Der $O(1)$ Scheduler im 2.5er Linux Kernel

1 Einführung

2001 begann Ingo Molnar einen Scheduler für den Linuxkernel zu entwerfen, der die bekannten und beliebten Vorzüge des bisherigen Schedulers beibehalten sollte und die inzwischen gut erforschten Schwächen beheben sollte [1].

An Vorzügen galt es zu behalten [2]:

- Gute interaktive Performance bei hoher Load: das System muß schnell auf Benutzereingaben reagieren
- Gute Scheduling-/Wakeup-Performance bei wenigen lauffähigen Prozessen
- Fairness: kein Prozess sollte unvernünftig lange keinen Timeslice zugeteilt bekommen
- Prioritäten: Prozesse müssen ihrer Wichtigkeit nach gekennzeichnet werden können
- SMP Effizienz: Keine CPU soll leer laufen, wenn es Arbeit zu erledigen gibt
- SMP Affinität: Prozesse sollten an einer CPU "kleben" bleiben und eher selten zu einer anderen wechseln
- Zusätzliche Features: Realtime Scheduling, feste CPU-Bindungen

Die zu vermeidenden Nachteile waren:

- Die $O(n)$ Operationen, um z.B. den geeignetsten Prozess zu finden, der jetzt eine CPU bekommen soll. Systemzustände mit sehr vielen lauffähigen Prozessen, wie z.B. laufende JVMs, führten so dazu, dass sehr viel Zeit im Scheduler verbraucht wurde, was die Gesamtperformance des Systems spürbar drosselte.
- SMP Probleme: bisher gab es eine einzige Runqueue, egal wieviel CPUs sich im System befanden. So mußte bei jeglicher Schedulingoperation ein globales Lock gesetzt werden, sodaß parallele Contextswitches auf verschiedenen CPUs nicht parallel ausgeführt werden konnten. Außerdem kam es zu dem Problem des "Random Bouncing" von Prozessen zwischen CPUs. Das lag daran, dass die Timeslices erst dann neu berechnet wurden, wenn alle lauffähigen Prozesse ihre Zeitscheibe verbraucht hatten. D.h. je mehr CPUs sich im System befinden, desto wahrscheinlicher ist es, dass einzelne CPUs rumidlen müssen, bis tatsächlich alle Prozesse fertig sind. Vor allem bei Anwesenheit von interaktiven Prozessen (d.h. welche, die nach kurzer Aktivität schlafen) trat dieses Problem auf. Zusätzlich werden nun auf die zufällig freiwerdenden Prozessoren Prozesse verschoben, obwohl dort nur scheinbar Leerlauf herrscht.

2 Neues Design

2.1 Die Runqueues

Anstatt einer globalen Runqueue gibt es nun pro Prozessor eine Runqueue. Diese wiederum besteht aus zwei Arrays, einem Array der zu dieser CPU affinen Prozesse, die ihre Zeitscheibe noch nicht aufgebraucht haben (active) und einem mit Prozessen, deren Zeitscheibe bereits aufgebraucht ist (expired). Beide Arrays sind bezüglich der Prozesspriorität sortiert. Beide Arrays werden über Pointer referenziert, so dass der active und der expired Pointer einfach geswitched werden, wenn das active Array leer ist. Somit wird dieses Array zum Auffangbehälter für Prozesse, die ihre Zeitscheibe verbraucht haben. Auch können so beliebig viele Prozesse in einer Runqueue verwaltet werden, da die Neuberechnung einer Zeitscheibe umittelbar dann stattfinden kann, wenn ein Prozess in das expired Array verschoben wird. Somit bleibt das $O(1)$-Design gewahrt. Zur Veranschaulichung des Scheduling siehe Abbildung 1.

Um den Prozess mit der höchsten Priorität schnell zu finden, wird ein 64-Bit Bitmapcache als Index verwendet, der es mittels zweier x86 BSFL Befehlen erlaubt, den gewünschten Prozess herauszufiltern.

