Realtime Linux

- Begriffsdefinitionen: Hard-/Soft- Realtime, Latency und Jitter
- RTAI (RealTime Application Interface)
  - Mikro- vs. Nanokernel
  - seL4 microkernel
- Linux RT (preemtable Kernel)
  - Standard Kernel und RT-patches
  - Linux Schedule Policies
  - Locking Techniken (Semaphoren,Mutex..)
  - High-Res Timer
  - POSIX/Linux Capabilities

Harald Nikolisin

Kaum ein anderer Begriff in der Computertechnologie wurde so oft misinterpretiert wie der des *Realtime Computing*.

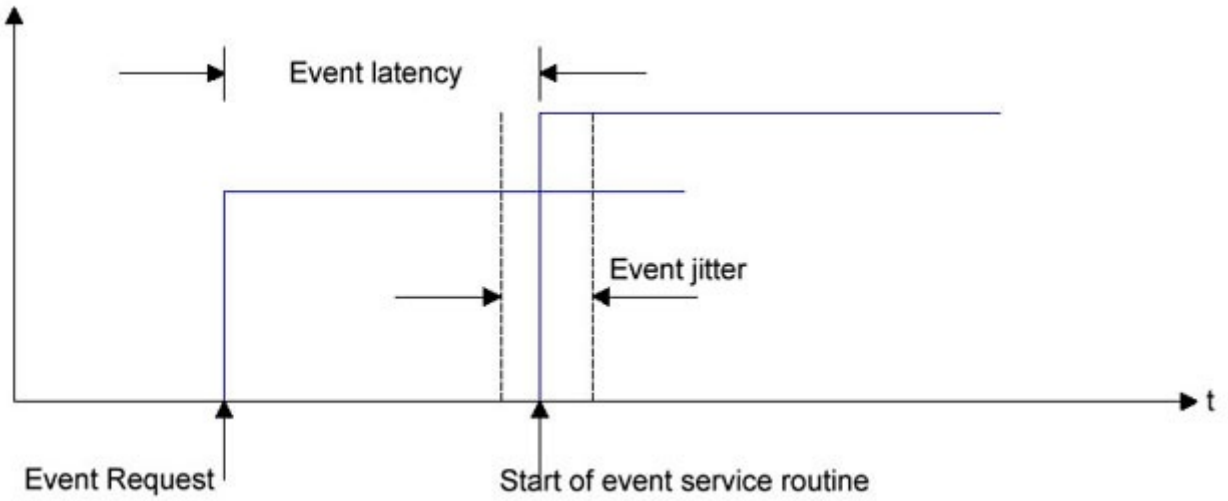
Ein Realtime System (RT System) hat festgelegte Fristen (**operational deadlines**) zwischen der Auslösung eines Ereignisses und der Reaktion darauf.

Wie groß die Frist ist spielt dabei keine Rolle - ein Realtime System muss daher nicht zwangsläufig schnell oder performant sein - es muss nur festgelegte Antwortzeiten garantieren können.
Bei der Art dieser Garantie unterscheidet man **Hard-** vs. **Soft RT-Systeme**.
Bei den harten RT-Systemen ist eine **Überschreitung** der Antwortfrist **inakzeptabel** und führt zum Systemversagen, Beispiele sind kritische Regelungssteuerungen.
Weiche RT-Syteme sind wesentlich ungenauer definiert. Man erwartet zwar eine Antwort innerhalb einer gegebenen Zeitspanne, ein Überschreiten führt aber nicht zwangsläufig zu einem irreperablen Schaden.

Die Dauer zwischen dem Auftreten der Stimulierung bis zur Ausführung der Reaktion nennt man **Latency**.
Die Variation des Zeitpunktes der Reaktion nennt man **Jitter**.

