Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 95

Kern-Technik

Weit über 100 Kernelthreads treiben ihr Wesen auf jedem normalen Linux-System. Das Gewusel wirft Fragen auf, die zu beantworten sich diese Kern-Technik zur Aufgabe macht. Um das Credo gleich vorwegzunehmen: Masse ist hier ausnahmsweise Klasse. Eva-Katharina Kunst, Jürgen Quade

© pssdesign1, Fotolia

Mancher Desktop-Linux-Benutzer, der einen Blick auf die vollständige Liste der Rechenprozesse riskiert, erschrickt: Direkt nach dem Reboot tummeln sich bereits weit über 200 Tasks mit zum Teil kryptischen Namen (siehe Kasten „Taskliste anzeigen" und Abbildung 1). Kein Wunder, dass immer mehr User in Internetforen besorgt nachfragen, ob wirklich jeder Job nötig ist oder einige in Wahrheit Ungeziefer sind.

Bei genauerem Hinsehen entpuppt sich reichlich die Hälfte der Jobs als Kernelthreads. Deren Anzahl steigt zudem von Kernelversion zu Kernelversion, was auf der einen Seite die zunehmende Komplexität des Betriebssystemkerns selbst reflektiert und auf der anderen Seite auch die stetigen Fortschritte bei der Modularisierung.

Ein Kernelthread unterscheidet sich von einem gewöhnlichen Thread im Userspace dadurch, dass er Ressourcen – besonders Speicherbereiche – nur im Kernel beansprucht. Folglich ist er auch nicht mit einer Programmdatei assoziiert, die den Thread-Code enthält. Ansonsten bestimmt der Scheduler die Abarbeitungszeiten des Kernelthread-Codes wie die eines Applikationsthread. Der Scheduler wird jedoch einen gerade laufenden Kernelthread nicht grundlos unterbrechen, denn der gibt hoffentlich selbst regelmäßig die CPU frei.

Einfluss auf den Abarbeitungszeitpunkt eines Kernelthread nimmt der Kernelprogrammierer über die bekannten Methoden der Priorisierung oder der Auswahl des Schedulingverfahrens. Wer jetzt Angst um die Leistungsfähigkeit des Systems bekommt: Die meisten Kernelthreads befinden sich fast immer im Schlafzustand und konsumieren damit nur Hauptspeicher, aber keine CPU-Zeit.

Erben von der Verwandtschaft

An den niedrigen PIDs, den Prozess-Identifikationsnummern der Threads, ist erkennbar, dass Linux diese beim Hochfahren erzeugt. Da Unix-Systeme ein hie-

Taskliste anzeigen

Ubuntu-User können die »Systemüberwachung« starten und bekommen über »Prozesse« die Taskliste zu Gesicht (Abbildung 2). Im Tabellenkopf sowie in den Tabellenzeilen konfigurieren sie mit der rechten Maustaste zusätzliche Tabellenspalten und ändern die Prioritäten einzelner Tasks.

Auf der Konsole helfen die Kommandos »ps«, »top« und »htop« weiter. »top« und »htop« stellen die aktivsten Jobs oben an, was es erleichtert, Rechenzeitfresser zu identifizieren. »htop« ist extra zu installieren, bietet aber die umfangreichsten Möglichkeiten. Insbesondere blendet es auf einfache Weise die oft verwirrenden Kernelthreads aus (Abbildung 3).

Das Tool »ps« mit seinen überaus vielen Parametern lässt die Wahl zwischen BSD- und klassischem Unix-Style: Im BSD-Modus leitet ein Minuszeichen die Optionen ein. Beispielsweise zeigt »ps -ef« alle Prozesse, »ps -cef« zusätzlich die Prioritäten. Im Unix-Modus informiert »ps axwu« über alle Tasks.

