Der O(1)-Scheduler im Kernel 2.6

Die Reihenfolge zählt

Die Leistungsentfaltung von Betriebssystemen hängt entscheidend von der Performance und der Strategie des Schedulers ab, der die Prozessliste führt und lauffähigen Prozessen die CPU scheibchenweise zuteilt. Der Scheduler des Kernels 2.6 ist komplett umgeschrieben und erfüllt die Anforderungen in nachweisbar hohem Maße. Timo Hönig

Die Erneuerung des Scheduler im Kernel 2.6 gehört zu den unauffälligen Maßnahmen – kein Programm kann dadurch auch nur eine Funktion mehr bieten. Trotzdem profitieren praktisch alle Anwender von der Generalüberholung: Die Embedded-Fraktion freut die Reaktionsfreudigkeit samt „weicher" Echtzeiteigenschaften, große Multiprozessorsysteme kommen dank viel besserer SMP-Skalierung des neuen Kernels unter Last richtig in Fahrt und die Besitzer normaler Desktop-PCs müssten mit Entzücken auf die zügige Benutzer-System-Interaktion reagieren.

Die Struktur, die Verwaltungsstrategien und die praktischen Vorteile des neuen Schedulers im Vergleich zu dem des Kernels 2.4 vermittelt der folgende Beitrag. Hinzu kommen einige Grundlagen der Prozessverwaltung, die dem Verstehen des Funktionsprinzips dienlich sind.

Prozessverwaltung

Linux verwaltet alle Prozessinformationen mit Hilfe einer doppelt verketteten Liste – der Taskliste. Die Listenelemente (siehe Abbildung 1) sind die Prozessdeskriptoren (»task_struct«) der Prozesse. Der Deskriptor hält alle Informationen seines Prozesses fest: Prozessidentifikator (PID), Adressraum, Prozessstatus, Priorität, Signalhandler et cetera – das »ps«-Kommando macht diese Informationen dem Benutzer zugänglich. Wer neugierig ist, kann die komplette Struktur »task_strut« im Kernel »include/linux/sched.h« studieren.

Ein Verweis reicht auch

Im Vergleich zu Linux 2.4 speichert der Kernel 2.6 auf dem Kernelstack eines jeden Prozesses nicht mehr den kompletten Prozessdeskriptor, sondern eine Struktur (»thread_info«), die einen Verweis auf den eigentlichen Prozessdeskriptor der Taskliste beinhaltet. Den Prozessdeskriptor selbst erzeugt wie bisher der Slab-Allocator dynamisch.

Den Zustand eines Prozesses speichert die Variable »state« des Prozessdeskriptors. Der Scheduler kennt insgesamt fünf Zustände:

- »TASK_RUNNING« kennzeichnet den Prozess als lauffähig. Er muss auf kein Ereignis warten und kann daher vom Scheduler der CPU zugeordnet werden. Alle Prozesse im Zustand »TASK_RUNNING« zieht der Scheduler für die Ausführung in Betracht.
- Ein blockierter Prozess befindet sich im Zustand »TASK_INTERRUPTIBLE«, da er auf ein Ereignis wartet und vor dessen Eintreten nicht ablaufen kann. Ein Prozess im Zustand »TASK_INTERRUPTIBLE« wird über zwei unterschiedliche Wege in den Zustand »TASK_RUNNING« versetzt: Entweder tritt das Ereignis ein, auf das er gewartet hat, oder der Prozess wird durch ein Signal aufgeweckt.
- »TASK_UNINTERRUPTIBLE« gleicht dem Zustand »TASK_INTERRUPTIBLE«, mit dem Unterschied, dass ein Signal den Prozess nicht aufwecken kann. Der Zustand »TASK_UNINTERRUPTIBLE« wird nur verwendet, wenn zu erwarten ist, dass das Ereignis, auf das der Prozess wartet, zügig eintritt, oder wenn der Prozess ohne Unterbrechung warten soll.
- Wurde ein Prozess beendet, dessen Elternprozess noch nicht den Systemaufruf »wait4()« ausgeführt hat, verbleibt er im Zustand »TASK_ZOM-

BIE«. So kann auch nach dem Beenden eines Kindprozesses der Elternprozess noch auf seine Daten zugreifen. Nachdem der Elternprozess »wait4()« aufgerufen hat, wird der Kindprozess endgültig beendet, seine Datenstrukturen werden gelöscht. Endet ein Elternprozess vor seinen Kindprozessen, bekommt jedes Kind einen neuen Elternprozess zugeordnet. Dieser ist nunmehr dafür verantwortlich, »wait4()« aufzurufen, sobald der Kindprozess beendet wird. Ansonsten könnten die Kindprozesse den Zustand »TASK_ZOMBIE« nicht verlassen und würden als Leichen im Hauptspeicher zurückbleiben.

- Den Zustand »TASK_STOPPED« erreicht ein Prozess, wenn er beendet wurde und nicht weiter ausführbar ist. In diesen Zustand tritt der Prozess ein, sobald er eines der Signale »SIGSTOP«, »SIGTST«, »SIGTTIN« oder »SIGTTOU« erhält.

