zuordnen und erscheint als „unknown“ – hierbei handelt es sich um Kernel-Timer, die keinem bestimmten Task zugewiesen werden können.

Ein genauerer Blick auf die „unknown“ Kernel-Timer-Expires, insbesondere anhand des Felds `function` des zugehörigen Tracepoints, ermöglicht Rückschlüsse auf die beteiligten Kernel-Subsysteme. Die entsprechende Auswertung ist in Abbildung 9.4 auf der nächsten Seite dargestellt. Auf eine logarithmische Skalierung wurde diesmal verzichtet, um die Verteilung besser sichtbar zu machen.

Die Analyse zeigt, dass der Großteil der „unknown“-Kernel-Timer durch den Timer Tick verursacht wurde[10], konkret 71 %. Der zweitgrößte Anteil entfällt mit 23,9 % auf `napi_watchdog` – einen zentralen Timer für das Netzwerk- und Paketverarbeitungs-Subsystem. Beide, der periodische Timer Tick sowie das NAPI-Handling, sind direkte Folgen aktiver Task-Verarbeitung und Netzwerkaktivität.

---

unnötigen Timer-Events können das Erreichen tiefer Schlafzustände der CPU verhindern und somit die Energieeffizienz beeinträchtigen.

10 Die theoretisch maximale Anzahl an Timer Ticks ergibt sich zu 250, Hz × 20, Sekunden × 32, CPUs = 160, 000 reguläre Timer-Expires.

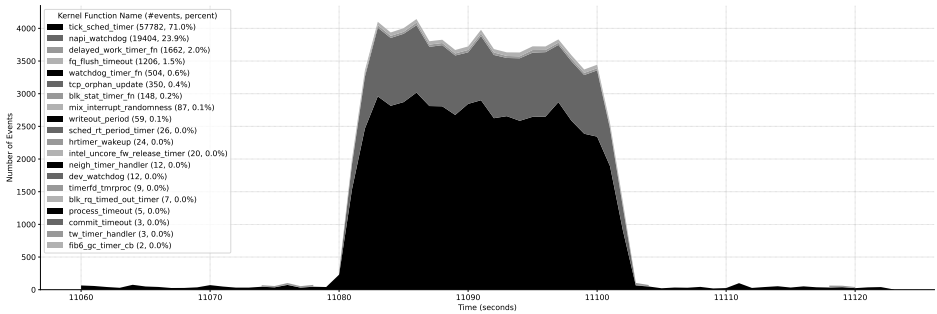


Abbildung 9.4.: Visualisierung Kernel Timer Events über Zeit während Ausführung eines Netzwerk-Last-Generators.

9.7. Task Timer Slack

Ein interessante Konfigurationsoption, das auf der Ebene einzelner Tasks individuell angewendet werden kann, ist der *Timer Slack*. Durch diesen kann die Auflösungsgranularität kommandierter Timer beeinflusst werden. Der Schalter zielt darauf ab, dass der Kernel weniger Unterbrechungen und längere Schlafphasen aufweist, indem Timer, die in einem ähnlichen zeitlichen Bereich liegen, zusammengefasst werden. Konfigurieren lässt sich dieser Wert für jeden Task über `/proc/<pid>/timerslack_ns`. Beim Lesen liefert diese Datei den aktuellen Slack, standardmäßig sind dies 50 µs. In Auflistung 9.13 auf der nächsten Seite sind exemplarisch einige Tasks abgebildet. Bis auf den Prozess `rtkit-daemon` (PID 1198), der mit 2,5 s konfiguriert wird, sind alle Prozesse ausnahmslos mit 50 µs konfiguriert.

