

Scheduling im Linux Kernel

Stefan Schick

13. Januar 2011

Inhaltsverzeichnis

- 1 Allgemeines
- 2 Scheduling Classes
- 3 Completely Fair Scheduler
- 4 Quellen

History

- 1.0 - verkettete Liste mit lauffähigen Prozessen
- 2.0 - SMP Support
- 2.5 - $O(1)$ Scheduler - Ingo Molnar
- 2.6 - Rotating Staircase Deadline Scheduler - Con Kolivas
- 2.6.23 - Completely Fair Scheduler - Ingo Molnar

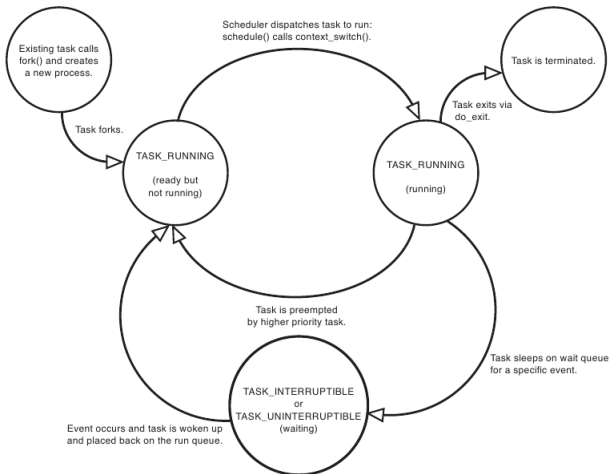
Ziele

- geringe Latenz
- wenig Overhead
- hoher Durchsatz

Tasks

- Linux kennt keine Threads, jeder Thread wird als eigener Prozess behandelt
- im Scheduler werden Prozesse als Tasks bezeichnet
- struct task_struct in include/linux/sched.h
- pid, parent, children, cpumask, sched_entity, nice

Taskstates



Task erzeugung

- Copy on Write - Parent und Child verwenden zunächst gleiche Speicherseiten, erst bei einem Schreibzugriff wird eine Seite kopiert
- `fork()` - kopiert `task_struct` usw. für child und weist neue pid zu
- `exec()` - lädt neues Executable in den Prozess Kontext und führt dieses aus

Cooperative vs. Preemptive Multitasking

- cooperative Multitasking - Task muss Prozessor von sich aus wieder freigeben
- preemptive Multitasking - Scheduler kann aktuell laufenden Task verdrängen
- → Linux ist preemptive

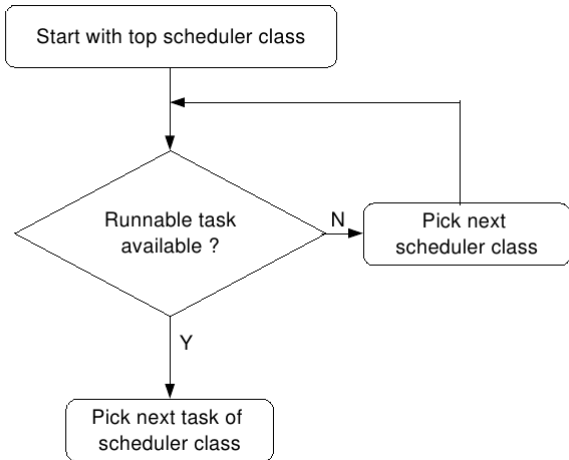
I/O vs. Processor bound

- I/O bound - Prozess schläft meiste Zeit. Muss nach aufwachen aber sofort abgearbeitet werden. Läuft kurz aber oft.
- Processor bound - Prozess läuft lange, schläft selten.

Scheduler Classes

- `rt_sched_class`
 - Kümmert sich um `SCHED_FIFO/RR` Tasks
 - $O(1)$ Prioritäts Array
- `cfs_sched_class`
 - Kümmert sich um `SCHED_NORMAL(Linux)/SCHED_OTHER(Posix)` Tasks
 - $O(\log(n))$ Red-Black-Tree
- `idle_sched_class`
 - Kümmert sich um `idle` Tasks

Scheduler Decision



Die Idee

- Die ideale Multitasking CPU bearbeitet jeden Task parallel mit dem gleichen Bruchteil an CPU Leistung
- jeder Task bekommt $1/n$ der CPU-Zeit, wobei n die Anzahl der lauffähigen Prozesse ist
- z.B. 4 Tasks laufen parallel, somit bekommt jeder Task 25% der CPU Leistung
- somit gibt es keinen Zustand in dem irgendein Task mehr von der CPU bekommt als der Rest

Completely Fair Scheduling

- in der Realität läuft jeder der 4 Tasks für ein viertel der Zeit mit 100% CPU Leistung
- modelliert diese CPU indem festgehalten wird wie unfair ein Task relativ zu den anderen Tasks behandelt wird
- auf einer idealen CPU wäre diese Unfairness immer 0
- sie entsteht sobald mehr Tasks als CPUs vorhanden sind
- wenn ein Task läuft erhöht sich die CPU Zeit die dieser allen anderen Tasks schuldet

Timeslices

- Die Zeit die ein Task läuft bis er preempted wird nennt man Timeslice
- lange Timeslice - schlechte Interaktive Performance, hohe Latenz
- kurze Timeslice - viele Context-Switches - viel Overhead
- CFS hat keine Timeslices im klassischen Sinn
- Die CPU Zeit die ein Prozess bekommt ist eine Funktion basierend auf der Auslastung des Systems modifiziert durch nice values
- Nanosekundengenaue abrechnung

Prioritäten

- nice -20 bis +19
- wie nett ein prozess zu einem anderen ist
- Prozesse mit hohen nice-values bekommen einen kleineren Anteil der CPU
- Prozesse mit niedrigen nice-values bekommen einen größeren Anteil der CPU

vruntime

- die vom jeweiligen Task verbrauchte CPU Zeit gewichtet durch den nice-Wert
- die Zeit wird in Nanosekunden angegeben und ist entkoppelt von Timer Ticks
- ist in `sched_entity` welches zum `task_struct` gehört in `include/linux/sched.h` definiert
- der Task mit der geringsten `vruntime` darf als nächstes laufen
- bei overflow $vruntime \bar{v}runtime - min_vruntime$
- `vruntime` wird periodisch und wenn ein Task seinen Taskstate ändert geupdatet
- dabei wird berechnet wie lange der Task schon lief und wie lange er noch laufen sollte

Runqueues

- jede CPU hat eine eigene Runqueue
- Runqueues werden als Red-Black-Trees organisiert
- Self balancing Binary Search Tree

Taskstate Änderungen

- Neue Tasks
 - $vruntime = min_vruntime$
- aufgewachte Tasks
 - werden von der waiting list in die Runqueue eingefügt
 - $vruntime = MAX(old_vruntime, min_vruntime - sched_latency)$

Quellen

- Linux Kernel Development 3rd Edition / Robert Love / 2010
- BFS vs. CFS - Scheduler Comparison / Groves, Knockel, Schulte / 2009
- The Completely Fair Scheduler / Thomas Gleixner / 2008
- The Linux „Completely Fair Scheduler“ / Ben Nayer
- <http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/>
- [git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git](https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git)

Fragen