Abbildung 2 zeigt die Prozesszustände und ihre Übergänge.

Die Entwicklungsziele für den Scheduler

In jedem Multitasking-Betriebssystem stellt der Scheduler die Basis für die virtuelle Nebenläufigkeit der Prozesse: Allein er wacht darüber, dass lauffähige Prozesse CPU-Zeit erhalten. Außerdem berechnet er Prozessprioritäten und Timeslices (Zeitscheiben). Die CPU(s) sollen optimal ausgenutzt werden: Solange es ablauffähige Prozesse gibt, soll ein Prozess der CPU zugeteilt sein.

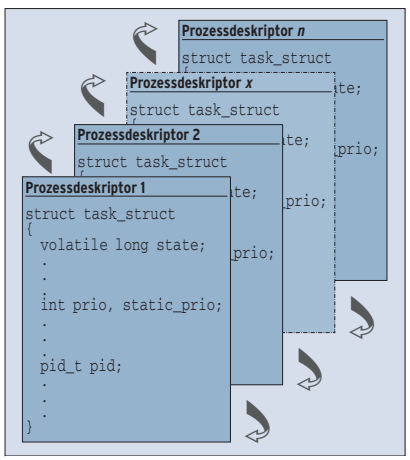

Abbildung 1: Linux verwaltet Prozessinformationen mit einer doppelt verketteten Liste – der Taskliste.

Die Ziele, die sich die Entwickler um Ingo Molnar für den Scheduler von Kernel 2.6 gesetzt hatten, sind besonders ehrgeizig: Perfekte SMP-Skalierung, SMP-Affinität, geringe Latenz auch bei hoher Systemlast, faire Prioritätenverteilung und eine Komplexität der Ordnung O(1). Dabei gab es für die meisten Einsatzgebiete am Scheduler des Kernels 2.4 gar nicht nicht so viel auszusetzen. Nur bei SMP-Systemen mit mehr als vier CPUs und in Worst-Case-Situationen geriet das System aus den Fugen.

Kaum verwunderlich also, dass die Idee, den Scheduler tief greifend zu überarbeiten und vieles zu erneuern, zunächst bei Linus & Co. auf teils rauen Widerstand stieß. Letztlich konnte Ingo Molnar aber die Vorteile seines Konzepts verdeutlichen. Dass seine Überlegungen sinnvoll waren, spiegelt sich in den messbar positiven Eigenschaften des neuen Schedulers wider.

Alle Linux-Scheduler bisher besaßen eine Komplexität der Ordnung O(n): Die Kosten (also der Aufwand) für das Scheduling wuchsen linear mit der Anzahl n der lauffähigen Prozesse. Das Ziel des neuen Schedulers war, den Scheduling-Aufwand von der Anzahl der lauffähigen Prozesse abzukoppeln, was der Ordnung O(1) entspricht. Konkret: Der Scheduler benötigt für die Verwaltung von fünf lauffähigen Prozessen die gleiche Zeit wie für 500 000. Das hieß aber, dass alle Scheduling-Algorithmen so umgeschrieben oder ersetzt werden mussten, dass sie der Ordnung O(1) genügen.

Fairplay: Prozessprioritäten

Seine Strategie bewirkt, dass der neue Scheduler interaktive Prozesse (wie Shells und Editoren) auch unter hoher Systemlast regelmäßig in den Zustand »TASK_RUNNING« versetzt und ablaufen lässt. Die Prozessprioritäten entscheiden, welchen lauffähigen Prozess die CPU beim nächsten Kontextwechsel zugeteilt bekommt – den mit der zum Zeitpunkt des Kontextwechsels höchsten Priorität. Das Besondere: Die Priorität eines Prozesses ändert sich dynamisch während seiner Laufzeit.

Alle lauffähigen Prozesse verwaltet der Scheduler in einer Run-Queue (pro CPU). Sie ist die zentrale Datenstruktur, auf der der Scheduler arbeitet. Neben Verweisen auf die gerade laufende Task, enthält sie Verweise zu den zwei Priority-Arrays »active« und »expired«. Listing 1 zeigt einen Ausschnitt der Struktur »runqueue«, Listing 2 die Struktur der Priority-Arrays.

Das »active«-Array listet alle lauffähigen Prozesse, deren Zeitscheibe noch nicht abgelaufen ist. Wenn die Zeitscheibe eines Prozesse abläuft, verschiebt der Scheduler den Eintrag vom »active«- in das zweite Array »expired«.

Statisch vs. dynamisch

Es gibt zwei unterschiedliche Prioritäten: die statische Prozesspriorität, also die vom »nice«-Wert bestimmte »static_prio«, und die dynamische (effektive) Prozesspriorität (»prio«), die der Scheduler aufgrund der Interaktivität eines Prozesses berechnet. Der Wertebereich des »nice«-Values reicht von -20 (dem höchsten) bis 19 (dem niedrigsten).