Der aufmerksame Leser wird bemerken, dass der auf Seite 610 vorgestellte Tracepoint `hrtimer_start`, exakt die darunterliegenden Mechanismen implementiert um den Timer Slack abzubilden. Ein Task hat immer einen korrespondierenden Slack, wenn ein Timer gestartet wird, wird der kommandierte Schlafzeitpunkt (z.b. bei Aufruf von `nanosleep`), als `softexpire` gesetzt. Die Variable `expire` wird auf `softexpire + timerslack_ns` gesetzt. Oder anderes: frühstmöglich kann ein Timer mit den kommandierten Wert ablaufen (`softexpire`), spätmöglichst aber zu `expire` (`softexpire + timerslack_ns`). Ist der Wert von `timerslack_ns` 0, dann sind `softexpire` und `expire` identisch. Der Kernel wird damit angewiesen den Timer fristgerecht ablaufen zu lassen und ein verschieben zu vermeiden.

Der wesentliche Nachteil des Timer Slacks besteht darin, dass er für alle Timer eines Tasks denselben Wert konfiguriert. Insbesondere größere Programme verwenden jedoch eine

Vielzahl unterschiedlich priorisierter Timer. Einige davon könnten problemlos um mehrere Mikrosekunden verschoben werden, während andere keinerlei Verzögerung tolerieren. Dieses *Alles-oder-nichts*-Verhalten stellt die zentrale Schwäche des Schalters dar.

Ein weiterer Nachteil des Schalters besteht darin, dass der Slack bei Echtzeitprozessen keine Berücksichtigung findet. In solchen Fällen ist dies jedoch häufig auch nicht sinnvoll.

```
$ grep -rin . /proc/*/{timerslack_ns,comm} | sort -n | less
/proc/1198/comm:1:rtkit-daemon
/proc/1198/timerslack_ns:1:2500000000
/proc/1/comm:1:systemd
/proc/1/timerslack_ns:1:50000
/proc/2075/comm:1:gnome-shell
/proc/2075/timerslack_ns:1:50000
/proc/2093/comm:1:dbus-daemon
/proc/2093/timerslack_ns:1:50000
[...]
```

Auflistung 9.13: Übersicht aktiver Prozesse mit zugehörigen *comm*-Namen und eingestellten *timerslacka_ns*-Werten aus dem */proc*-Verzeichnis.

Auch wenn die Einsatzmöglichkeiten eingeschränkt sind, kann das Setzen des Slacks bei Prozessen, die andernfalls nicht modifizierbar wären, von Vorteil sein. Dabei ist jedoch unbedingt darauf zu achten, dass Applikations-kritische Intervall nicht verletzt werden. Wird ein Prozess beispielsweise regelmäßig im Abstand von 10 ms aufgerufen, darf der Slack nicht auf 100 ms gesetzt werden – der gesamte Programmablauf würde dadurch gestört. An diesem konkreten Beispiel zeigt sich ein weiteres grundlegende Dilemma des Schalters: Der Nutzer muss über präzise Kenntnisse der betreffenden Applikation verfügen, um negative Effekte sicher ausschließen zu können.

9.8. Manueller Timer Slack

Eine mächtigere, da zielgerichtere Lösung um Timer Systemglobal auszurichten, besteht darin, die Ausrichtung manuell vorzunehmen. Gezielt kann für jeden Timer der mögliche Slack angegeben werden. Zeitkritische Timer können weiterhin auf die exakte Zeit belassen werden. Gerade aber bei Timern wo eine Abweichung von 10 ms, 100 ms oder gar 250 ms vernachlässigbar wäre, sind ideale Kandidaten.

Leider bietet Linux oder POSIX keine Timerfunktion an, welche ähnliches leistet. In Auflistung 9.14 auf Seite 617 ist ein erster Prototyp für eine Funktion abgebildet. Den interessierten Leser wird dies als Übung überlassen. Als Tipp: der Timer darf nicht relativ gesetzt werden, wenn die Funktionalität über `clock_nanosleep` implementiert wird, ist das zweite Argument (flags) mit den Parameter `TIMER_ABSTIME` zu übergeben. Der Rest der

Funktion ist die Abfrage der aktuellen Zeit plus Module Berechnungen im in das Raster zu rücken.

