Beispiel: 4.3 BSD UNIX Scheduler

Der Scheduler des 4.3 BSD UNIX Betriebssystems benutzt eine dem Round-Robin Verfahren überlagerte Prioritätensteuerung. Bei jedem Prozeßwechsel sucht der Dispatcher die nicht-leere Run-Queue mit höchster Priorität und startet den am längsten wartenden Prozeß.

Die Zeitscheibenlänge, die ein Prozeß zugewiesen bekommt, beträgt eine Zehntelsekunde.

Prozesse haben eine Priorität zwischen 0 **(hoch)** und 127 **(niedrig)**.Die Basispriorität *PUSER* für Nutzerprozesse hat den Wert 50.

Die Variable *p_usrpri* enthält die Prozeßpriorität, die vom Dispatcher ausgewertet wird.

Runqueue-Verwaltung

Der Dispatcher benutzt 32 Run-Queues, in die die Prozesse gemäß ihrer Priorität eingeordnet werden.
Die Nummer der Run-Queue wird gemäß

$$rq := \left\lfloor \frac{p\_usrpri}{4} \right\rfloor$$

bestimmt.

Dynamische Prioritäten

Grundidee ist, daß CPU-intensive Prozesse an Priorität verlieren und kurze oder E/A-intensive Prozesse Priorität gewinnen.

Die Priorität eines Prozesses wird verändert, falls

1. der Prozess länger als eine Sekunde blockiert auf I/O gewartet hat: Priority Boosting
2. der Prozeß rechnet. Dann verliert der Prozeß an Priorität.
3. Zusätzlich wendet der BSD Scheduler einmal pro Sekunde einen Vergissfilter auf die Prioritäten an.

Für das Scheduling benutzt der 4.3 BSD Scheduling für jeden Prozeß die Variablen *p_cpu* und *p_nice*:
Die Variable *p_cpu* wird vom UNIX-Kernel verwaltet und enthält eine Schätzung für die vom Prozeß aufgenommene CPU Leistung. Der Inhalt von *p_nice* kann vom Besitzer eines Prozesses festgelegt werden und darf ganzzahlige Werte zwischen -20 und $+20$ enthalten. Positive Werte bedeuten, daß der Prozeß bereit ist, weniger als den ihm zustehenden Anteil am Prozessor zu erhalten. Negative Werte erhöhen den Anteil am Prozessor. Wird kein Wert für *p_nice* angegeben, so wird $p\_nice = 0$ gesetzt.

Die Priorität eines Prozesses hängt dynamisch von seiner bisher verbrauchten CPU–Zeit ab. Zu jedem Time Tick (alle 10 ms) wird für den in der CPU laufenden Prozeß die Variable p_cpu inkrementiert:

$$p\_cpu := p\_cpu + 1.$$

Alle 4 Time Ticks wird die Priorität aller Prozesse gemäß

$$p\_usrpri := min\{PUSER + \frac{p\_cpu}{4} + 2 \cdot p\_nice, 127\}$$

neu bestimmt, wobei $PUSER$ eine Konstante mit dem Wert 50 ist.

Vergissfilter

Damit die Prioritäten nicht ständig wachsen, wird der Wert p_cpu von allen Prozessen einmal pro Sekunde durch einen Vergissfilter korrigiert:

$$p\_cpu := \frac{2 \cdot load}{2 \cdot load + 1} \cdot p\_cpu + p\_nice.$$

Dabei bezeichnet *load* die durchschnittliche Anzahl der rechenbereiten Prozesse (inklusive des rechnenden Prozesses) über ein vorangegangenes 1-Minuten-Intervall.

Ablaufsteuerung unter 4.4 BSD UNIX

Jedem Prozess sind **zwei** Prioritäten zugeordnet: Eine im User-Mode und eine im Kernel-Mode.