Linux 2.6 kennt standardmäßig 140 Prioritätslevels (siehe Abbildung 3). Hierbei

Listing 1: Struktur der Run-Queue

```
01 struct runqueue {
02       /* Spinlock um Run-Queue zu schützen */
03       spinlock_t lock;
04
05       /* Zahl der lauffähigen Prozesse */
06       unsigned long nr_running;
07       /* Zahl der bisherigen Kontextwechsel */
08       unsigned long nr_switches;
09       /* Zeitstempel des Letzten Tauschs von active-
   und expired-Array */
10       unsigned long expired_timestamp;
11       /* Zahl der Prozess im Zustand
   TASK_UNINTERRUPTIBLE */
12       unsigned long nr_uninterruptible;
13
14       /* Verweis auf Prozessdeskriptor des momentan
   ablaufenden Prozesses */
15       task_t *curr;
16       /* Verweis auf Prozessdeskriptor der Idle-Task */
17       task_t *idle;
18       /* Verweis auf Memory Map des zuletzt ablaufenden
   Prozesses */
19       struct mm_struct *prev_mm;
20
21       /* Verweise auf active- und expired-Array */
22       prio_array_t *active, *expired;
23       /* Priority-Arrays */
24       prio_array_t  arrays[2];
25
26       ...
27 }
```

entspricht null der höchsten und 139 der niedrigsten Priorität. Die Levels von eins bis 99 sind für Tasks mit Echtzeitpriorität reserviert. Alle anderen Prozesse erhalten zunächst gemäß ihres »nice«-Werts eine Priorität: Der »nice«-Wert (-20 bis 19) wird einfach in den Bereich ab 101 gemappt. Während des Ablaufs eines Prozesses verändert sich durch seinen Interaktivitätsgrad aber seine Priorität (siehe unten).

Beide, das »active«- und das »expired«-Array, führen für jede Priorität eine verkettete Liste der Prozesse mit entsprechender Priorität. Eine Bitmap hält fest, für welche Priorität mindestens eine Task existiert. Alle Bits werden bei der Initialisierung auf null gesetzt. Beim Eintragen eines Prozesses in eines der beiden Priority-Arrays, wechselt entsprechend der Priorität des Prozesses das korrespondierende Bit im Priorität-Bitmap auf eins. Startet ein Prozess mit dem Nice null, setzt der Scheduler das 120. Bit des Priorität-Bitmaps im »active«-Array und reiht ihn in die Prozessliste mit Priorität 120 ein (siehe Abbildung 4).

Prioritäten allerorten

Analog dazu löscht sich das entsprechende Bit im Priorität-Bitmap, sobald der Scheduler den letzten Prozess einer gegebenen Priorität aus einem der beiden Priority-Arrays austrägt. Es ist – wie noch in Linux 2.4 – nicht mehr nötig, die komplette Liste der lauffähigen Prozesse zu durchsuchen, um für den nächsten Taskwechsel den Prozess mit der höchsten Priorität auszumachen. Der Scheduler muss lediglich das erste gesetzte Bit des Priorität-Bitmaps finden.

Da das Bitmap eine feste Größe besitzt, benötigt die Routine hierfür einen konstanten Zeitraum – unabhängig von der Anzahl der lauffähigen Prozesse. Anschließend führt der Scheduler den ersten Prozess aus der verketteten Liste dieser Priorität aus. Prozesse gleicher Priorität bekommen die CPU nacheinander in einem Ringverfahren (Round Robin) zugeteilt.

Listing 2: Struktur der Priority-Arrays

```
struct prio_array {
        /* Zahl der Prozesse */
        int nr_active;
        /* Priorität-Bitmap */
        unsigned long bitmap[BITMAP_SIZE];
        /* Für jede Priorität eine Liste */
        /* der Prozesse mit entsprechender Priorität */
        struct list_head queue[MAX_PRIO];
};
```

Prozess-Zeitscheiben

Die Zeitscheibe eines Prozesses gibt an, wie lange er laufen darf ohne verdrängt zu werden. Die Größe der Zeitscheibe eines Prozesses ist von seiner Priorität abhängig: Prozesse mit hoher Priorität erhalten mehr CPU-Zeit als solche mit niedriger. Die kleinste Zeitscheibe beträgt 10, die längste 200 Millisekunden. Ein Prozess mit dem »nice«-Wert null erhält die Standard-Zeitscheibe von 100 Millisekunden.

Ist die Zeitscheibe eines Prozesses aufgebraucht, muss der Scheduler sie neu berechnen und den Prozess aus dem »active«- in das »expired«-Array verschieben. Sobald »active« leer ist – alle lauffähigen Prozesse haben ihre Zeitscheibe aufgebraucht –, tauscht der Scheduler einfach das »active«- gegen das »expired«-Array aus. Effektiv wechseln nur die zwei Pointer der Run-Queue die Plätze.

Dieser Kniff macht die Reduktion der Komplexität im Vergleich zum alten Scheduler deutlich: In Linux 2.4 werden die Zeitscheiben aller Prozesse auf einmal neu berechnet – immer dann, wenn alle Prozesse ihre Zeitscheiben aufgebraucht haben. Mit steigender Prozesszahl dauert die Berechnung immer länger.