Auch wenn der Mechanismus sehr leistungsfähig ist, weist er zwei entscheidende Nachteile auf: Zum einen setzt er voraus, dass alle beteiligten Prozesse auf ein gemeinsames Zeitraster ausgerichtet werden. Wenn beispielsweise Prozess *Foo* bei einer Schlafdauer von 10 ms auf 0,04 s aufgerundet wird, Prozess *Bar* hingegen auf 0,03,s abgerundet, kommt keine globale Synchronisation zustande. Es muss daher eine explizite Vereinbarung zwischen allen beteiligten Prozessen getroffen werden, wie das zeitliche Alignment erfolgen soll.

Der zweite Nachteil besteht darin, dass Eingriffe in die Applikationen erforderlich sind. In Fällen, in denen kein Quellcode verfügbar ist oder Änderungen aus Wartbarkeitsgründen nicht möglich sind, lässt sich dieses Vorgehen nicht umsetzen. In Embedded-Systemen hingegen, bei denen der Großteil der Software unter Kontrolle eines einzelnen Teams entwickelt wird, kann dieses Verfahren angewendet werden. Auch wenn nicht jede Komponente – wie etwa der *Init Service* – angepasst werden kann, lassen sich dennoch signifikante Vorteile für eigene Applikationen erzielen.

In Abbildung 9.5 ist der Effekt synchronisierter Timer exemplarisch dargestellt. Durch deren Einsatz ergeben sich sowohl intraprozessuale Optimierungen als auch systemweite Vorteile, sofern mehrere Applikationen einem gemeinsamen Zeitraster folgen.

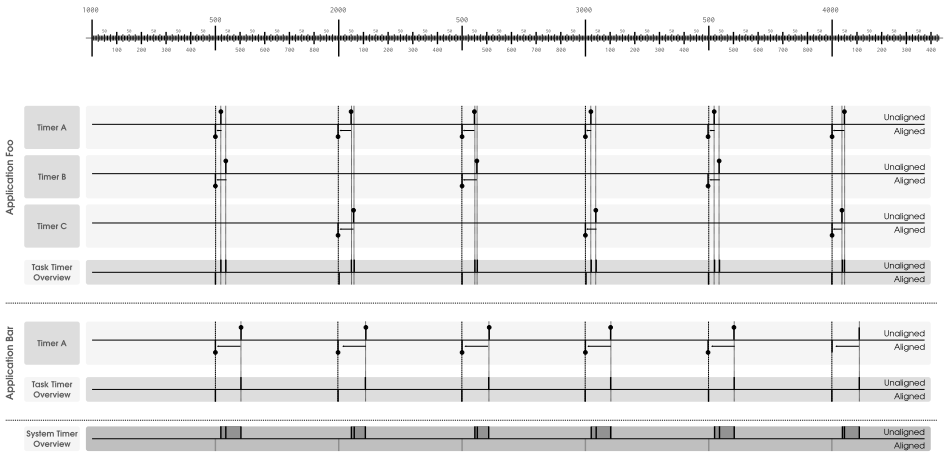


Abbildung 9.5.: Beispiel Manuelles Timer Alignment mit Übersicht mit Prozess-lokaler und System globaler Sicht.

Im Unterschied zu den sysfs `timerslack_ns` Mechanismus, ist dieser Mechanismus auch

Anwendbar auf Prozesse, welche mit einer Echtzeitpolicy wie `SCHED_RR` betrieben werden.

```
enum timer_slack {
  SLACK_VALUE_EXACT = 0,
  SLACK_VALUE_10NS,
  SLACK_VALUE_100NS,
  SLACK_VALUE_1US,
  SLACK_VALUE_10US,
  SLACK_VALUE_100US,
  SLACK_VALUE_1MS,
  SLACK_VALUE_10MS,
  SLACK_VALUE_100MS,
  SLACK_VALUE_1S
};

int nanosleep_aligned(struct timespec sleep_time, enum slack_value slack_value);
```

Auflistung 9.14: Definition möglicher *timer_slack*-Stufen sowie Deklaration einer potentiellen Funktion zur Ausrichtung von *nanosleep* mit einstellbarem Slack-Wert.