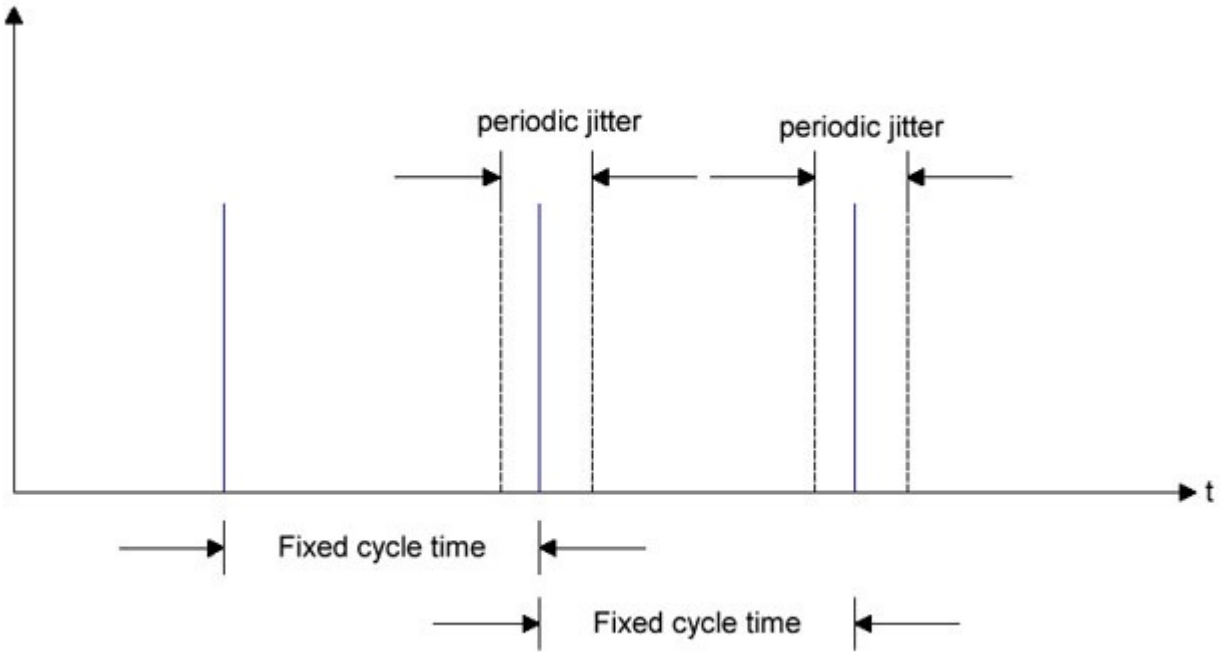
Hard RT-Systeme tendieren zu keinem Jitter und zu einer Ausführung der Reaktion nach einer festgelegten Zeit.
Bei Soft RT-Systemen tritt meistens ein Jitter auf, so dass die Ausführungsfristen einfach größer bemessen werden.

Latenzzeit (Latency)



Jitter

ist die Ungenauigkeit der Zeitpunkte von Ereignissen, welche oftmals periodisch auftritt.



RTAI vs. Linux-RT

RTAI (RealTime Application Interface)

- Kernel in Kernel System
- Harte Echtzeit möglich
- RT Entwickler müssen spezielle Bibliotheken benutzen
- Open Source
- RT Programm im Kernel Space bei Mikrokernel, sonst im User Space

Linux-RT (preemption)

- Standard Kernel (ev. mit Patches)
- Nur weiche Echtzeit möglich
- RT Entwickler benutzen Standard API's
- OpenSource
- RT Programm im User Space

RTAI Überblick

- Früher mit **RT-HAL** Mikrokernel
- wg. patentrechtlichen Problemen nun mit **ADEOS** Nanokernel
- auf nicht-x86 Plattformen weiterhin RT-HAL Mikrokernel

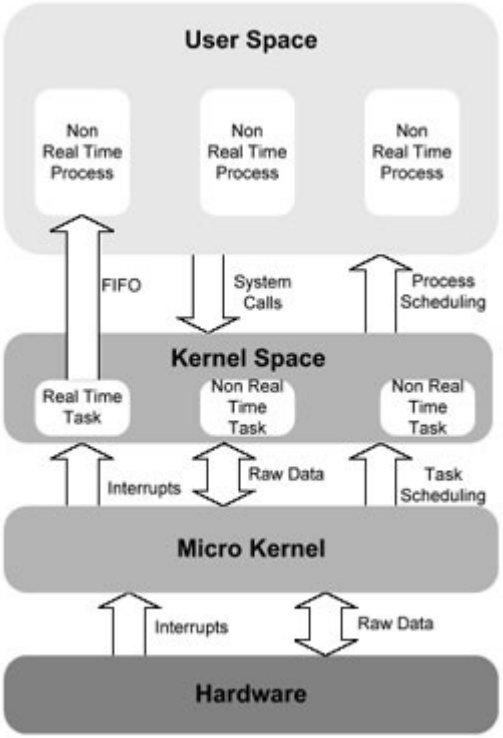
RTAI Subsystem besteht vornehmlich aus:

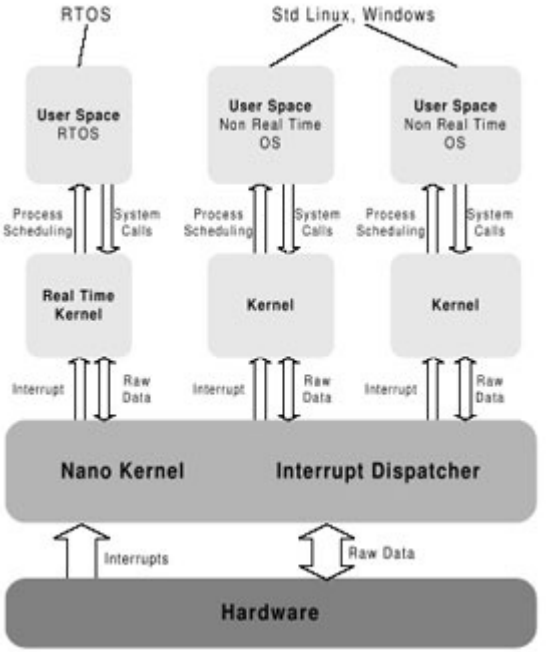
- Echtzeit Scheduling
- Interprozess Kommunikation (IPC)
- Verarbeitung von Echtzeit Interrupts durch die ADEOS Schnittstelle

Die fertige **ELinOS** Distribution besteht aus:

- ADEOS nanokernel
- RTAI 3.1 als Echtzeitdomäne
- Durch das RTAI Subsystem wird die rtai_hal domäne hinzugefügt.
- Kernel 2.6 mit ADEOS patch als Linux Wirtsdomäne

Vergleich Mikro- vs. Nanokernel





seL4 – L4 microkernel

- microkernel mit 10kLOC
- gleichzeitig hypervisor
- minimierte TCB (*trusted computing base*)

<u>der seL4 microkernel besteht nur aus:</u>

- Virtual Memory
- Interprozess Kommunikation (IPC) & Threads
- Verarbeitung von Echtzeit Interrupts durch die ADEOS Schnittstelle

<u>Zielsetzungen</u>

- seL4 versteht sich nur als microkernel und nicht als OS
- technisch gesehen ähnelt es einem Nanokernel
- Zugriffsmöglichkeit nur über capabiltes, welche einem Objekt zugeordnet sind und daher nicht mit Linux Capabilites (ACL like) gleichzusetzen sind
- Gesamte Codebasis auf Bugfreiheit geprüft
- Sichert Vertraulichkeit, Integrität und Verfügbarkeit zu (daher RT-fähig).