Neuberechnung der Prioritäten und Zeitscheiben

Wie bereits in Linux 2.4 trifft der neue Scheduler nicht aufgrund der statischen Prozesspriorität (»static_prio«) seine Entscheidungen, sondern anhand der dynamischen. Die dynamische Priorität ergibt sich aus der statischen und der Prozessinteraktivität.

Gemäß seiner Interaktivität erhält ein Prozess vom Scheduler entweder einen Bonus oder ein Penalty (Malus). Interaktive Prozesse gewinnen über einen Bonus maximal fünf Prioritätslevels hinzu, während jene Prozesse, die eine geringe Interaktivität aufweisen, maximal fünf Prioritätslevels durch ein Penalty verlieren. Die dynamische Priorität eines Prozesses mit einem »nice«-Wert fünf beträgt demnach im besten Fall null und im schlechtesten zehn.

Um den Grad der Interaktivität eines Prozesses zu bestimmen, muss bekannt sein, ob der Prozess eher I/O-lastig (I/O-Bound) oder eher CPU-intensiv (Processor-Bound) ist. Prozesse, die die meiste Zeit auf Ein- oder Ausgaben warten (zum Beispiel eine Shell, die auf Eingaben des Benutzers wartet), gehören zur

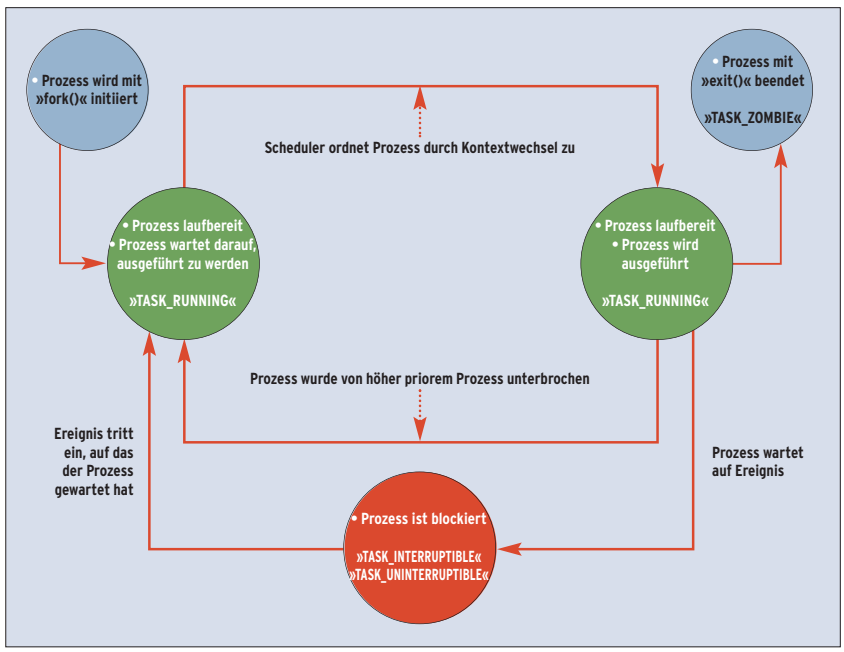

Abbildung 2: Die möglichen Prozesszustände und ihre Übergänge.

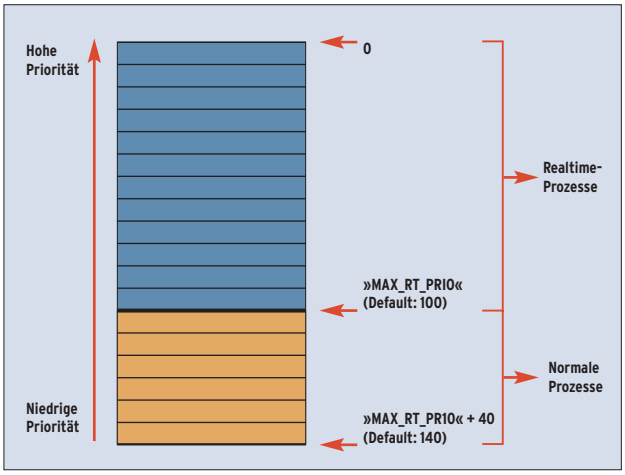

Abbildung 3: Linux 2.6 kennt standardmäßig 140 Prioritätslevels.

ersten Kategorie. Prozesse, die viel Code ausführen und nur selten warten (wie ein Video-Codec), fallen in die Kategorie Processor-Bound.

Überwachung pur

Um Prozesse einer der beiden Kategorien zuordnen zu können, protokolliert der Kernel für jeden Prozess, wie viel Zeit er mit Schlafen verbringt, beispielsweise weil er auf ein I/O-Ereignis wartet, und wie lange er die CPU in Anspruch nimmt. Die Variable »sleep_avg« (Sleep Average) im Prozessdeskriptor speichert dafür eine Entsprechung in dem Wertebereich von null und zehn (»MAX_SLEEP_AVG«).