1. Benutzerprozesse laufen i.d.R. im User-Mode mit Basispriorität PUSER = 50. Die Priorität im User-Mode wird im Feld `p_usrpri` im Prozeßleitblock gemerkt. Der Wert liegt im Bereich 50-127.
2. Bei einem Systemaufruf wechselt der Prozeß in den Kernel-Mode und wird i.d.R. blockiert, weil er auf ein Ereignis wartet (I/O, sem_wait(), waitpid(), ...). Im `p_priority` Feld im Prozeßleitblock wird **die dem Ereignis zugeordnete Priorität** gemerkt.
3. Tritt das Ereignis ein, wird der Prozeß in die zu `p_priority` gehörende Warteschlange einsortiert (Priority Boosting). Seine Basispriorität ändert sich dabei nicht! Nach Ablauf der Zeitscheibe wird er wieder gemäß der Basispriorität eingereiht.

Table 4.2 Process-scheduling priorities.

Priority	Value	Description
PSWP	0	priority while swapping process
PVM	4	priority while waiting for memory
PINOD	8	priority while waiting for file control information
PRIBIO	16	priority while waiting on disk I/O completion
PVFS	20	priority while waiting for a kernel-level filesystem lock
PZERO	22	baseline priority
PSOCK	24	priority while waiting on a socket
PWAIT	32	priority while waiting for a child to exit
PLOCK	36	priority while waiting for user-level filesystem lock
PPAUSE	40	priority while waiting for a signal to arrive
PUSER	50	base priority for user-mode execution

Quelle:

Marshall Kirk McKusick, Keith Bostic, Michael J. Karels and John S. Quarterman: *The Design and Implementation of the 4.4BSD Operating System, Addison Wesley, 1996.*

Bemerkung:

1. Das BSD Schedulingverfahren ist responsiv dank Priority Boosting.
2. Prozesse mit geringer Priorität können verhungern (nicht starvation-free).
3. Verfahren skaliert nicht:
    1. Anpassen der Priorität aller Prozesse alle 4 Time Ticks.
    2. Anpassen von p_cpu aller Prozesse einmal pro Sekunde gemäß dem Vergissfilter.

Beispiel: Windows/NT

Prioritäten-basiertes unterbrechendes Thread-Scheduling

- Zeitscheibenlänge:
  NT Workstation: 30 ms Windows 2000 Professional: 20 ms
  NT Server: 180 ms Windows Uniprocessor Server: 120 ms
- Strategie:
  1) rechenbereiter Thread höchster Priorität läuft
  2) nach Ablauf der Zeitscheibe: Round-Robin
- Unterschied zu UNIX
  1) Ein Thread höherer Priorität unterbricht den laufenden Thread.
  2) Der unterbrochene Thread wird an den Anfang seiner Warteschlange gestellt.
  3) Verbrauchte CPU-Zeit geht nicht in die Berechnung der Prioritäten ein.

Thread-Zustände

- ready: rechenbereit
- running: rechnend
- standby: Der Thread, der als nächster rechnen soll, kommt in den Zustand “standby“. Windows 2K prüft, ob die Priorität des Standby-Threads größer als die Priorität des laufenden Threads ist. Falls ja, wird der laufende Thread unterbrochen.
- waiting: blockiert
- transition: Das Ereignis, auf das der Thread gewartet hat, ist eingetreten, aber der Prozeß ist aktuell noch nicht rechenbereit.

  Beispiel: Ein Thread wartet auf Eingabe. Während der Wartezeit wird der Stack ausgelagert.

Wiederholung: Windows/NT-Terminologie: Mehrere Threads innerhalb eines „Prozesses“.

Process object

Object body attributes	**Services**
Process ID	Create process
Security descriptor	Open process
Base priority	Query process information
Default processor affinity	Set process information
Quota limits	Current process
Execution time	Terminate process
I/O counters	
VM operation counters	
Exception/debugging ports	
Exit status	

Windows 2000 Process Object Attributes

Process ID	A unique value that identifies the process to the operating system.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that the process's threads have performed.

Security Descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Exeption/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception.
Exit status	The reason for a process's termination.