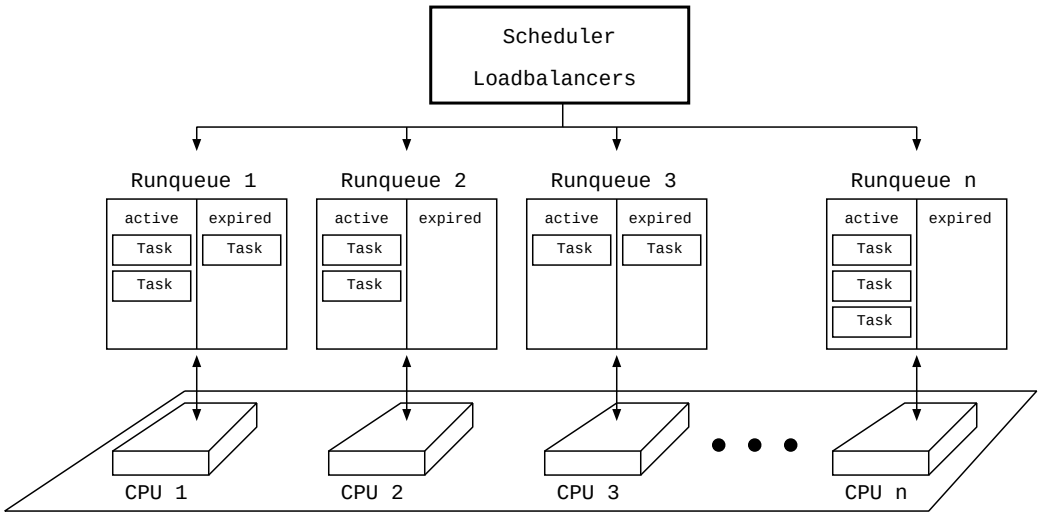

Abbildung 1: Scheduling ab dem 2.5.2 Kernel

2.2 Behandlung interaktiver Prozesse

Eine der schwierigsten Aufgaben ist es, auch unter hoher Systemlast schnelle Antwortzeiten für interaktive Prozesse zu erreichen. Sind beispielsweise die Zeitscheiben zu lang, dauert es merklich, bis der durch ein Ereignis aufgeweckte interaktive Prozess an CPU-Zeit gelangt. Als "interaktiv" werden Prozesse bezeichnet, die die meiste Zeit schlafen, da sie auf Benutzereingaben warten. Damit ein System subjektiv als schnell erscheint, ist es wichtig, dass diese Prozesse so schnell wie möglich an die CPU gelangen, da sie ja nur kurz Rechenzeit beanspruchen und somit die restlichen Prozesse auch nicht behindern.

Um interaktive Prozesse bevorzugen zu können, wird ein *Sleeping Average* Wert berechnet, mittels dessen eine effektive Priorität berechnet wird, die die tatsächliche Priorität eines Prozesses um den Wertebereich $[-5, +5]$ dynamisch verändern kann. Somit wird häufiges Schlafen belohnt. Wird ein Prozess nach dieser Heuristik nun als interaktiv eingestuft, wird er wieder in das active-Array eingereiht, wenn er seine Zeitscheibe aufgebraucht hat. Da so aber auch die Gefahr besteht, dass Prozesse im expired Array "verhungern" können, werden interaktive Prozesse ins expired Array eingehängt, sobald Prozesse im expired Array eine bestimmte Wartezeit überschritten haben. Auch sinkt die Bevorzugung eines interaktiven Prozesses, wenn er zuviele Zeitscheiben hintereinander verbraucht, bis er schließlich nicht mehr als interaktiv eingestuft wird.

2.3 Der Loadbalancer

Der Loadbalancer sorgt dafür, dass alle CPUs möglichst gleichmäßig ausgelastet sind. Um selbst möglichst wenig Systemlast zu erzeugen, wird der Laodbalancer nur alle 250 ms aufgerufen, wenn Prozesse aktiv sind. Im Idle-Systemzustand wird er jedoch alle 1-10 ms aufgerufen. Ist eine CPU unbeschäftigt, wird er jedoch sofort aufgerufen. Auf jeder CPU wird eine eigene Instanz des Loadbalancers gestartet, so dass auch dieser Part des Schedulers massiv parallel laufen kann.

Das Vorgehen des Loadbalancers ist folgendermaßen: es wird die am stärksten ausgelastete Runqueue herausgesucht. Aus deren expired Array werden die Prozesse beginnend mit der höchsten Priorität untersucht, ob sie migrierbar sind. Ist das exired Array leer, oder ist kein Prozess des expired Arrays migrierbar, so wird als nächstes das active Array untersucht. Prozesse gelten als nicht migrierbar, wenn sie gerade laufen, wenn sie nach dem `cpus_allowed` Bitfeld nicht auf der Ziel-CPU laufen dürfen oder wenn sie auf ihrer aktuellen CPU *cache-hot* sind, d.h. wenn zuviele positive Cacheeffekte ungenutzt bleiben würden. Auf diese Weise wird versucht soviele Prozesse zu verschieben, dass die Verteilung wieder ausgeglichen ist. Der Loadbalancer kann sehr einfach erweitert werden, um auch komplexere CPU- und Cachehierarchien zu berücksichtigen (z.B. NUMA [3]).

Dass der Loadbalancer unter Last eher selten aufgerufen wird, ist insofern wichtig, als er kein $O(1)$ Verhalten aufweist, sondern zur Klasse $O(n)$ gehört.

3 Meßergebnisse

Laut den Benchmarkergebnissen in [4] zeigt der neue Scheduler z.B. bei einem simulierten Chatserver mit einigen Clients eine Steigerung der Leistung um 20%. Besonders bemerkenswert ist die Leistungssteigerung, wenn die Testprozesse mit Priorität 19, also im Batchmode laufen.

Auch die Anzahl der möglichen Context Switches steigt stark an, auf dem 2-Prozessor Testsystem um 300% im Vergleich zum 2.5.2er Kernel mit altem Scheduler. Richtig beeindruckend zeigen sich die parallelen Runqueues auf dem 8-Prozessor Testsystem, wo der $O(1)$-Scheduler 60 mal mehr Context Switches schafft als der alte Scheduler.

Zwischen 25 und 100% liegen die Leistungssteigerungen bei fork()-Tests, da im neuen Scheduler Kindprozesse erstmal auf dem gleichen Prozessor wie der Elternprozess laufen, was zeitraubende SMP-Interlocks überflüssig macht.

Man darf also auf den 2.6er Kernel gespannt sein, der sich in jeder Situation durch das flüssigere Scheduling positiv bemerkbar machen sollte.

Literatur

[1] Interview mit Ingo Molnar, `http://www.kerneltrap.com/node.php?id=517`

[2] Goals, Design and Implementation of the new ultra-scalable O(1) scheduler, `linux/Documentation/sched-design.txt`

[3] NUMA architectures and user level scheduling - a short introduction, Christoph Koppe,
`http://www4.informatik.uni-erlangen.de/Projects/FORTWIHR/ELiTE/numa_ult_intro.html`

[4] `http://lwn.net/2002/0110/a/scheduler.php3`