Läuft ein Prozess, verringert seine »sleep_avg« mit jedem Timer-Tick ihren Wert. Sobald ein schlafender Prozess aufgeweckt wird und in den Zustand »TASK_RUNNING« wechselt, wird »sleep_avg« entsprechend seiner Schlafzeit erhöht – maximal bis zu »MAX_SLEEP_AVG«. Der Wert von »sleep_avg« ist somit maßgebend, ob ein Prozess I/O- oder Processor-Bound ist. Interaktive Prozesse haben eine hohe »sleep_avg«, minder interaktive eine niedrige.

Mit der Funktion »effective_prio()« berechnet der Scheduler die dynamische Priorität »prio« basierend auf der statischen »static_prio« und der Interaktivität »sleep_avg« des Prozesses. Zum Berechnen der neuen Zeitscheibe greift der Scheduler auf die dynamische Prozesspriorität zurück. Dazu mappt er den Wert in den Zeitscheibenbereich »MIN_TIMESLICE« (Default: 10 Millisekunden) bis »MAX_TIMESLICE« (200 Millisekunden).

Interaktive Prozesse mit hohem Bonus und großer Zeitscheibe können ihre Priorität jedoch nicht missbrauchen, um die CPU zu blockieren: Da die Zeit, die der Prozess beim Ausführen im Zustand »TASK_RUNNING« verbringt, in die Berechnung der »sleep_avg«-Variablen eingeht, verliert solch ein Prozess schnell seinen Bonus und mit ihm seine hohe Priorität und seine große Zeitscheibe.

Ein Prozess mit sehr hoher Interaktivität erhält nicht nur eine hohe dynamische Priorität und somit eine große Zeitscheibe: Der Scheduler trägt den Prozess nach Ablauf seiner Zeitscheibe auch wieder sofort in das »active«-Array ein, statt wie gewöhnlich ins »expired«-Array. Der Prozess wird dadurch seiner Priorität gemäß wieder zugeordnet und muss nicht auf das Austauschen der Priority-Arrays warten.

Goodness ist schlecht

Zum Vergleich: Linux 2.4 bestimmt die dynamische Priorität eines Prozesses mit der Funktion »goodness()«. Als Grundlage dient wieder der »nice«-Wert, aus dem sich die Basislänge der Zeitscheibe berechnet, die ein Prozess erhält. Für jeden hält der Zähler »counter« fest, wie viel Zeit er von seiner Zeitscheibe verbraucht, also die CPU in Anspruch genommen hat.

Tasks, die beim Aufruf von »goodness()« bisher wenig von ihrer Zeitscheibe verbraucht haben, bekommen eine hohe dynamische Priorität und umgekehrt. Linux 2.4 muss also bei jedem Kontextwechsel »goodness()« für alle lauffähigen Prozesse aufrufen, um den Prozess mit der höchsten dynamischen Priorität zu finden.

Symmetric Multi Processing: Skalierung und Affinität

Das Load-Balancing bis zum Kernel 2.4 funktioniert so: Bei jedem Kontextwechsel prüfte der Kernel, ob der neu zugeordnete Prozess nicht besser auf einer anderen CPU laufen sollte, und verschiebt ihn gegebenenfalls. Der Nachteil: Ein Kontextwechsel auf einer CPU beeinträchtigt die anderen CPUs, was sich auf die Performance niederschlägt. Auch neigt der alte Scheduler auf Multiprozessorsystemen dazu, Prozesse unnötig zwischen CPUs hin und her zu schieben (Process Bouncing).

Der neue Scheduler verhindert dies und sorgt dafür, dass Prozesse von der ihnen zugewiesenen CPU „angezogen“ werden. Seine neuen Datenstrukturen und die Operatoren für deren Bearbeitung sind auf gute SMP-Skalierbarkeit für Systeme mit vielen CPUs getrimmt (siehe Kasten „Benchmarking“). Jede CPU verwaltet eine eigene Run-Queue. Zudem betreffen die Locking-Mechanismen nur die für den Prozess relevante CPU.

Dafür, dass die Run-Queues nicht entarten, sorgt die Funktion »load_balance()«. Sie hält die Anzahl der Prozesse pro CPU etwa im Gleichgewicht und wird immer dann aufgerufen, wenn die Run-Queue eines Prozessors leer ist. Außerdem bedient ein Timer zyklisch die Funktion:

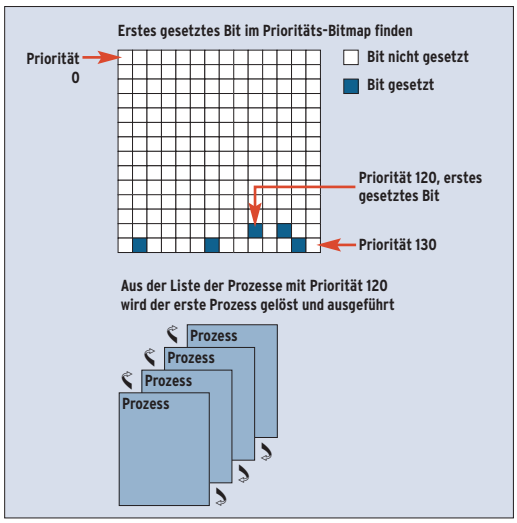

Abbildung 4: Beim Eintragen eines Prozesses in eines der beiden Priority-Arrays, wechselt entsprechend der Priorität des Prozesses das korrespondierende Bit im Prioritäts-Bitmap auf eins.