Thread object

Object body attributes	**Services**
Thread ID	Create thread
Thread context	Open thread
Dynamic priority	Query thread information
Base priority	Set thread information
Thread processor affinity	Current thread
Thread execution time	Terminate thread
Alert status	Get/Set context
Suspension count	Suspend/Resume
Impersonation token	Alert thread
Termination port	Test thread alert
Thread exit status	Register termination port

Windows 2000 Thread Object Attributes

Thread ID	A unique value that identifies a thread when it calls a server.
Thread context	The set of register values and other volatile data that defines the execution state of a thread.
Dynamic priority	The thread's execution priority at any given momer
Base priority	The lower limit of the thread's dynamic priority.
Thread processor affinity	The set of processors on which the thread can run, which is a subset or all of the processor affinity of the thread's process.
Thread execution time	The cumulative amount of time a thread has executed in user mode and in kernel mode.

Alert status	A flag that indicates whether the thread should execute an asynchronous procedure call.
Suspension count	The number of times the thread's execution has been suspended without being resumed.
Impersonation token	A temporary access token allowing a thread to perform operations on behalf of another process (used by subsystems).
Termination port	An interprocess communication channel to which the process manager sends a message when the thread terminates (used by subsystems).
Thread exit status	The reason for a thread's termination.

$$\mathit{ThreadBasePriority} := \mathit{ProcessBasePriority} + \{-1, -2, 0, 1, 2\}$$

i.d.R. für Benutzerprozesse:
$\mathit{ThreadBasePriority} := \mathit{ProcessBasePriority} = 8$

Die Base Priority hängt von der geerbten Priority Class ab:
Real-Time, High, Above Normal, Normal, Below Normal, Idle

Innerhalb einer Priority Class wird weiter unterteilt in (sogenannte Relative Priority): Time-Critical (Saturation), Highest (2), Above-Normal (1), Normal (0), Below-Normal (-1), Lowest (-2), Idle (Saturation)

Saturation steht für den höchsten bzw. kleinsten Wert, der innerhalb der Priority Class möglich ist.

Die Base Priority hängt von der geerbten Priority Class und der Relative Priority ab:

	Real-Time	High	Above Normal	**Normal**	Below Normal	Idle
Time-Critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above-Normal	25	14	11	9	7	5
Normal	24	13	10	**8**	6	4
Below-Normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

Priority Boosting: Die Prioritäten der Threads werden in gewissen Situationen „kurzfristig" erhöht.

Windows-Feature: Priority Boosting **mit/ohne** Variable Quantums

Der Quantum Value Wert wird aus der Variable Quantum Table gemäss des Quantum Index (bzw. auch Priority Separation genannt) ermittelt:

	Short Quantum Index			Long Quantum Index		
Index	0	1	2	3	4	5
Variable	6	12	18	12	24	36
Fixed	18	18	18	36	36	36

1. Nach I/O:

Ereignis	*boost*
Disk	1
CD-ROM	1
network	2
keyboard	6
mouse	6
sound	8

Einem Thread sind eine *base prio* und eine *current prio* zugeordnet.

$$current\_prio\_neu := min\{baseprio + boost, 15\}$$

Nach jeder Zeitscheibe wird die Priorität (*current prio*) um eins erniedrigt, bis die *base prio* wieder erreicht ist.

2. Threads einer interaktiven Sitzung:
base prio = 8
boost = 6 für ein Quantum (Zeitscheibe)
aber: Quantum := 3× Quantum

Bem.: Die Boost-Werte sind nur *Empfehlungen*. Treiberentwickler können davon abweichen.