9.9. Abschließendes

Nachdem die grundlegenden Mechanismen und Werkzeuge zur Zeitmessung und Leistungsanalyse vorgestellt wurden, beleuchtet dieses abschließende Kapitel einige spezifische Zeitgeber, Zählregister sowie damit verbundene Konzepte im Detail. Der Fokus liegt auf der Vielfalt der über `perf` verfügbaren Zeit- und Zählquellen, ihren Eigenschaften und Anwendungsbeschränkungen. Darüber hinaus werden mit dem Virtual Dynamic Shared Object (VDSO) eine wichtige Optimierungstechnik für zeitrelevante Systemaufrufe sowie spezielle Alarm-Timer für das Wecken aus dem Suspend-Zustand erläutert, die für das Verständnis moderner Systemarchitekturen und deren Optimierung von Bedeutung sind.

9.9.1. Überblick Zeit und Zählregister

Im vorangegangenen Kapitel wurden mehrfach Zeitmessungen thematisiert. `Perf` stellt eine Vielzahl an Zeitquellen bereit, die zur Datenerfassung verwendet werden können. Abhängig vom Anwendungsfall kann der jeweils geeignete Zeitgeber ausgewählt werden. Eine Übersicht der verfügbaren Zeitquellen ist in Auflistung 9.15 dargestellt.

```
$ perf list | magic-perf-time-related-event-filtering
bus-cycles
task-clock
cpu-cycles
ref-cycles
cpu-clock
task-clock
msr/aperf/
```

```
msr/mperf/
msr/pperf/
msr/tsc/
```

Auflistung 9.15: Gefilterte *perf* Ereignisliste mit zeitbezogenen Events und MSR-Zählern zur Analyse von CPU-Zyklen und Taktverhalten.

An dieser Stelle sei angemerkt, dass nicht alle Events den Sampling Mode unterstützen; einige erlauben ausschließlich den Zugriff im Counting Mode. Dies sollte bei der Auswahl von Events berücksichtigt werden, insbesondere wenn eine freie Wahl der zu samplenden Events beabsichtigt ist. Die typische Fehlermeldung in einem solchen Fall ist in Auflistung 9.16 dargestellt.

```
$ perf record -C 0 -e msr/tsc/ -- sleep 1
Error:
The sys_perf_event_open() syscall returned with 22 (Invalid) for event (msr/tsc/).
/bin/dmesg | grep -i perf may provide additional information.
```

Auflistung 9.16: Fehlgeschlagener Versuch, *perf* mit dem Event *msr/tsc/* auf CPU-Kern *0* auszuführen; ungültiges Argument laut *sys_perf_event_open* Syscall.

In Auflistung 9.17 werden die Zähler *tsc*, *mperf*, *aperf* und *pperf* im Counting Mode erfasst. *tsc* zählt kontinuierlich alle Taktzyklen und dient als Zeitbasis. *mperf* erfasst Zyklen bei maximaler Frequenz, während *aperf* die tatsächlich aktiven Zyklen zählt. *pperf* bildet eine herstellerspezifische Metrik für die effektive Leistung ab. Das Verhältnis von *aperf* zu *mperf* erlaubt Rückschlüsse auf Frequenzskalierung und Energieeffizienz.

```
$ perf stat -C 0 -e msr/tsc/,msr/mperf/,msr/aperf/,msr/pperf/ -- sleep 1
     2.998.063.622      msr/tsc/
        19.165.290      msr/mperf/
         6.613.048      msr/aperf/
         6.546.463      msr/pperf/

       1,000996703 seconds time elapse
```

Auflistung 9.17: Messung verschiedener MSR-Zähler auf CPU-Kern *0* über eine Sekunde mit *perf stat*, darunter *tsc*, *mperf*, *aperf* und *pperf*.