Wenn das System gerade idle ist, ruft er »load_balance()« je Millisekunde auf, unter Last alle 200 Millisekunden.

»load_balance()« arbeitet für jede Run-Queue getrennt. Als Erstes lockt der Scheduler die aktuelle Run-Queue, was Veränderungen an ihr durch konkurrierende Zugriffe während des Load-Balancing verhindert. Mit der Funktion »find_busiest_queue()« sucht der Scheduler unter den Run-Queues der anderen CPUs jene mit den meisten Prozessen. Hat keine der Queues mindestens 25 Prozent mehr Prozesse zu erledigen als die aktuelle, beendet sich »load_balance()« und nimmt somit sinnvollerweise keinen Ausgleich zwischen den Run-Queues vor.

Andernfalls liefert »find_busiest_queue()« die Run-Queue mit den meisten Prozessen zurück und »load_balance()« entscheidet, ob aus ihrem »active«- oder »expired«-Array Prozesse entnommen werden. Die gewählten Prozesse verschiebt der Scheduler später in die Run-Queue der aktuellen CPU. »load_balance()« entnimmt Prozesse bevorzugt dem »expired«-Array, da sie wahrscheinlich seit längerer Zeit nicht mehr gelaufen sind und sich deshalb nicht im Cache einer CPU befinden. Nur wenn »expired« leer ist, kommen Prozesse aus dem »active«-Array zum Zuge.

Aus dem gewählten Array wird über das Prioritäts-Bitmap die Liste der Prozesse mit höchster Priorität ausgewählt. Dann wird aus dieser Liste jener Prozess gelöst, der sich weder im Cache der CPU befindet noch zum Zeitpunkt des Load-Balancing schläft und auch durch SMP-Affinität nicht daran gehindert wird, die CPU zu wechseln. Dieser Vorgang wird so lange wiederholt, bis die »runqueues« bis auf das 25-Prozent-Limit ausgeglichen sind. Solange die Run-Queues aber nicht entarten, hat »load_balance()« keinen Anlass dafür, Prozesse zwischen den CPUs zu transferieren.

Als Sonderfall ist es möglich, explizit festzulegen, dass ein Prozess nur auf bestimmten Prozessoren läuft (Hard Affinity) und vom Load-Balancing ausgeschlossen wird. Das passiert über die Bitmaske »cpus_allowed« im Prozessdeskriptor, jedes Bit entspricht einer CPU im System. Die Initialisierung setzt zunächst immer alle Bits auf eins und der Prozess darf auf jeder CPU laufen. Mit der Funktion »sched_affinity()« ist die Bitmaske jetzt beliebig manipulierbar – mindestens ein Bit muss aber gesetzt bleiben. Anschließend verschmäht der Prozess garantiert alle CPUs, deren »cpus_allowed«-Flag nicht gesetzt ist.

Prozess-Preemption und Kontextwechsel

Alle blockierten Prozesse – sie warten ohne Schuld des Schedulers auf ein Ereignis – werden in den so genannten Wait-Queues verwaltet. Prozesse, die von »TASK_RUNNING« in den Zustand »TASK_INTERRUPTIBLE« oder »TASK_UNINTERRUPTIBLE« wechseln, gelangen

in diese Warteschlange. Anschließend ruft der Kernel »schedule()« auf, damit ein anderer Prozess die CPU erhält.
Sobald das Ereignis eintritt, auf das der Prozess in einer Wait-Queue wartet, wird er aufgeweckt, wechselt seinen Zustand in »TASK_RUNNING« zurück, verlässt die Wait-Queue und betritt die Run-Queue. Falls der aufgewachte Prozess eine höhere Priorität besitzt als der gerade ablaufende, unterbricht der Scheduler den aktuell laufenden Prozess zugunsten des eben aufgewachten.
Den Kontextwechsel löst die erwähnte Funktion »schedule()« aus. Der Kernel erfährt über das Flag »need_resched«, wann er »schedule()« aufrufen muss. »need_resched« ist als Nachricht realisiert, die an den Kernel geschickt wird, sobald das Flag gesetzt ist. Das macht die Funktion »scheduler_tick()«, sobald die aktuelle Zeitscheibe abläuft. Zum anderen setzt die Funktion »try_to_wakeup()« das Flag, sollte ein schlafender Prozess in den Zustand »TASK_RUNNING« wechseln, der eine höhere Priorität als der zurzeit ablaufende Prozess besitzt.