Bemerkung:

1. Konstanter Aufwand für Einhängen und Herausholen aus dem Runqueue-Array (Für jede Priorität eine Runqueue).
2. Das NT-Schedulingverfahren ist responsiv dank Priority Boosting. Ein *geboosteter* Thread bekommt schnell die CPU, da ein Thread höherer Priorität den laufenden Thread unterbricht.
3. Starvation vermeiden: Damit Prozesse mit geringer Priorität nicht verhungern, wird periodisch der Balance Set Manager gestartet. Der Balance Set Manager sucht jede Sekunde nach Threads, die länger als 4 s im Zustand rechenbereit sind, um deren Priorität für eine Zeitscheibe zu erhöhen:
   $current\_prio\_neu := 15$, wobei das Quantum Value auf 3 gesetzt wird[10].
   Der damit verbundene Overhead wird begrenzt: Es werden maximal 16 Threads geprüft und maximal 10 Prioritäten erhöht.

[10] entspricht Halbierung der Zeitscheibe

Windows NT unterstützt SMP-Maschinen (Symmetric Multiprocessing):

- eine globale Warteschlange,
- Affinity_Scheduling: „Prozessor" bevorzugt bei der Auswahl des nächsten zu rechnenden Threads solche, die schon von ihm gerechnet wurden. $\Longrightarrow$ effiziente Cache-Nutzung!

Zusammenfassung: Windows Scheduling

- Prioritäts-basiertes unterbrechendes Thread-Schedulingverfahren
- 32 Prioritätsklassen: [0-15] dynamische Prioritäten, [16-31] feste Prioritäten
- Zeitscheibe (Quantum) wird als Bruchteil eines Clock Ticks realisiert[11]: One-third of a clock tick
- benutzt Priority Boosting und Quantum-Stretching
- Windows Scheduling ist überparametrisiert.

[11] Wegen Backward Compability zu Versionen vor Windows Vista

Beispiel: Linux

- Linux verwaltet und plant Threads.
- Diese werden als Tasks bezeichnet und mittels der Datenstruktur `task_struct` verwaltet. Jeder Thread besitzt eine eigene Task-Datenstruktur.

  Beispiel: Ein Prozess mit 1 Thread wird mittels einer Task-Datenstruktur verwaltet.

  Ein Prozess mit n Threads wird mittels n Task-Datenstrukturen verwaltet.
- Um POSIX-konform zu bleiben, besitzen alle Tasks einer Anwendung (d.h. alle Threads eines Prozesses) dieselbe Prozess-ID (PID).

  Beispiel: Das Kommando
  > kill <pid>
  terminiert alle Threads mit der angegeben PID.

- Timesharing-Prozesse bekommen dynamische Prioritäten zwischen 100-139.
- Linux benutzt **statische** Prioritäten für sogenannte Echtzeitprozesse. Diese haben die höchsten Prioritäten 0-99. Es rechnet jeweils der rechenbereite Thread mit der höchsten Priorität.
  - Real-Time FIFO: Non-preemptive Scheduling-Verfahren. **Ausnahme:** Falls ein Thread mit höherer Priorität als der laufende rechenbereit wird, wird der laufende Thread unterbrochen und der Thread höherer Priorität darf rechnen.
  - Real-Time Round Robin: Zeitscheibenverfahren
    Auch hier läuft jeweils der Thread höchster Priorität.
    Aber bei mehreren Threads gleicher Priorität wird ein Zeitscheibenverfahren mit festem Quantum angewandt.

Achtung: Statische Prioritäten sind nur eine Voraussetzung zur Implementierung von Echtzeitsystemen, aber nicht hinreichend!