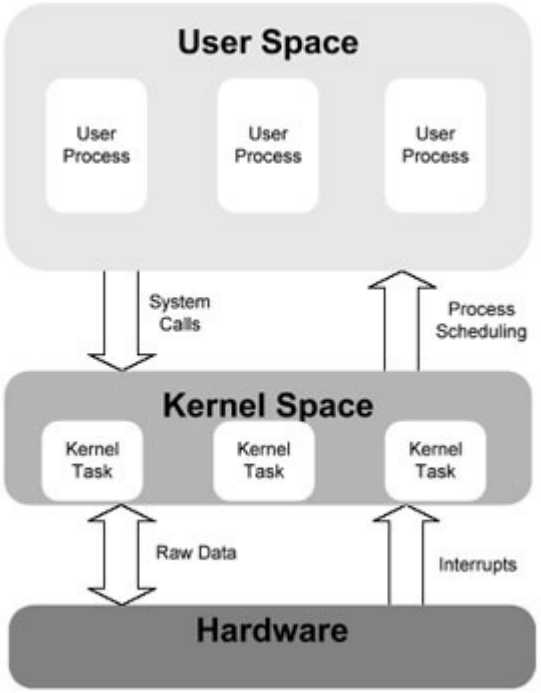
Linux RT Überblick

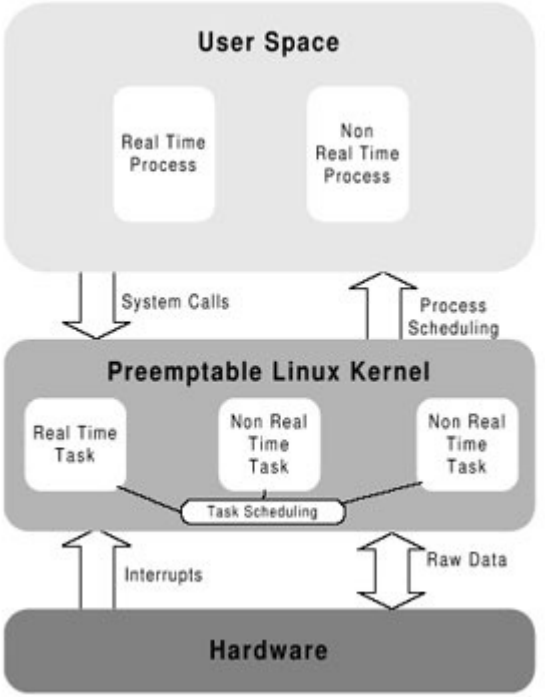
- Der Standard (*Vanilla*) Kernel enthält mittlerweile alle RT Features
- Diese müssen natürlich in der Kernel Konfiguration angeschaltet werden
- Bei älteren Kernel kann man RT Patches (Molnar & Gleixner) anwenden

Der Vanilla Kernel **2.6.39** enthält mittlerweile alle RT Features:

- Deterministic scheduler
- Preemption support
- PI Mutexes (Priority Inversion)
- HRT (High-Resultion Timer)
- Preemptive Read-Copy Update
- IRQ Threads
- Voller RT Preemption Support
- kein **B**ig **K**ernel **L**ock mehr

Änderungen beim Preemptable Kernel





Unter **Preemption** versteht man in einem Multitasking Betriebssystem das Aussetzen eines laufenden Prozesses in Bevorzugung eines anderen.

- Im User-Space auf allen UNIX Betriebssystemen möglich
- Im Kernel-Space nur bei modernen Betriebssystemen, wie Linux seit Kernel 2.6
- Linux Scheduler Classes sind: **C**omplete **F**air **S**cheduler mit einem **O(log n)** Algorithmus und der RealTime Scheduler, die beide die **runqueues** pro CPU bilden.
- Im Gegensatz zum alten O(1) scheduler gibt es keine Processor- und I/O-bound Heuristiken, welche Timeslices durch Prioritätsänderungen beeinflussen.
- POSIX 1003.1c-1995: Multithreaded Programming (für Linux sind Threads Prozesse)

- Linux Scheduler beherrscht sowohl **Soft-** als auch **Hard Affinity**
- Ohne manuellen Eingriff Balance zw. Lastverteilung und gering. Prozessorwechsel
- Hard Affinity durch `task_struct.cpus_allowed` und `sched_setaffinity()`

Linux Scheduling

Seit 2007 (Kernel **2.6.23**) gibt es unter Linux ein **scheduling Framework**, welches auf oberster Ebene mit Prioritätsklassen arbeitet. Mit Stand 2013 gibt es 4 Prioritätsklassen, welche von oben nach unten abgearbeitet werden.

1. *stop_sched_class*

Super tasks, welche alle anderen unterbrechen können, aber selber nicht unterbrochen werden können, bspw. Systemaufrufe bzgl. Hotplug oder CPU-affinity.

2. *rt_sched_class*

Realtime tasks unterteilt in 99 Prioritätsstufen.

3. *fair_sched_class*

„Normaler“ Prioritätsbereich des **C**omplete **F**air **S**cheduler, dessen tasks über den sog. *nice* Wert priorisiert werden können.