Bei aller Fülle an Parametern fehlt »ps« eine einfache Möglichkeit, die für viele Anwender verwirrenden Kernelthreads auszublenden. Die sind an zwei Kriterien erkennbar: Erstens stammen in leidlich aktuellen Versionen alle Kernelthreads vom »kthreadd« mit der PID 2 ab, zweitens ist das Codesegment mit keiner (Programm-)Datei assoziiert. (Letzteres ist aber auch bei Zombie-Prozessen der Fall.) Dieses Wissen in eine Kommandozeile gegossen ergibt

```
ps -o pid,ppid,pri,comm,flags,%cpu,sz,%mem?
 --ppid 2 -N
```

und produziert die Taskliste ohne Kernelthreads. Ungleich mehr Komfort bietet »htop«, das Kernelthreads einfach per [Shift]+[K] ein- oder ausblendet.

rarchisches Prozessmodell implementieren, bei denen ein Thread grundsätzlich von einem Elternthread abstammt, ist der Urahn sämtlicher Kernelthreads der »kthreadd«.

Die Aufgabe des Kthreadd besteht nur darin, bei Bedarf und im Auftrag anderer Kernelkomponenten Kernelthreads zu erzeugen und ihnen eine eindeutige Erbmasse mitzugeben: Die erzeugten Threads sollen keinerlei Ressourcen im Userspace belegen und keine Dateien öffnen. Standardmäßig sind alle Signale blockiert, und auf einer Mehrkernmaschine darf jeder Core den neuen Thread abarbeiten. Außerdem ist jeder neue Kernelthread als Kind des Kthreadd ganz normal priorisiert.

Der Kernel baut den Kthreadd damit als Stammvater aller Kernelthreads direkt nach dem Init-Prozess zusammen und aktiviert ihn. Folglich besitzt er auch die PID 2, die zugleich die Parent PID (PPID) jener Tasks bildet, die er erzeugt hat. Die PID 3 ist ebenfalls einem Kernelthread vorbehalten, dem ersten von mehreren »ksoftirqd«. Er wird aktiv, wenn mehr Soft-IRQs auftreten, als das System sequenziell verarbeiten kann.

Ein Blick auf das Ebenen- und Kontextmodell von Linux verdeutlicht die Problematik (siehe Kasten „Ebenen- und Kontextmodell des Kernels“).

```
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0 120088  6160 ?        Ss   08:51   0:02 /sbin/init spla
sh
root           2  0.0  0.0      0     0 ?        S    08:51   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    08:51   0:00 [ksoftirqd/0]
root           5  0.0  0.0      0     0 ?        S<   08:51   0:00 [kworker/0:0H]
root           7  0.0  0.0      0     0 ?        S    08:51   0:10 [rcu_sched]
root           8  0.0  0.0      0     0 ?        S    08:51   0:00 [rcu_bh]
root           9  0.0  0.0      0     0 ?        S    08:51   0:00 [migration/0]
root          10  0.0  0.0      0     0 ?        S    08:51   0:00 [watchdog/0]
root          11  0.0  0.0      0     0 ?        S    08:51   0:00 [watchdog/1]
root          12  0.0  0.0      0     0 ?        S    08:51   0:00 [migration/1]
root          13  0.0  0.0      0     0 ?        S    08:51   0:00 [ksoftirqd/1]
root          15  0.0  0.0      0     0 ?        S<   08:51   0:00 [kworker/1:0H]
root          16  0.0  0.0      0     0 ?        S    08:51   0:00 [watchdog/2]
root          17  0.0  0.0      0     0 ?        S    08:51   0:00 [migration/2]
root          18  0.0  0.0      0     0 ?        S    08:51   0:00 [ksoftirqd/2]
root          20  0.0  0.0      0     0 ?        S<   08:51   0:00 [kworker/2:0H]
root          21  0.0  0.0      0     0 ?        S    08:51   0:00 [watchdog/3]
root          22  0.0  0.0      0     0 ?        S    08:51   0:00 [migration/3]
root          23  0.0  0.0      0     0 ?        S    08:51   0:00 [ksoftirqd/3]
root          25  0.0  0.0      0     0 ?        S<   08:51   0:00 [kworker/3:0H]
root          26  0.0  0.0      0     0 ?        S    08:51   0:00 [kdevtmpfs]
--Mehr--
```

Abbildung 1: Weit über 100 Kernelthreads laufen in modernen Linux-Distributionen und verwirren die User.