Linux-Scheduler - Historie

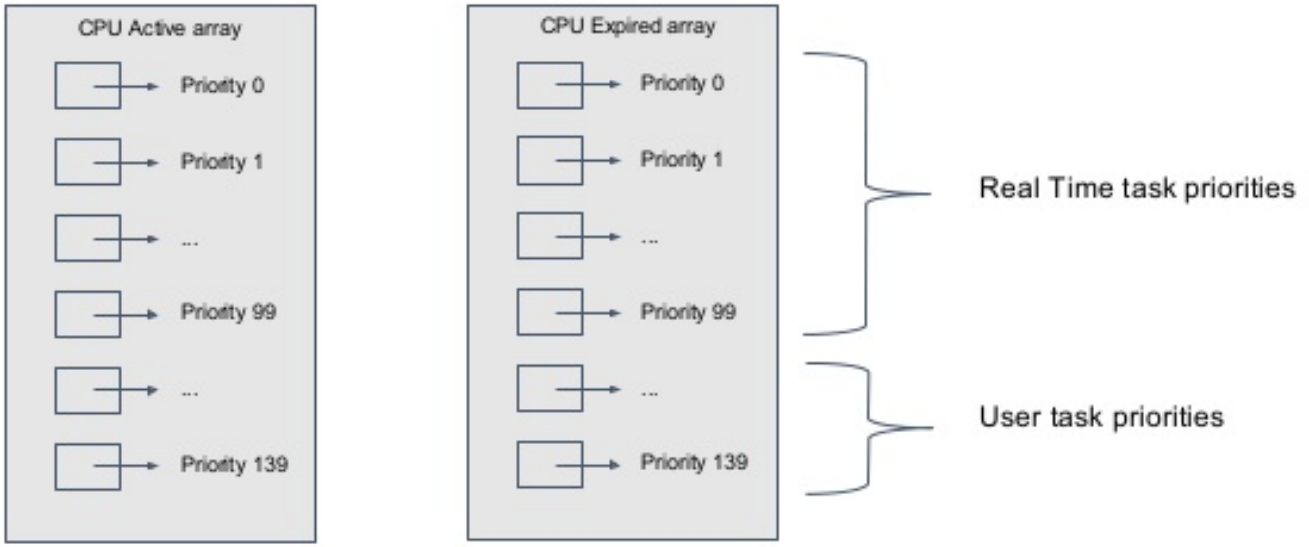
Linux 1.0: Simple verkettete Liste der lauffähigen Prozesse. Scheduler Entscheidung O(n).

Linux 2.5: O(1) Scheduler von Ingo Molnar (01/2002). Nach Priorität sortiertes Array von Runqueues pro CPU (SMP-Support, CPU-Affinität).

Kennzeichen:

- Enqueue and dequeue of tasks and next task selection in constant time,
- bitmap of 140 Bits to find highest non-empty priority queue,
- verbrauchte CPU-Zeit geht nicht in die Prioritätenberechnung ein,

The O(1) scheduler (Cont..)



Quelle: Viresh Kumar: *The Linux Kernel Scheduler (For Beginners)*, 2017

- Vermeiden von Starvation: Pro CPU werden zwei Arrays mit Runqueues verwaltet: Active und Expired. Pro *Epoche* kann jede Tasks nur eine vorgegebene Zeit rechnen, danach gilt sie als *expired* und wird in die zugehörige Expired-Runqueue eingekettet. Wenn das Active-Array leer ist, werden die Arrays getauscht.

Linux 2.6.23: Completely Fair Scheduler (CFS) von Ingo Molnar (04/2007).