Eine Beschreibung der zeitspezifischen Events ist in Tabelle 9.1 auf Seite 621 aufgeführt. Falls diese oder weitere Zeitquellen berücksichtigt werden müssen, ist es empfehlenswert, auch die jeweilige plattformspezifische Dokumentation heranzuziehen, da plattformspezifische Abweichungen erheblich sein können.

9.9.2. Virtual Dynamic Shared Object

Virtual Dynamic Shared Object (VDSO) ist eine spezielle, vom Kernel im Userspace bereitgestellte Bibliothek. Sie ermöglicht es Prozessen, bestimmte Systeminformationen

ohne einen teuren Systemaufruf (syscall) abzurufen. Damit stellt die VDSO eine Abkürzung für häufig genutzte Kernelinformationen dar. Zu den als VDSO bereitgestellten Systemaufrufen zählen unter anderem `gettimeofday(2)` und `clock_gettime(2)`[11].

Das VDSO wird vom Kernel so implementiert, dass das entsprechende Objekt direkt in den Adressraum des Prozesses eingeblendet wird. Das eingeblendete Objekt unterscheidet sich funktional kaum von regulären Bibliotheken, die durch den dynamischen Linker in den Speicher geladen werden – tatsächlich handelt es sich um ein reguläres ELF-Objekt, wie gleich gezeigt.

VDSO stellt eine gezielte Optimierung dar, um die Kosten für häufige Systemaufrufe zu senken. Zwar wurden Systemcall-Kosten durch moderne CPU- und Betriebssystemoptimierungen bereits deutlich reduziert, dennoch bietet der Ersatz systemcall-basierter Funktionen durch direkte Funktionsaufrufe mit geringerem Overhead einen spürbaren Vorteil bei hoher Aufruffrequenz. Wird beispielsweise `clock_gettime()` über einen klassischen Systemcall aufgerufen, kann dies etwa 100 ns pro Aufruf kosten. Die VDSO-Variante reduziert diese Zeit auf nur etwa 20 ns bis 30 ns pro Funktionsaufruf.

In Auflistung 9.18 wird gezeigt, wie die VDSO extrahiert werden kann. Zusätzlich werden alle exportierten Symbole, insbesondere die enthaltenen Funktionen, sichtbar gemacht.

```
$ cat /proc/1/maps | grep vdso
7fb4488af000-7fb4488b1000 r-xp 00000000 00:00 0                      [vdso]

$ COUNT=$((0x7fb4488b1000-0x7fb4488af000))
$ dd if=/proc/1/mem bs=1 skip=$((0x7fb4488af000)) count=$COUNT of=vdso.elf
8192+0 records in
8192+0 records out
8192 bytes (8.2 kB, 8.0 KiB) copied, 0.00769258 s, 1.1 MB/s

$ eu-readelf -s vdso.elf
Symbol table [ 3] '.dynsym' contains 15 entries:
Num:    Value   Size Type    Bind   Vis       Ndx Name
  0: 00000000      0 NOTYPE  LOCAL  DEFAULT UNDEF
  1: 00000b00    896 FUNC    WEAK   DEFAULT    12 clock_gettime@@LINUX_2.6
  2: 00000800    715 FUNC    GLOBAL DEFAULT    12 __vdso_gettimeofday@@LINUX_2.6
  3: 00000e80     99 FUNC    WEAK   DEFAULT    12 clock_getres@@LINUX_2.6
  4: 00000e80     99 FUNC    GLOBAL DEFAULT    12 __vdso_clock_getres@@LINUX_2.6
  5: 00000800    715 FUNC    WEAK   DEFAULT    12 gettimeofday@@LINUX_2.6
  6: 00000ad0     46 FUNC    GLOBAL DEFAULT    12 __vdso_time@@LINUX_2.6
  7: 00001460    161 FUNC    GLOBAL DEFAULT    12 __vdso_sgx_enter_enclave@@LINUX_2.6
  8: 00000f20    954 FUNC    GLOBAL DEFAULT    12 __vdso_getrandom@@LINUX_2.6
  9: 00000ad0     46 FUNC    WEAK   DEFAULT    12 time@@LINUX_2.6
 10: 00000b00    896 FUNC    GLOBAL DEFAULT    12 __vdso_clock_gettime@@LINUX_2.6
 11: 00000000      0 OBJECT  GLOBAL DEFAULT   ABS LINUX_2.6@@LINUX_2.6
 12: 00000ef0     42 FUNC    GLOBAL DEFAULT    12 __vdso_getcpu@@LINUX_2.6
 13: 00000ef0     42 FUNC    WEAK   DEFAULT    12 getcpu@@LINUX_2.6
 14: 00000f20    954 FUNC    WEAK   DEFAULT    12 getrandom@@LINUX_2.6
```