4. *idle_sched_class*

Gibt es in den oberen Klassen keinen lauffähigen Prozess, so wird der sog. *IDLE-task* aktiviert.

Linux Scheduling Policies

SCHED_NORMAL

Standard Prozess (fair_sched_class) ohne RT Prioritäten, aber mit NICE Wert

SCHED_BATCH

Standard Prozess (fair_sched_class), welcher keine SCHED_NORMAL Prozesse unterbrechen kann.

SCHED_FIFO

FIFO klassifizierter Prozess (rt_sched_class) läuft solange, bis er
- blockiert
- sched_yield() aufruft – hat keinen Effekt solange kein RT-Prozess mit mindestens gleicher Priorität exisitert.
- Von einem mindestens gleich priorisiertem Prozess preempted wird

FIFO klassifizierte Prozesse laufen – sofern Sie die höchst priorisierten Prozesse auf dem System sind - solange Sie möchten. Interessant ist der FIFO Scheduler bei gleich priorisierten Prozessen. Ein neuer Prozess wird immer an den Anfang der Prozessliste geschoben.

SCHED_RR

RoundRobin klassifizierte Prozesse (rt_sched_class) bekommen vom Scheduler eine Timeslice zugewiesen und werden an das Ende der Prozessliste, der jeweiligen Priorität angehängt.
Bei unterschiedlich priorisierten Prozessen gibt es keinen Unterschied zu FIFO Prozessen.

Posix.1b (IEEE Std 1003.1b-1993)

Dieser Posix Standard bzgl. Prozessprioritäten bildet die Basis für die Mechanismen im Prioritäts Scheduling.
Linux besitzt 2 Prioritätsbereiche – die ehemals zum ersten Prioritätsbereich gehörende dynamische Priorität des damaligen O(1) schedulers ist obsolet.

1. Statische Priorität = Nice Value (erster Prioritätsbereich)

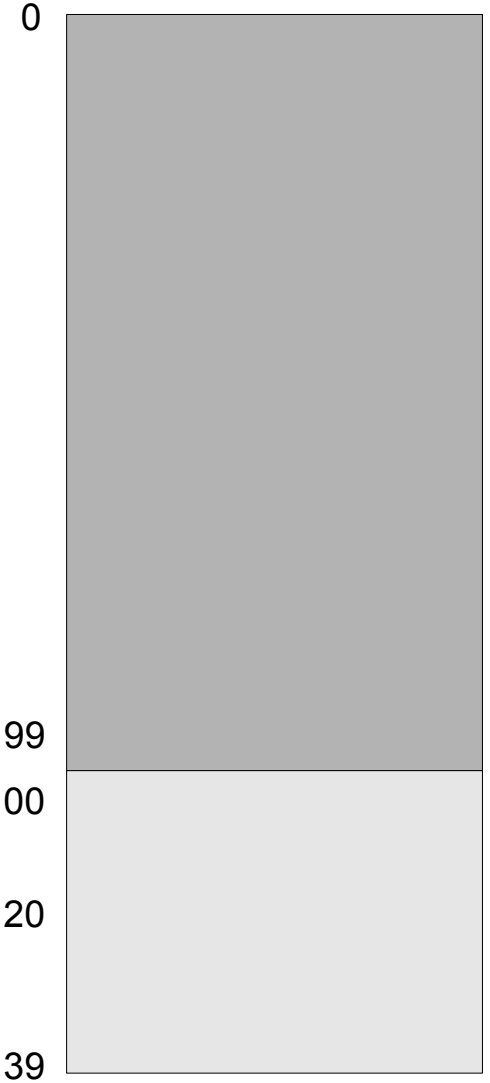
Nice = -20 höchste Priorität
Nice = +19 niedrigste Priorität
Der **C**omplete **F**air **S**cheduler **(fair_sched_class)** gewichtet die verfügbaren Prozessorzeiten. Die Gewichtung, welche durch den Nice Wert verändert wird, bestimmt dann wiederum die Timeslices, die dem Prozess zur Verfügung steht.
Nur User mit **CAP_SYS_NICE** capability dürfen NICE < 0 setzen (0 = Standardwert)

2. RealTime Priorität (zweiter Prioritätsbereich)

POSIX schreibt mindestens 32 Abstufungen vor, Linux hat 99 (höhere Number = höhere Priorität). Die *scheduler class* **sched_rt** übernimmt die Verwaltung dieser Tasks.

