Echtzeit - Linux

Dipl.-Inform. Stefan Riesner

Standard-Linux vs. Echtzeit

- Linux Kernel verwendet grobgranulare Synchronisation
- Keine Unterbrechung von Standard Tasks während Systemaufrufen
- Höherpriorisierte Tasks können von niederpriorisierten blockiert werden
- Hintergrund Hardware-Optimierungsstrategien (Speichermanagement etc.)
- ...
- Linux 2.6.x mit stark verbessertem Kernel ermöglicht hochwertige Soft-Echtzeit Anwendungen.

Historisches

- Wichtigste Linux-Echtzeiterweiterungen seit 1996 (Linux 2.0.25):

  RTLinux: M. Barabanov (Master-Thesis) am New Mexiko Institute of Technology (NMT)

  RTAI: ***Real-Time Application Interface***
  Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano (DIAPM)

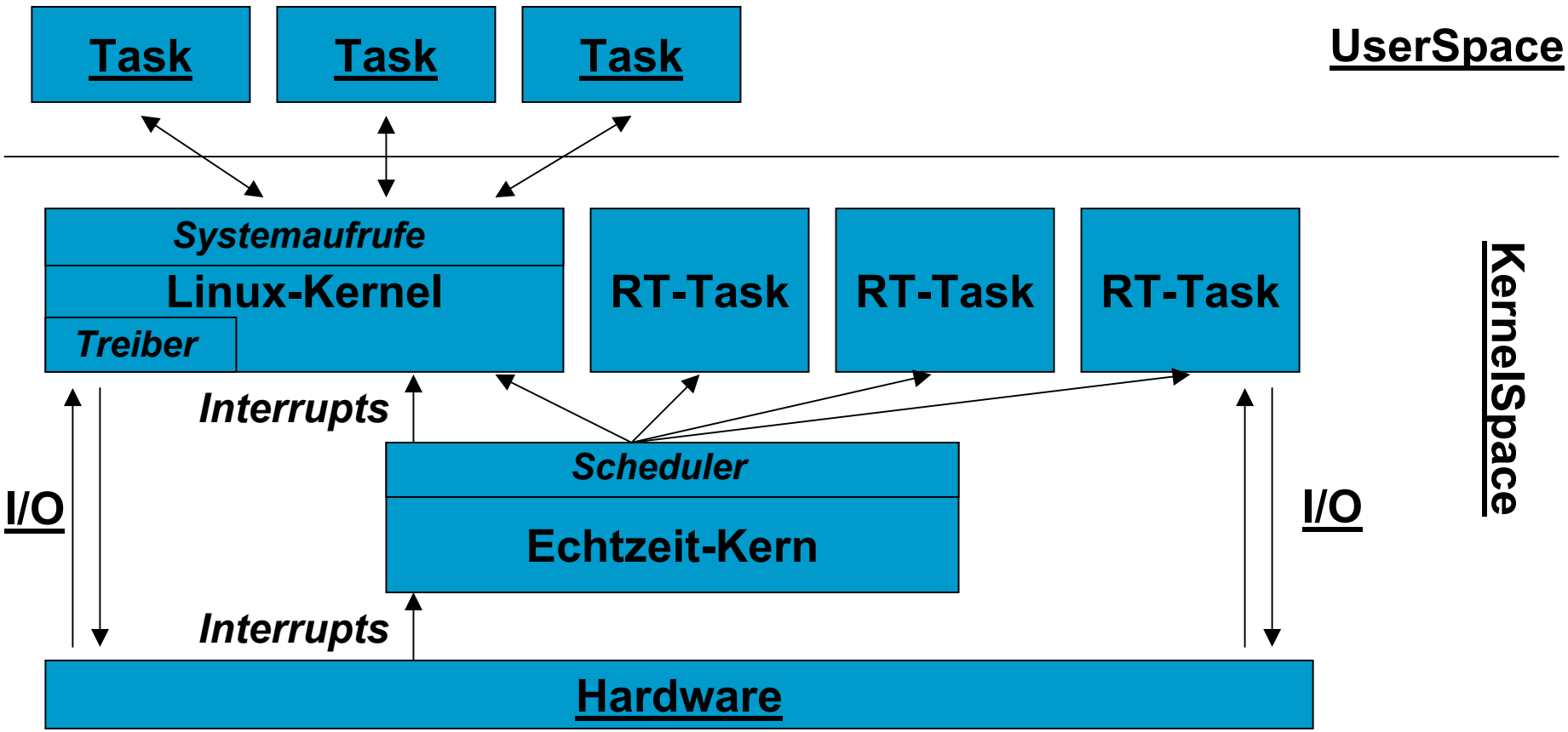
- Beide Systeme sind sehr ähnlich bei Linux 2.0.x, Unterschiede seit 2.2.x

- RTLinux nun patentiert mit kommerziellem Support (www.fsmlabs.com)

- RTAI wird (nach Aussprache) vom RTLinux-Patent nicht berührt. LGPL 2.
  (www.aero.polimi.it/~rtai/index.html)

Design

Grundidee: Das normale Linux läuft als Idle-Task eines neuen Schedulers. System-Interrupts werden zunächst vom Echtzeitkern behandelt.