Soft-IRQs in der unteren Hälfte

Seit seinen Urzeiten realisiert Linux etwas komplexere Interrupt-Service-Routinen in zwei Teilen: Der so genannte Hard-IRQ erledigt die zeitkritischen Aufgaben, im anschließenden zweiten Teil – früher Bottom Half (BH) genannt – sind die zeitintensiven dran.

Den zeitkritische Teil gestaltet der Programmierer zeitlich möglichst kurz. In Linux sind die Bottom Halves klassisch als so genannte Soft-IRQs ausgeprägt, deren Bearbeitung direkt nach den Hard-IRQs bei zurückgesetzter Interruptsperre beginnt. Hard-Interrupts dürfen Soft-IRQs also jederzeit unterbrechen.

Manchmal werden Soft-IRQs schneller aktiviert, als sie das System bearbeiten kann. So eine Heavy-Load-Sitation liegt vor, wenn beispielsweise ein Hard-IRQ den aktiven Soft-IRQ unterbricht und der Hard-IRQ seinerseits den gleichen Soft-IRQ erneut triggert.

In diesem Fall bekommt der Kernelthread »softirqd« die Abarbeitung der Soft-IRQ-Funktion übergeben. Wie häufig eine solche Heavy-Load-Situation auftritt, ist übrigens an der verbrauchten CPU-Zeit des »softirqd« erkennbar. Der »softirqd« ist mehrfach vorhanden – für jeden CPU-Kern ein Mal. Die Geschwister-Threads starten erst mit der Aktivierung des jeweiligen CPU-Kerns.