[11] Eine geeignete Stelle zur Erläuterung der VDSO ließ sich nur schwer im Buch finden. Im Kontext der Zeitmessung erscheint sie jedoch am sinnvollsten, da insbesondere zeitbezogene Kernel-Funktionen von dieser Technik profitieren.

Auflistung 9.18: Extraktion und Analyse der *vdso* aus dem Speicherabbild von *PID 1*; Symboltabelle zeigt exportierte Zeit- und Zufallsfunktionen wie *clock_gettime* und *getrandom*.

Hinweis 9.6 In der Vergangenheit wurde diskutiert, auch `getpid(2)` über das VDSO bereitzustellen. In Randfällen zeigte sich jedoch, dass die Umsetzung komplexer ist als zunächst angenommen. Zudem wurde das Argument vorgebracht, dass eine Applikation, die `getpid()` derart häufig aufruft, stattdessen besser selbst optimiert werden sollte.

9.9.3. Alarm Timer

An dieser Stelle sollen der Vollständigkeit halber die beiden Timer `CLOCK_REALTIME_ALARM` und `CLOCK_BOOTTIME_ALARM` vorgestellt werden. Diese sind mit ihren Gegenstücken ohne das Suffix `_ALARM` vergleichbar, mit dem entscheidenden Unterschied, dass bei deren Auslösung ein System aus dem Suspend-Zustand geweckt wird. Um zu verhindern, dass jede Applikation derart drastische Auswirkungen auf das Systemverhalten haben kann, ist dies an die Capability `CAP_WAKE_ALARM` gebunden. Es ist beinahe magisch, wie ein mit `systemctl suspend` in den Suspend-Zustand versetztes System nach <n> Sekunden mit einem `ALARM` Timer wieder zum Leben erweckt wird.

Für Systemdesigner, die ein energieeffizientes Gesamtsystem gestalten möchten, bietet die Kombination eines aggressiven Suspendierungsmechanismus mit `CLOCK_REALTIME_ALARM` und `CLOCK_BOOTTIME_ALARM` die Möglichkeit, ein höchst effizientes System zu entwickeln. Android setzt beispielsweise intensiv auf `ALARM` Timer.