Posix.1b (IEEE Std 1003.1b-1993)

Gesamtpriorität - gibt bspw. **`cat /proc/<ID>/sched`** aus – 0 ist die höchste Priorität, 139 die niedrigste



RT-Prioritätsbereich

- erfordert SCHED_FIFO/SCHED_RR
- unter Linux im Bereich 0 – 99
- auch hier inverse Nummerierung!
  sched_setparam(11) ergibt: Gesamtprio: 88
- keine statische Prioritäten (via nice)

„nice“ Prioritätsbereich

- erfordert SCHED_NORMAL/SCHED_BATCH
- unter Linux im Bereich 100-139
- nice=0 führt zur default prio 120

Locking Techniken im Kernel Space

Spin Locks

- **Nicht preemptable** - daher nur für kurze Operationen geeignet! Länge der Spin Locks ist daher equivalent zur Systemlatency
- Wenig Overhead, bei weniger als zwei Prozesskontextwechsel bevorzugt
- Kann im Interrupt Handler eingesetzt werden
- Kann nicht schlafengelegt werden

Semaphoren

- Semaphoren im Linux Kernel sind *schlafende Locks*
- Wenn ein Prozess eine Semaphore wieder freigibt, wird der erste auf diese Semaphore wartende Prozess aufgeweckt
- Kann wg. des Schlafmechanismus nicht im Interrupt Handler eingesetzt werden
- **Ist preemptable**, daher v.a. längere Locks geeignet
- Es kann nur eine Semaphore (**Mutex** = Mutual Exclusion) oder mehrere vergeben werden

Big Kernel Lock (BLK)

Altlast des Systems: Globaler Spin Lock des Kernels, der schlafen gelegt werden kann - arbeite daher nur im Prozesskontext
-> ausgebaut in den RT-Patches (Molnar,Gleixner), oder seit Vanilla Kernel 2.6.39

High Resolution Timer (HRT)

- Benötigt entsprechende Hardware auf dem Chipsatz

High Precision Event Timer (HPET)

..ersetzt/ergänzt *Programmable Interval Timer* (PIT) und *Real-Time Clock* (RTC)

- Diese laufen mit mindestens 10MHz

Auf Softwareseite kommen die sog. **POSIX Timer** zum Einsatz

- Bieten nanosekunden Auflösung an

4 POSIX Timer stehen unter Linux zur Verfügung:

- CLOCK_MONOTONIC
- CLOCK_REALTIME
- CLOCK_PROCESS_CPUTIME_ID
- CLOCK_THREAD_CPUTIME_ID

CLOCK_PROCESS../CLOCK_THREAD.. benutzen unter **x86** das **Timestamp Counter Register** (TSC), weswegen sie in der Auflösung der CPU Taktung laufen!

CLOCK_MONOTONIC/CLOCK_REALTIME benutzen den HPET

Dieser ist aber an die sog. Jiffies gebunden, das bedeutet:

- Läuft das System mit 250Hz (Kernel **CONFIG_HZ**) – hat ein Jiffy 0.004 sec.
- Läuft das System *tickless* (Kernel **CONFIG_NO_HZ**) – ist das Ergebnis des Posix Timers die Genauigkeit des HPET (höchstens 1ns: POSIX Beschränkung)
- **Tickless Kernel** also wesentlich genauer (im Vanilla Kernel seit 2.6.21)

POSIX/Linux Capabilities

Der Superuser ***root*** besitzt alle Rechte des System. Dennoch ist es möglich mit den sog. Capabilities - die aus dem nicht veröffentlichten POSIX Standard 1003.1e stammen und von Linux erweitert wurden – spezifische Rechte wahrzunehmen ohne den Prozess mit *euid* root zu starten.