Widersprüchliche Namensgebung

Die Bottom Halves anstatt in Form von Soft-IRQs als Kernelthreads abarbeiten

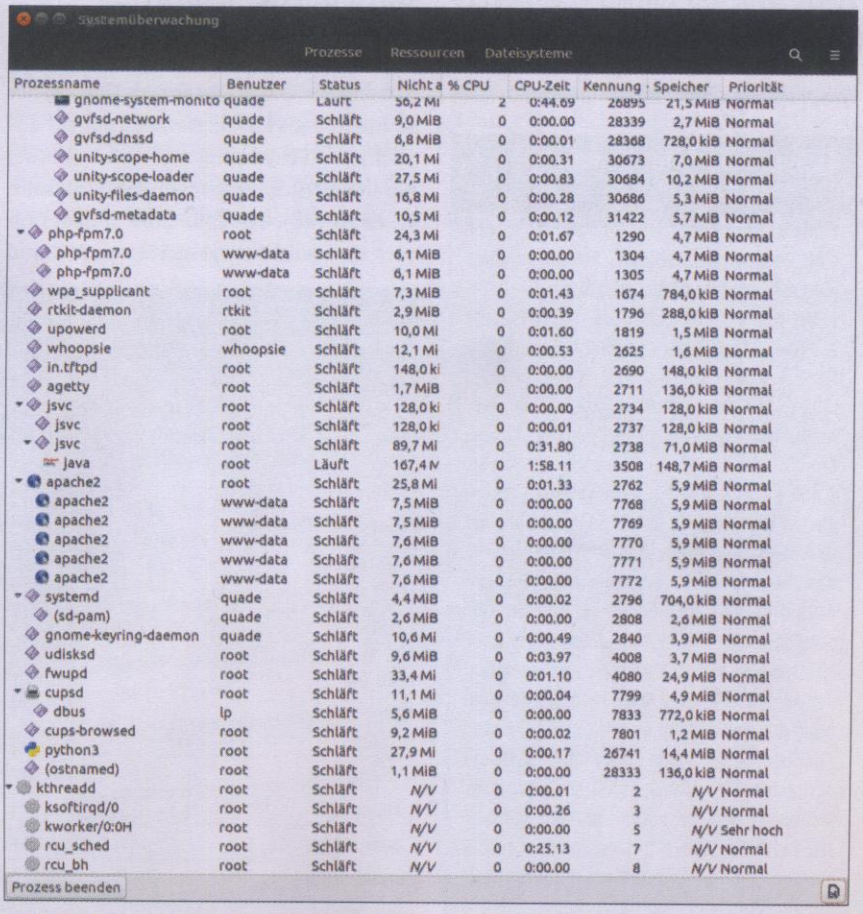

Abbildung 2: Ubuntu zeigt die Liste der Tasks in seiner »Systemüberwachung« an.

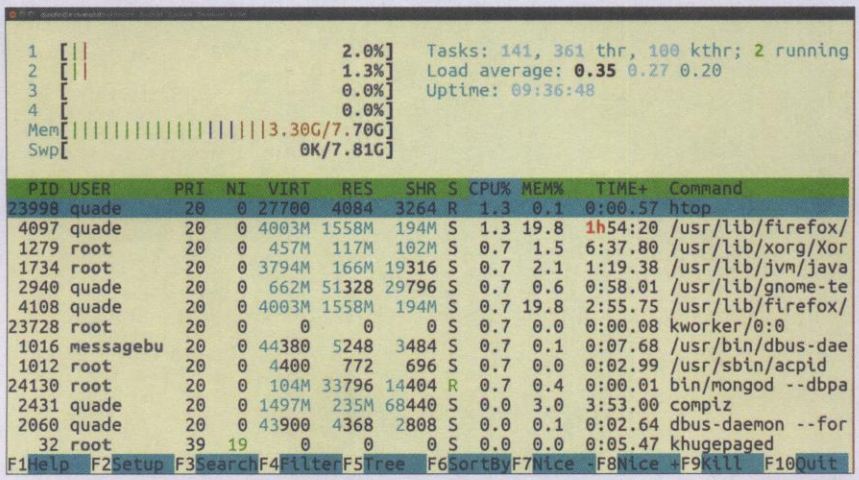

Abbildung 3: Bequem, informativ und hilfreich: Der Tasklisten-Anzeiger »htop«.

zu lassen, ist eine zukunftsweisende und gute Idee. Per Bootparameter ist das sogar für den gesamten Kernel durch den Systemarchitekten einstellbar.

Hauptnachteil dieser Interruptthreads – vom Wort her eigentlich ein Widerspruch – ist eine Effizienzeinbuße, denn immerhin muss bei jedem Interrupt der Scheduler tätig werden. Sehr vorteilhaft aber ist die mit den Kernelthreads verbundene Möglichkeit, den Threads Prioritäten zu geben und damit das Realzeit-Verhalten des Systems zu verbessern. Und mit der Ausprägung eines Interruptthread lässt sich – anders als früher – ein Interrupt sogar schlafen legen.

Im Prozesslisting tauchen die vordefinierten Interruptthreads unter dem Namen »irq«, Schrägstrich, Nummer, Bindestrich, Name auf, zum Beispiel als »irq/16-mmc0«. In diesem Fall repräsentiert der Name die Interruptnummer plus – soweit bekannt – deren Bedeutung.

Gute Sache: Migration verdrängt

Ein anderer Dauergast in den Top Ten der Kernelthreads ist »migration«, der für das Multicore-Scheduling im Linux-Kernel verantwortlich zeichnet (siehe Tabelle 1). Er ist für jeden CPU-Core einmal präsent und wird in regelmäßigem Abstand aktiv oder auch, wenn einer der Systemcalls »exit()«, »sleep()« oder »clone()« aufgerufen wird. Stellt »migration« eine ungleichmäßige Lastverteilung zwischen den CPU-Kernen fest, schiebt er Jobs von einem Kern zum nächsten. Details zu dem komplexen Vorgang verrät [1].