Event	Typ	Beschreibung	Frequenzbezug	Granularität
`bus-cycles` (`PERF_COUNT_HW_BUS_CYCLES`)	HW	Zählt Buszyklen, wenn die Prozessor-Busschnittstelle aktiv ist. Hängt von der Workload-Interaktion mit Speicher/IO ab. Hohe Werte können auf Datenübertragungs-Engpässe hindeuten.	Hängt von Busaktivität ab	System / Prozessor-Interface
`task-clock`	SW	Misst die gesamte CPU-Zeit (in Millisekunden), die von einem spezifischen Task/Prozess genutzt wurde (Summe über alle CPUs).	Zeitmessung	Pro Task / Prozess
`cpu-cycles` (`PERF_COUNT_HW_CPU_CYCLES`)	HW	Zählt die tatsächlichen Taktzyklen, die von einer CPU/Core ausgeführt wurden. Die Zählrate ändert sich mit der CPU-Frequenz (P-States, Turbo Boost) und stoppt in Halt-Zuständen (C-States > C0).	Variabel (Core-Takt)	Pro CPU / Core
`cpu-clock`	SW	Misst die verstrichene Zeit (in Millisekunden), während der eine CPU aktiv war (nicht im Idle-Zustand). Summiert über alle Prozesse auf dieser CPU.	Zeitmessung	Pro CPU
`ref-cycles` (`PERF_COUNT_HW_REF_CYCLES`)	HW	Zählt Referenztaktzyklen mit einer konstanten Frequenz (oft TSC-basiert). Nützlich für Zeitmessungen, da unbeeinflusst von CPU-Frequenzänderungen (P-States). Kann in tiefen Schlafzuständen (C-States) anhalten.	Konstant (Referenztakt)	Pro CPU / Core
`msr/aperf/`	MSR	Zählt über ein MSR die tatsächlich ausgeführten Core-Taktzyklen (APERF - Actual Performance). Berücksichtigt die variable Taktfrequenz.	Variabel (Core-Takt)	Pro Core
`msr/mperf/`	MSR	Zählt über ein MSR die Taktzyklen mit der Referenz-Frequenz (MPERF - Maximum Performance / Reference). Läuft mit konstanter Rate, während der Core aktiv ist. Verhältnis APERF/MPERF ergibt effektive Frequenz.	Konstant (Referenztakt)	Pro Core
`msr/tsc/`	MSR	Liest den Zeitstempelzähler (TSC) über ein MSR. Ein hochauflösender Zähler, der (meistens) mit konstanter Frequenz läuft.	Konstant (TSC-Takt)	Pro CPU / Core
`CPU_CLK_UNHALTED.REF_TSC`	HW	Zählt Referenzzyklen (TSC-Takt, konstante Frequenz), wenn der Core nicht in einem Halt-Zustand (C-State > C0) ist. Nicht beeinflusst durch Frequenzänderungen (P-States, Turbo Boost). Nützlich zur Messung der aktiven Zeit bei konstanter Rate.	Konstant (TSC-Takt, wenn aktiv)	Pro Core
`CPU_CLK_UNHALTED.THREAD`	HW	Zählt die tatsächlichen Core-Taktzyklen, wenn der Hardware-Thread (logischer Prozessor) nicht angehalten (C-State > C0) ist. Beeinflusst durch Frequenzänderungen (P-States, Turbo Boost).	Variabel (Core-Takt, wenn aktiv)	Pro Hardware-Thread
`CPU_CLK_UNHALTED.CORE`	HW	Zählt die tatsächlichen Core-Taktzyklen, wenn der gesamte Core nicht angehalten ist (mindestens ein Thread aktiv, C-State > C0). Beeinflusst durch Frequenzänderungen (P-States, Turbo Boost).	Variabel (Core-Takt, wenn aktiv)	Pro Core
`REF_CPU_CYCLES`	HW (Alias)	Generischer Name (oft Alias für `ref-cycles` oder `CPU_CLK_UNHALTED.REF_TSC`). Zählt Referenztaktzyklen mit konstanter Frequenz, wenn der Prozessor nicht im Halt-Zustand ist. Unbeeinflusst von dynamischer Frequenzskalierung.	Konstant (Referenztakt, wenn aktiv)	Pro CPU / Core

Tabelle 9.1.: Typ, Beschreibung, Frequenzbezug und Granularität von ausgewählten Perf verfügbaren Time Events.