Folgende Tabelle zeigt einige wichtige Capabilites.
Alle Capabilites sind in folgendem Header definiert:
`/usr/include/linux/capability.h`

CapabilityR	**Beschreibung**
CAP_SYS_NICE	nice Value bis -20 verringern, RT-Priorität im vollen Bereich nutzen und CPU Affinität fremder Prozesse bestimmen
CAP_IPC_LOCK	beliebig großen Speicherbereich im RAM locken
CAP_NET_RAW	Sockets vom Typ RAW (IpV4) und Pakete (Ethernet Frames) benutzen
CAP_SYS_RESOURCE	Ressourcen Limits verändern

Die Capabilites werden entweder von Kernel Prozessen geändert oder im Userland mit den libcap-progs (setcap,getcap) auf Filesystem Ebene definiert. Dazu muss der Kernelparameter CONFIG_SECURITY_FILE_CAPABILITIES auf yes gesetzt sein. Auf Novell Systemen muss auch der (Kernel) Bootparameter file_caps=1 sein.

Resource Limits

Privilegierte Instanzen können auch Systemlimits (Resource Limits) setzen

Eine privilegierte Instanz ist jemand mit CAP_SYS_RESOURCE Attribut, i.d.R. der ***root*** User, der über alle Capabilites verfügt.

Resource Limits	**Beschreibung**
RLIMIT_AS	max. Größe des Adressraumes [Bytes]
RLIMIT_CPU	max. Verbrauch CPU Zeit [s]
RLIMIT_MEMLOCK	max. Anzahl von Bytes die im Speicher gelockt werden kann
RLIMIT_NICE	Maximalwert, bis zu der ein Prozess sein nice-Value abmindern kann (normalerweise 0)
RLIMIT_RTPRIO	max. RT Priorität

Weitere Mechanismen

Speicher

Swapping unbedingt vermeiden

- Prefaulting data in den physikalischen Speicher via **`mmap()`**
- Speicherbereich schützen mit **`mprotect()`**
- Locking data im physikalischen Speicher mit **`mlock()`**

Prozessor

So ausgefeilt auch der Multitasking Ansatz auch ist. Nicht jeder Prozess der sich in einer kritischen Region des Kernels befindet, kann sofort unterbrochen werden. Latenzen sind unvermeidbar und je nach Kernelstand unterschiedlich. Bestimmte Prozesse wie CD automout polling oder Änderung des X Server Status erhöhen die Latenz zusätzlich.

Ein Effektiver Ansatz ist daher das abschirmen (shielding) eins Prozessorkerns für den RT-Prozess.
Dafür wird bereits im Bootprozess (System V Init) eine CPU Core aus dem verfügbaren Pool herausgenommen.

CPU Shielding & Affinity

- Unter **shielding** versteht man das Abschirmen einer CPU gegenüber Zugriffen, d.h. der Kernel darf diese weder für kernel- oder userland Prozesse benutzen, noch Interrupt Routinen darauf laufen lassen.
- Wenn man dann einen Prozess exklusiv an eine CPU bindet, was bedeutet dass er nur auf dieser laufen darf, nennt man dies **affinity**.

Sowohl shielding als auch affinity bekommen nicht als Parameter nicht eine CPU, sondern ein sog. CPU-Set, welches aus einem oder mehreren Prozessoren bestehen kann.

Implementierung auf Standard Distributionen:

Shielding nur über Bootparameter welcher dem **init** Prozess ein CPU-Set entzieht
Affinit über POSIX Methode `sched_setaffinity()`

Beispielimplementierung auf einer RT Distribution: *RedHawk Linux*

- `shield` command erlaubt on-the-fly shielding zur Laufzeit
- zusätzlich Shieldingsspezifierung: Prozesse,Interrupts,lokale Interrupts
- `run` command erlaubt affinity für jeden beliebigen Prozess
- zusätzliche Methode `mpadvise()` mit filigranen Auswahlmöglichkeiten eines CPU-Sets

Quellen

Linux Kernel Development
Robert Love; Novell Press

Linux System Programming
Robert Love; O'Reilly

Advanced Programming in the UNIX Environment
W. Richard Stevens, Stephen A. Rago; Addison Wesley Professional

Red Hawk Linux User's Guide
Concurrent 2008

http://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html

http://rt.wiki.kernel.org

http://www.ibm.com/developerworks/linux/library/l-cfs/index.html