Wachhund als Teil des Gesundheitswesens

Neben dem »softirqd« kümmern sich drei weitere Thread-Arten um den Gesundheitszustand des Kernels: »watchdog«, »oom_reaper« und »khungtask«. Die Watchdog-Infrastruktur überwacht das System als Ganzes auf Lebendigkeit. Sie besteht aus einem Kernelthread pro CPU-Kern mit Namen »watchdog« und dem Kernelthread »watchdogd«.

Der Kernelthread jedes CPU-Kerns mit der höchsten Priorität wird – falls das nicht umkonfiguriert oder abgeschaltet ist – alle 4 Sekunden aktiv und setzt dann einen Zeitstempel zurück. Versäumt die Watchdog-Infrastruktur dies, schreibt ein höher priorisierter Soft-IRQ eine Auflistung der wichtigsten Systemparameter ins Syslog, um dem Admin eine Möglichkeit zu geben, die Ursache der Störung zu evaluieren. Ein solcher Softlockup eines einzelnen CPU-Kerns ist dann in anderen Kernelthreads zu suchen.

Der »watchdogd« ist Teil des Watchdog-Gerätetreibers im Linux-Kernel, der einen Software-Watchdog realisiert. Der wach

Ebenen- und Kontextmodell des Kernels

Kontext bezeichnet die Umgebung, in der eine Codesequenz abläuft. Sie entscheidet über die Funktionalität, auf welche die Codesequenz zurückgreifen kann. Außerdem legt der Kontext fest, wie kritische Abschnitte zu sichern sind.

Der Linux-Kernel kennt vier Ebenen und vier Kontexte (Abbildung 4). Kernelthreads laufen auf der Kernel-Ebene im Kernelkontext. Damit können sie sich zwar schlafen legen, aber nicht sinnvoll Daten zwischen Kernel und Userland austauschen.

Von den Applikationen aufgerufene Systemcalls arbeitet Linux ebenfalls auf der Kernel-Ebene ab. Sie laufen im Prozess- oder Userkontext und dürfen nicht nur schlafen, sondern auch Daten zwischen Kernel und Userland transferieren.

Für Codesequenzen im Interruptkontext sind sowohl der Schlafen-Zustand als auch jeder Datenaustausch zwischen Userland und Kernel tabu. Das gilt für Hard- und Soft-IRQs gleichermaßen. Zu letzteren gehören beispielsweise Timer oder Tasklets.

Tabelle 1: Ausgewählte Kernelthreads

Threadname	Aufgabe
kthreadd	Startet Kernelthreads mit eindeutiger Erbmasse
softirqd	Bearbeitet verstärkt auftretende Soft-IRQs
irq/*Interruptnummer-Bedeutung*	Interruptthreads
kworker	Kernelthread für diverse allgemeine Aufgaben
bioset	Schützt vor Deadlocks bei blockorientierter Ein- und Ausgabe
migration	Multicore-Scheduling – verteilt die Last zwischen den CPU-Kernen
rcu_sched	Gibt nicht mehr benötigte Synchronisationsobjekte von Kernelthreads frei (Read Copy Update)
rcu_bh	Gibt nicht mehr benötigte Synchronisationsobjekte von Soft-IRQs frei (Read Copy Update)
cpuhp	Integriert und initialisiert CPU-Kerne (CPU-Hotplugging)
kdevtmpfs	Legt Gerätedateien an (ähnlich »udevd«)
khungtask	Watchdog zum Identifizieren von geblockten CPU-Kernen
oom_reaper	Gibt Speicherseiten in hängenden Systemen frei
watchdog	Systemüberwachung auf Lebendigkeit einzelner CPU-Kerne
watchdogd	Systemüberwachung auf Lebendigkeit des Gesamtsystems

nicht nur über einen einzelnen CPU-Core, sondern über das gesamte System.