Gemeinsamkeiten / Unterschiede von RTAI und RTLinux

- RTLinux verändert Linux-Kernel-Methoden für den Echtzeiteingriff
  => Kernel-Versions-Änderungen haben große Auswirkungen

- RTAI fügt Hardware Abstraction Layer (HAL) zwischen Hardware und Kernel ein. Hierzu sind nur ca. 20 Zeilen Code am Originalkern zu ändern. HAL selbst umfasst kaum mehr als 50 Zeilen => Transparenz

- RTAI ist frei, RTLinux in freier (Privat, Ausbildung) und kommerzieller Version

- Beide Ansätze verwenden ladbare Kernel Module für Echtzeittasks

RTAI – Programmierung (1)

RTAI-(RTLinux-)Echtzeitmodule müssen typische Struktur eines Kernel Moduls besitzen, d.h. insb. die Funktionen:

```
int init_module(void)
{ /* Aufgerufen bei insmod */
   ...
}

void cleanup_module(void)
{ /* Aufgerufen bei rmmod */
   ...
}
```

zur Initalisierung bzw. dem Aufräumen.

RTAI – Basismodule (Auszug)

rtai	Grundlegendes RTAI Framework
rtai_sched	Periodisches und *One-Shot* Scheduling
rtai_shm	Gemeinsame Datenbereiche für Echtzeit Tasks und / oder Nichtechtzeit Anwendungen (inkl. Semaphore)
rtai_fifos	First-In / First-Out Datenpuffer zur Kommunikation zwischen Echtzeit Tasks und / oder Echtzeit und Nichtechtzeit Anwendungen über Unix-Filesystem
rtai_pqueue	Kernel-Safe Message Queues
lxrt	User-Space Soft-/Hard Echtzeitmodule !!!
rt_com	Echtzeittreiber für serielle Schnittstelle

...

RTAI – Anwendung

Beispielstartsequenz:

```
insmod rtai

insmod rtai_sched

insmod rtai_fifos

insmod mytask.o
```

Beispielendsequenz:

```
rmmod mytask

rmmod rtai_fifos

rmmod rtai_sched

rmmod rtai
```

RTAI – Programmierung (2)

- Um auf die Funktionen der Module zuzugreifen zu können sind die passenden Include-Anweisungen in den C-Programmcode aufgenommen werden, z.B:

```
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
```

- Periodische Tasks werden in RTAI-Modulen durch den Aufruf von

```
int rt_task_init(RT_TASK *task, void (*rt_thread)(int), int data, int stack_size,
                 int priority, int uses_fpu, void (*signal)(void))
```

initalisiert, aber noch nicht gestartet.

rt_thread stellt Pointer auf periodisch zu **bedienende** Funktion dar.

```
#include <rtai.h>
#include <rtai_sched.h>

static RT_TASK mytask;

static void mythread(int data)
{ ...
}

int init_module(void)
{ rt_task_init(&mytask, mythread, 3, 2000, 2, 1, 0);
  /* int data = 3, 2000 Bytes Stack, Priorität 2 (0=höchste),
     FPU ON (1), ContextSwitch-Handler NONE (0)
  */
  ...
}
```

RTAI – Programmierung (3)

Zum Starten des Task ist ein Timer einzurichten

```
RTIME tick_period = start_rt_timer(nano2count(TICK_PERIOD));
```

und der Task mittels

```
rt_task_make_periodic(&mytask, rt_get_time() + tick_period, tick_period);
```

periodisch endgültig zu starten (beides i.d.R. in init_module).

- TICK_PERIOD entspricht der gewünschten Zykluszeit in Nanosekunden.
- Die Funktion nano2count dient der Umrechnung in RTAI-interne Darstellung.
- rt_get_time() ermittelt aktuelle Systemzeit.

RTAI – Programmierung (4)

Innerhalb der Task-Funktion ist der Ablauf an geeigneten Programmstellen mittels der Funktion rt_task_wait_period() bis zum nächsten Zyklus an den Scheduler zurückzugeben:

```
static void mythread(int data)
{ while(1)
   {  /* Periodisch auszuführender Code */
      rt_task_wait_period();
   }
}
```

Zum Beenden des Task im cleanup:

```
void cleanup_module(void)
{  rt_task_delete(&mytask);
   stop_rt_timer();
}
```

Zusammenfassender Vergleich RTAI vs. RTLinux

	RTLinux	RTAI
POSIX komp.	1003.13, 1003.1b, 1003.1c	1003.1b (pqueues only) 1003.1c
IPC	Shared memory, FIFO, Mailboxes	Shared memory, FIFO, Mailboxes, MessageQueues, RPC-net,
Scheduler	EDF, Rate-Montonic	EDF, Rate-Monotonic, RR
User-Space Echtzeit	Software Interrupts	LXRT API im UserSpace
Strategie	Kernel Modifikation	RTHAL

Ausführlicher Vergleich in www.opengroup.org/rtforum/apr2001/slides/kuhn.pdf