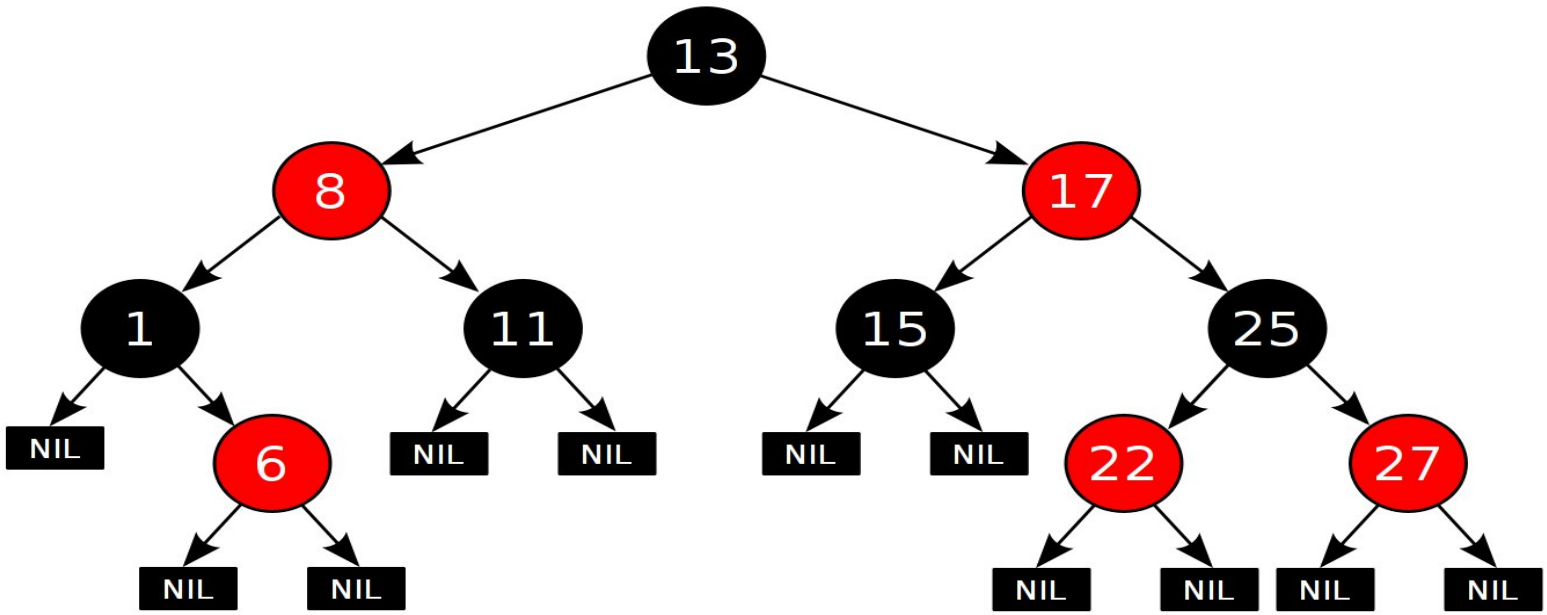
Angebliche Motivation: Bessere Performance auf Desktops, d.h. bessere Unterstützung von interaktiven Prozessen

Kennzeichen:

1. Neue Datenstruktur: Keine verketteten Listen mehr, sondern Red-Black-Tree.
2. Task Group Scheduling:
   Tasks einer Gruppe werden gemeinsam geschedult. Damit wird Fairness erreicht, wenn eine Task viele weitere Tasks erzeugt.

Red-Black-Tree (rbtree.h)

- Selbstbalancierter binärer Suchbaum mit Worst-Case-Komplexität O(log(n)) (Rudolf Bayer, 1972)
- Tasks sortiert durch Depth-First-Suche von links nach rechts
- meist Zugriff auf linkes Element (cached)



entnommen Wikipedia

Funktionsweise CFS - Überblick

- Scheduling Entity (SE) kann Task oder Task-Group sein
- Red-Black-Tree ist nach *vruntime* geordnet: die bisher verbrauchte, gewichtete CPU-Zeit
  $\Longrightarrow$ Starvation vermeiden
- wenn *current* nicht mehr left-most Node ist (und Granularity-Zeit (Zeitscheibe) überschritten), wird *current* verdrängt.
- CFS versucht *den Idealfall* (?) zu erreichen, dass alle SEs gleiche *vruntime* haben
- *vruntime* wird auf *min_vruntime* gesetzt, wenn Task in die Runqueue (RB-Tree) kommt.
- Einfügen in Red-Black-Tree: O(log(n))
  Ermitteln des nächsten zu rechnenden Threads: Zeiger auf das Element ganz links: O(1)

Einfluß der Priorität:

- *vruntime* wird um Laufzeit/*nr_running* gewichtet nach *prio* erhöht:
  $\Longrightarrow$ Zeit vergeht für Prozesse geringerer Priorität schneller!
  Prozesse höherer Priorität haben *besseren vruntime*-Wert und werden bevorzugt.
  $\Longrightarrow$ keine festen Zeitscheiben: 10 ms - 5120 ms
- `nice`-Kommando