Task-Abschneider

Der Kernelthread »Oom_reaper« firmiert ebenfalls als Retter aus Extremsituationen. Der Out-of-Memory-Abschneider steht seit Kernel 4.6 bereit, um ein wegen Speichermangel eingefrorenes Linux wieder lauffähig zu bekommen, wenn Linux-Standardmaßnahmen nicht gegriffen haben.

Ist das Betriebssystem äußerlich sichtbar wegen Speichermangels über einen längeren Zeitraum festgefahren, wird aber zunächst der nicht als Kernelthread ausgeprägte OOM Killer [2] aktiv. Der Out-of-Memory-Beauftragte sendet jenen Jobs eine Aufforderung zum Selbstmord, die augenscheinlich für den hohen Speicherverbrauch verantwortlich sind.

Unglücklicherweise ist nicht jeder Marodeur in der Lage, der suizidalen Aufforderung Folge zu leisten, beispielsweise weil er wegen eines Lock schläft, das gerade ein anderer Thread hält, der seinerseits aber wegen der Speicherknappheit nicht weiterkommt.

Genau hier springt der eingangs erwähnte »oom_reaper« in die Bresche. Die Kernelentwickler Mel Gorman und Oleg Nesterov haben nämlich identifiziert, dass bei einem Job, der per »SIGKILL« zum Selbstmord aufgefordert ist, sich Teile des Hauptspeichers (so genannte Anonymous Pages) schon ante mortem freigeben lassen. Und eben dies soll »oom_reaper« tun [3].

Sekundenschlaf

Wer in seinem Syslog die Meldung »INFO: task Name blocked for more than 120 seconds« findet, macht erste Bekanntschaft mit dem »khungtask«-Kernelthread. Der überprüft alle Prozesse im System daraufhin, ob sie für mehr als 120 Sekunden im Schlafzustand verweilen, aus dem sie nur aufweckbar sind, wenn das Ende der Schlafen-Bedingung erreicht ist (Taskzustand »TASK_UNINTERRUPTIBLE«).

Das passiert häufiger bei Zugriffen auf Hintergrundspeicher, muss aber nicht jedes Mal ein Fehler sein. Entdeckt der Kernelthread eine solche Task, erzeugt er die genannte Meldung. Die Überwachungszeit von 120 Sekunden kann der Systemverwalter übrigens über das Sys-Filesystem (»/proc/sys/kernel/hung_task_timeout_secs«) ebenso anpassen wie auch das weitere Verhalten (»hung_task_panic«).

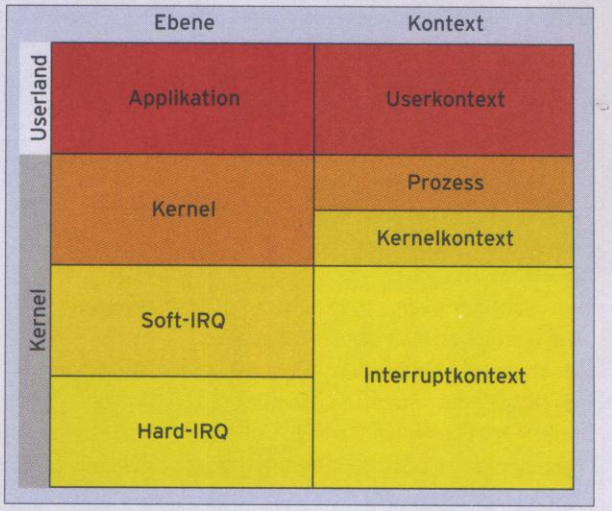

Abbildung 4: Ebenen- und Kontextmodell: Je nach Kontext gibt es für den Programmierer Einschränkungen.