Den eigentliche Wechsel zwischen zwei Prozessen realisiert die durch »schedule()« aufgerufene Funktion »switch_context()«. Der Kontextwechsel ist in zwei Schritte gegliedert: »switch_mm()« tauscht zunächst den virtuelle Adressraum des alten Prozesses gegen den des neuen. Anschließend sichert »switch_to()« den Zustand (Stack, Register) des bisher laufenden Prozesses und lädt den Zustand des neuen.

Kernel-Preemption

Anders als Kernel 2.4 ist der neue Linux-Kernel preemptiv [1]: Kernelcode, der gerade ausgeführt wird, kann unterbrochen werden. Vor dem Unterbrechen muss gewährleistet sein, dass sich der Kernel in einem Zustand befindet, der eine Neuzuordnung der Prozesse zulässt. Die Struktur »thread_info« jedes Prozesses enthält zu diesem Zweck den Zähler »preempt_count«. Ist er null, ist der Kernel in einem sicheren Zustand und darf unterbrochen werden.
Die Funktion »preempt_disable()« erhöht den Zähler »preempt_count« beim Setzten eines so genannten Locks um eins; die Funktion »preempt_enable()« erniedrigt ihn um eins, sobald ein Lock aufgelöst wird. Das Setzten des Locks (und damit das Verbot der Kernel-Preemption) wird immer dann notwendig, wenn beispielsweise eine von zwei Prozessen genutzte Variable vor konkurrierenden Zugriffen zu sichern ist.

Realtime

Für Prozesse mit so genannter Echtzeitpriorität (Priorität 1 bis 99) gibt es zwei Strategien: »SCHED_FIFO« und »SCHED_RR«. »SCHED_FIFO« ist ein einfacher First-in/First-out-Algorithmus, der ohne Zeitscheiben arbeitet. Wird ein Echtzeitprozess mit »SCHED_FIFO« gestartet, läuft er so lange, bis er blockiert oder freiwillig über die Funktion »sched_yield()« den Prozessor abgibt. Alle anderen Prozesse mit einer niedrigeren Priorität sind solange blockiert und werden nicht ausgeführt.
»SCHED_RR« verfolgt die gleiche Strategie wie »SCHED_FIFO«, aber zusätzlich mit vorgegebenen Zeitscheiben. Die

Benchmarking

Spätestens seit Galilei („Alles messen, was zu messen ist, und messbar machen, was noch nicht messbar ist") hat die Quantifizierung Konjunktur. Aber selbst ohne wissenschaftshistorisches Fundament ist der praktische Leistungsvergleich der Scheduler von Kernel 2.4 und 2.6 interessant. Um deren Performance, den Overhead und die Skalierbarkeit zu testen, eignet sich das Messprogramm Hackbench [3] von Rusty Russell.

Interprozess-Kommunikation als Maßstab

Der Benchmark startet gruppenweise Client- und Serverprozesse, die über Sockets Nachrichten austauschen. Beim den hier vorgestellten Testergebnissen [4] bestand jede Gruppe aus je 25 Clients und 25 Servern. Jeder Client schickte an jeden der 25 Server-Sockets 100 Nachrichten. Hackbench misst die für die Ausführung benötigte Zeit. Jeder Testlauf wurde viermal wiederholt und der Mittelwert als Resultat festgehalten.

Linux 2.6 erläuft sich einen großen Vorsprung

Für den ersten Benchmark (siehe Abbildung 5) kam ein System mit einer CPU (Intel Pentium III, 1 GHz) und 512 MByte RAM zum Einsatz. Auf dem Rechner liegen die Zeitwerte für den Testlauf mit 25 Prozessen bei Linux 2.4 und 2.6 noch nahe beieinander. Bei 100 Prozessen – was der typischen Prozesszahl eines Desktopsystems nahe kommt – hat Linux 2.6 mit 17,86 Sekunden bereits einen weiten Vorsprung vor 2.4 mit 37,63 Sekunden.

Die beiden anderen Diagramme fassen die Ergebnisse der Testläufe auf vier verschiedenen Systemen zusammen:
- 1 CPU: Pentium III 1 GHz, 512 MByte RAM
- 2 CPU: Pentium III 850 MHz, 1 GByte RAM
- 4 CPU: Pentium III 700 MHz, 4 GByte RAM
- 8 CPU: Pentium III 700 MHz, 8 GByte RAM

Die Abbildung 6 zeigt die Werte für Kernel 2.4 und Abbildung 7 für 2.6. Es ist schön deutlich zu sehen, dass Linux 2.6 bei steigender Prozesszahl auf allen vier Systemen ungleich besser skaliert als der alte Kernel.
Ab 150 Prozessen ist der Kernel 2.4 auf Zwei- und Acht-Prozessor-Systemen nahezu gleich schnell. Linux 2.6 skaliert dagegen bei steigender Prozesszahl in erwartetem Maß – auch auf dem Acht-Prozessor-System.