Scheduling auf Multicores:

In multicore environments the implementation of the scheduler becomes substantially more complex. Scalability concerns dictate using per-core runqueues. The motivation for per-core runqueues is that upon a context switch the core would access only its local runqueue, when it looks for a thread to run.[12]

- Seit Linux 2.0: SMP Support (Symmetrisches Multiprozessorsystem): Eine Runqueue pro CPU. $\Longrightarrow$ Eine *Runqueue* pro Core.
  Motivation: Affinity Scheduling: Thread soll wieder dort laufen, wo er schon einmal gerechnet hat. Damit soll eine bessere Cache-Nutzung erreicht werden.
- Linux kennt spezielle Systemaufrufe um Affinität zu spezifizieren.
- **Periodisch** wird CPU-Loadbalancing durchgeführt, sofern gesetzte Affinität nicht verletzt wird.

[12] https://people.ece.ubc.ca/sasha/papers/eurosys16-final29.pdf

Aber das Load-Balancing zwischen den Runqueues war zunächst nicht optimal, siehe:
Lozi, Jean-Pierre; Lepers, Baptiste; Funston, Justin; Gaud, Fabien; Quema, Vivian; Fedorova, Alexandra: *The Linux Scheduler: A Decade of Wasted Cores*, EuroSys 2016. (PDF)

"As a central part of resource management, the OS thread scheduler must maintain the following, simple, invariant: ***make sure that ready threads are scheduled on available cores.*** *As simple as it may seem, we found that this invariant is often broken in Linux. Cores may stay idle for seconds while ready threads are waiting in runqueues.*

$\Longrightarrow$ Vorschläge aus dem Paper führten 2016 zum Patch des Linux-Schedulers hinsichtlich seiner Performanz auf Multicoresystemen

Verwaltung blockierter Prozesse:

- Jedem Ereignistyp wird eine eigene Warteschlange zugeordnet.
- Veränderungen an einer dieser Warteschlangen findet im Rahmen der Interruptbehandlung statt.

  Problem: Es können nebenläufige schreibende Operationen auf der Warteschlange auftreten!

  Beispiel: Dies kann auf Multiprozessorsystemen, aber auch wenn eine Linuxkernelkomponente von einer anderen unterbrochen wird, auftreten.

  $\Longrightarrow$ Veränderungen an den Warteschlangen muss *synchronisiert* werden (mittels Spinlock).

Zusammenfassung

1. Analog zum BSD-Scheduling macht Linux prioritäten-basiertes Scheduling.
2. Es wird nur eine Warteschlange benutzt, die als Red-Black-Tree organisiert ist.
3. Der Red-Black-Tree ist nach der berechneten *vruntime* sortiert.
4. In die Berechnung der *vruntime* geht die vom Elternprozeß geerbte Priorität, der nice-Wert und - analog zu BSD - die bisher verbrauchte CPU-Zeit ein.
5. Einfügen in Red-Black-Tree: O(log(n))
   Ermitteln des nächsten zu rechnenden Threads: O(1)
6. Die *vruntime* eines Prozesses wird nur neu berechnet, nachdem er gerechnet hat.
7. Kein explizites Priority Boosting, sondern implizit.
8. Da die bisher verbrauchte CPU-Zeit in die Berechnung der *vruntime* mit eingeht, verschlechtern sich Prozesse, die rechnen. $\Longrightarrow$ vermeidet Starvation

Quellen:

- M. Tim Jones: *Inside the Linux 2.6 Completely Fair Scheduler - Providing fair access to CPUs since 2.6.23*,
- Wolfgang Mauerer: Professional Linux Kernel Architecture, John Wiley & Sons (Wrox), 2008
- Andrew S. Tanenbaum und Herbert Bos: Moderne Betriebssysteme, 4. Auflage, 2016.