So kann der Thread beispielsweise eine Kernelpanic ausgeben, wenn er eine blockierte Task entdeckt. Wer seine Einstellung dauerhaft macht will, tut das in der Datei »/etc/sysctl.conf«.

Räumkommando

Von zentraler Bedeutung für Linux sind die Kernelthreads, deren Namen mit »rcu« beginnen. RCU steht für Read Copy Update, eine sehr pfiffigen Technik zum Schutz kritischer Abschnitte, die den gleichzeitigen Zugriff von mehreren lesenden und einem schreibenden Job erlaubt [4]. Spannenderweise erzeugt der

lesende Zugriff bei dieser Technik in den meisten Fällen keinerlei Overhead und der Zugriff birgt nicht einmal die Gefahr von Deadlocks. Die Grundidee hinter dem US-Software-patentierten Verfahren: Ein schreibender Job nimmt die Änderungen an einer Kopie vor und tauscht danach die Zugriffszeiger der lesenden Codesequenzen aus.

Die Kernelthreads »rcu_sched« und »rcu_bh« dienen jetzt dazu, die nach dem Tausch zurückbleibenden alten Kopien der Objekte zu einem geeigneten Zeitpunkt – ähnlich einem Garbage Collector – aus dem Speicher zu entfernen. Dabei übernimmt der Kernelthread »rcu_sched« diese Arbeit für Codesequenzen, die im Kernel- oder Prozesskontext arbeiten, und »rcu_bh« für Codesequenzen im Interruptkontext.

Viele wissen nicht, dass Linux es bei Mehrkernmaschinen erlaubt, im laufenden Betrieb einzelne Cores aus- und auch wieder einzuschalten. Dafür ausgelegte Mehrprozessor-Hardware vorausgesetzt, funktioniert sogar das Ein- und Ausstecken ganzer CPU-Module. Der pro CPU-Kern einmal vorhandene Kernelthread »cpuhp« übernimmt das CPU-Hotplugging. Die Threads initialisieren bei diesem Vorgang die CPU-Kerne beziehungsweise Module und integrieren sie ins System.

Arbeitstiere im Überfluss

Bei der Durchsicht der deutlich über 100 Kernelthreads fällt die Häufung einzelner Namen auf. Besitzer von Maschinen mit acht Kernen stoßen beispielsweise auf mehr als 40 (!) »kworker«-Kernelthreads (Abbildung 5). Dieser Thread-Typ erledigt allgemeine Aufgaben im Kernel: Eine Vielzahl von Komponenten, insbesondere Gerätetreiber, nehmen seine Dienste in Anspruch.

Den Thread gibt es aber nicht deshalb so häufig, weil es so viel zu tun gäbe. Vielmehr instanziert der Kernel für jeden CPU-Kern gleich mehrere: drei normale und zwei hochpriore. Zu diesen 40 Threads (8 mal 5) kommen noch einige, die nicht einem CPU-Kern fest zugeordnet sind. Wer in seinem eigenen Kernelcode auf den Kworker-Thread zurückgreifen will, findet unter [5] Programmierhinweise.

Nicht ganz so häufig, dafür aber unabhängig von der Anzahl der CPU-Kerne ist der Kernelthread »bioset«. Mit seinen vielfachen Instanzen agiert er als Teil des IO-Subsystems und sorgt für den reibungslosen Datenverkehr zwischen Hintergrundspeicher (Festplatte, SSD) und Hauptspeicher.

Die Anzahl steht in Relation zur Anzahl der Speichercontroller. Ein neu eingesteckter USB-Stick führt direkt zu einem zusätzlichen Thread. Beruhigenderweise schlafen diese Threads meistens. Erst wenn das System unter Speicherdruck gerät, wird zur Verhinderung eines Deadlock der eine oder andere Bioset-Rettungs-Thread aktiv.