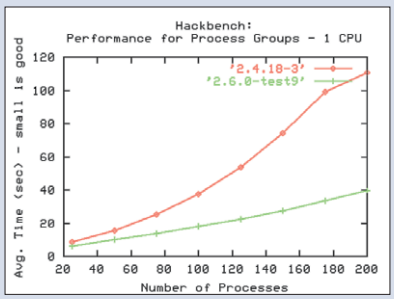

Abbildung 5: Benötigte Zeit zur Interprozess-Kommunikation in Abhängigkeit von der Anzahl beteiligter Prozesse auf einem Singleprozessor-System mit Kernel 2.4 und 2.6. (Quelle: [4])

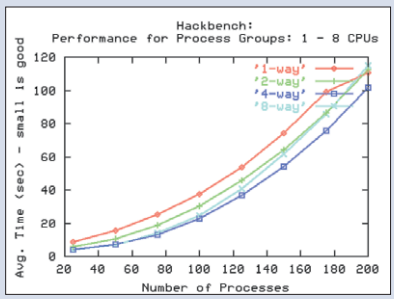

Abbildung 6: Benötigte Zeit zur Interprozess-Kommunikation in Abhängigkeit von der Anzahl beteiligter Prozesse auf Systemen mit ein, zwei, vier und acht CPUs mit Kernel 2.4.

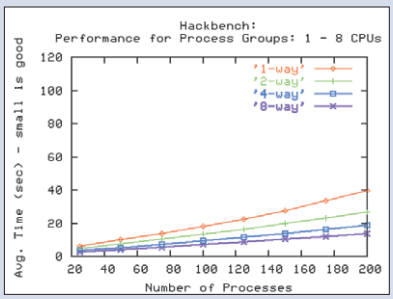

Abbildung 7: Benötigte Zeit zur Interprozess-Kommunikation in Abhängigkeit von der Anzahl beteiligter Prozesse auf Systemen mit ein, zwei, vier und acht CPUs mit Kernel 2.6.

CPU-Bedürfnisse der Echtzeitprozesse gleicher Priorität befriedigt der Scheduler per Round-Robin. Prozesse mit einer niedrigeren Priorität kommen überhaupt nicht zum Zuge. Der Scheduler vergibt für Echtzeitprozesse keine dynamischen Prioritäten. Prozesse ohne Echtzeitpriorität führt er mit der Strategie »SCHED_OTHER« aus.

Die Echtzeit-Strategien von Linux garantieren jedoch keine Antwortzeiten, was die Voraussetzung für ein hartes Echtzeit-Betriebssystem wäre. Der Kernel stellt jedoch sicher, dass ein lauffähiger Echtzeit-Prozess immer die CPU bekommt, wenn er auf kein Ereignis warten muss, er freiwillig die CPU abgibt und wenn kein lauffähiger Echtzeitprozess höherer Priorität existiert.

Um die Echtzeit-Möglichkeiten von Linux 2.6 auszuprobieren, sind die Scheduler-Tools **[2]** von Robert Love hilfreich: Mit dem Programm »chrt« lassen sich die Echtzeitattribute eines Prozesses einstellen. So kann man beispielsweise die Echtzeitpriorität (1 bis 99) oder die Scheduling-Strategie (»SCHED_FIFO«, »SCHED_RR« oder »SCHED_OTHER«) eines Prozesses festgelegen.

Zurück in die Zukunft

Der Artikel hat gezeigt, dass der Scheduler im Kernel 2.6 in Struktur und praktischer Leistung dem des 2.4ers vielfältig überlegen ist. Als weiterführende Literatur zum neuen Linux-Kernel eignen sich Robert Loves Buch „Linux Kernel Development“ **[5]** und Wolfgang Maurers „Linux Kernelarchitektur“ **[6]**.

Wer der eigenen Distribution sofort den kompletten Sprung zum neuen Kernel nicht zumuten kann oder will, muss nicht auf die Scheduling-Features von Linux 2.6 verzichten. Denn es gibt entsprechende Backports für Kernel 2.4. Benutzer, die ihren geliebten 2.4er Kernel tunen möchten, sollten einen Blick auf Con Kolivas „-ck“-Patchset werfen, das unter **[7]** verfügbar ist. Es enthält unter anderem den O(1)-Scheduler, die Preemptible-Kernel-Patches und außerdem die Interaktivitäts-Patches von Con Kolivas selbst. *(jk, Chr. Hellwig)* ■

Infos

[1] Preemptible Kernel: [http://kpreempt.sourceforge.net/], [http://tech9.net/rml/linux/]

[2] Scheduler-Tools: [http://www.tech9.net/rml/schedutils/]

[3] Hackbench: [http://developer.osdl.org/craiger/hackbench/src/hackbench.c]

[4] Craig Thomas' Hackbench-Resultate: [http://developer.osdl.org/craiger/hackbench]

[5] Robert Love, „Linux Kernel Development“: Sams Publishing, 2003, ISBN 0-672-32512-8

[6] Wolfgang Maurer, „Linux Kernelarchitektur – Konzepte, Strukturen und Algorithmen von Kernel 2.6“: Carl Hanser Verlag, 2004, ISBN 3-446-22566-8

[7] Con Kolivas Kernel-Patches: [http://kernel.kolivas.org/]