Kernelthreads für eingebettete Systeme

Mit dem Thread »kdevtmpfs« erzeugt und verwaltet der Kernel sein eigenes Gerätedatei-Verzeichnis »/dev/«. Dazu startet Linux den Kernelthread während des Bootens und noch bevor es die Gerätetreiber initialisiert. Das Gerätedatei-Verzeichnis ist technisch gesehen ein Temp-Filesystem, siedelt also nicht auf einem Hintergrundspeicher (SSD, Festplatte), sondern als virtuelles Dateisystem im Hauptspeicher [6]. Der Kernelthread bringt so Udevd um seinen Job, der seit einigen Jahren im Userland die Gerätedateien anlegt und verwaltet.

Da anders als Udevd der Kernelthread keine Konfiguration hat, bekommen alle Gerätedateien einen Namen, der dem Gerätenamen im Sys-Filesystem entspricht. Auf die Gerätedateien lesen und schreiben darf zunächst nur der Superuser Root. Programme wie der Udevd können zu einem späteren Zeitpunkt die Einträge ändern beziehungsweise anpassen. Eingebettete Systeme jedoch kommen dank »kdevtmpfs« auch ohne einen Udevd aus.

Masse ist Klasse

Die in diesem Artikel vorgestellten Kernelthreads sind bei Weitem nicht alle. Aber schon diese beweisen zur Genüge: Linux ist komplex geworden. Die Kernelhacker entnehmen immer mehr Funktionalität aus dem Kernel-Inneren und lagern den Code in extra Threads aus. Die Entkoppelung spiegelt auch interne Abläufe besser wider.

Die Beschäftigung mit Kernelthreads befriedigt allerdings nicht nur die Neugierde des Linux-Anwenders, sondern offenbart diverse Stellschrauben der Systemoptimierung. Unglücklicherweise gibt es über Kernelthreads bis dato viel zu wenig Dokumentation. Da wird wohl eine künftige „Kern-Technik" noch mal Thread-scheinig ranmüssen. *(jk)* ■

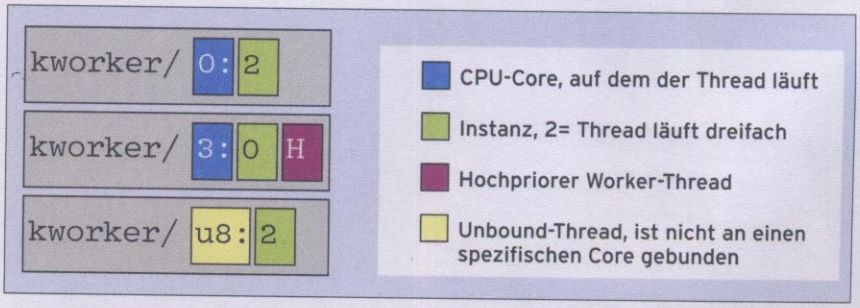

Abbildung 5: Namenskonvention von »kworker«-Kernelthreads. Die Zählungen beginnen bei null.

Infos

[1] Quade, Kunst, „Kern-Technik", Folge 33 zum Multicore-Scheduling: Linux-Magazin 05/07, S. 52
[2] OOM Killer: [https://linux-mm.org/OOM_Killer]
[3] Michal Hocko, „mm, oom: introduce oom reaper": [https://lwn.net/Articles/666024/]
[4] Quade, Kunst, „Kern-Technik", Folge 83 zu Read Copy Update: Linux-Magazin 11/15, S. 84
[5] Quade, Kunst, „Kern-Technik", Folge 84 zu Workqueues: Linux-Magazin 01/16, S. 82
[6] Quade, Kunst, „Kern-Technik", Folge 39 zum RAM- und Tmp-Filesystem: Linux-Magazin 05/08, S. 98

Die Autoren

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von Open Source. Jürgen Quade ist Professor an der Hochschule Niederrhein. Ihr gemeinsames Buch „Linux-Treiber entwickeln" ist Ende 2015 in vierter Auflage erschienen.