VxWorks

Vertreter eines klassischen Realzeitbetriebssystems mit einer starken Marktdurchdringung ist VxWorks der Firma Wind River. Ursprünglich als Realzeitbetriebssystem für einfache (16-Bit-)Mikrocontroller entwickelt, unterstützt VxWorks heute 32- und 64-Bit-Prozessoren. VxWorks gilt als ein skalierbares, deterministisches, zuverlässiges Realzeitbetriebssystem, das einen breiten Hardware-Support mitbringt. Der Mikrokernel benötigt wenige Ressourcen, das System ist dank guter Portierbarkeit für unterschiedliche Plattformen verfügbar. Die *Wind River Workbench* genannte Entwicklungsumgebung auf dem Host basiert auf Eclipse. Mit MILS gibt es eine VxWorks-Variante, die ihren Fokus auf Security legt, die Variante CERT legt den Fokus auf Safety.

5.6 Realzeitarchitektur auf Multicore-Basis

Eine weitere Möglichkeit, Realzeitanforderungen einzuhalten, bieten Mehrprozessor- bzw. Mehrkernmaschinen. Auf diesen Rechnern wird eine CPU allein für die Realzeitaufgaben reserviert. Bereits ein moderner Linux-Kernel bietet diese Möglichkeit an.

Aufgabe des Systemarchitekten ist es zunächst, die abzuarbeitenden Tasks in Realzeit- und Nichtrealzeittasks zu klassifizieren. Dabei müssen die zeitlichen Parameter der Prozesse und die sich ergebende Gesamtauslastung bestimmt werden. Die Gesamtauslastung muss gemäß erster Realzeitbedingung unterhalb von `Cores * 100%` liegen. Um deterministisches Zeitverhalten garantieren zu können, müssen ebenso Interrupts einzelnen CPU-Kernen exklusiv zugewiesen werden. Das muss jedoch durch die Firmware (Bios) unterstützt werden.

Typischerweise wird die Boot-CPU (die CPU mit der Nummer `0`) als die CPU eingeplant, die nicht Realzeitprozesse abarbeitet. Das hat oft hardwaretechnische Gründe, denn nicht bei jeder Hardware lassen sich Interrupts beliebig auf unterschiedliche Prozessorkerne verteilen.

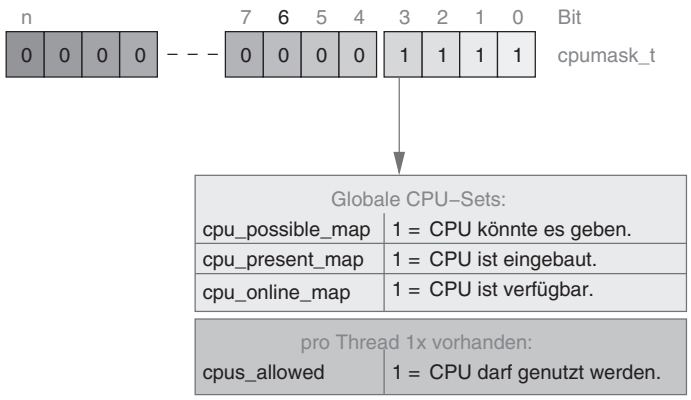


Abbildung 5-3 *Datenstruktur zur Verwaltung von mehreren Prozessorkernen*

Jedem Rechenprozess kann eine Affinität mitgegeben werden. Hierbei handelt es sich um die Kennung, auf welchen Prozessoren ein Prozess abgearbeitet werden darf (siehe Abbildung 5-3). Die Affinität lässt sich über die Funktion `sched_setaffinity()` programmtechnisch realisieren (siehe Abschnitt 4.2.3).

Beim weiteren Vorgehen gibt es zwei Möglichkeiten:

1. Alle Tasks, die keine zeitlichen Anforderungen haben, werden auf CPU 0 verlagert. Die Gesamtauslastung für diese CPU muss dabei unterhalb von 100 Prozent liegen. Die Realzeitprozesse werden auf den übrigen Prozessoren abgearbeitet, wobei der Scheduler selbst die Verteilung auf die einzelnen Prozessoren vornimmt.
2. Alternativ dazu kann der Systemarchitekt die Rechenprozesse direkt den einzelnen Rechnerkernen zuordnen.

In jedem Fall muss nach der Verteilung auf die Prozessoren für jeden Prozessor beziehungsweise Prozessorkern getrennt ein Realzeitnachweis geführt werden.

In Linux gibt es außerdem für die Zuordnung von Rechenprozessen auf Prozessoren das *cgroup*-Framework (Container- oder Control-Group) [KuQu08]. Dieses hilft Tasks (mit spezifischen Eigenschaften) und deren zukünftige Kindprozesse hierarchisch zu organisieren. Der Admin gruppiert über ein virtuelles Filesystem (type `cpuset`) sowohl Prozessoren als auch Speicherressourcen (*Memory Nodes*) und verteilt auf diese die Tasks. In Abbildung 5-4 ist zu sehen, dass diese Verteilung hierarchisch aufgebaut ist. Die oberste Gruppe enthält sämtliche Prozessoren und Memory Nodes. In einer Ebene darunter lassen sich die Ressourcen auf mehrere Untergruppen verteilen. Per Flag *cpu-exclusive* und *memory-exclusive* lässt sich noch festlegen, ob eine Ressource von mehreren Gruppen aus verwendet werden darf (überlappend) oder exklusiv der einen Gruppe zur Verfügung steht.

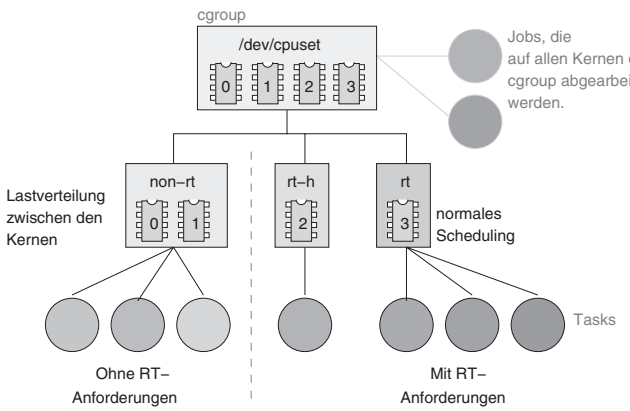


Abbildung 5-4
Hierarchische Prozessorganisation

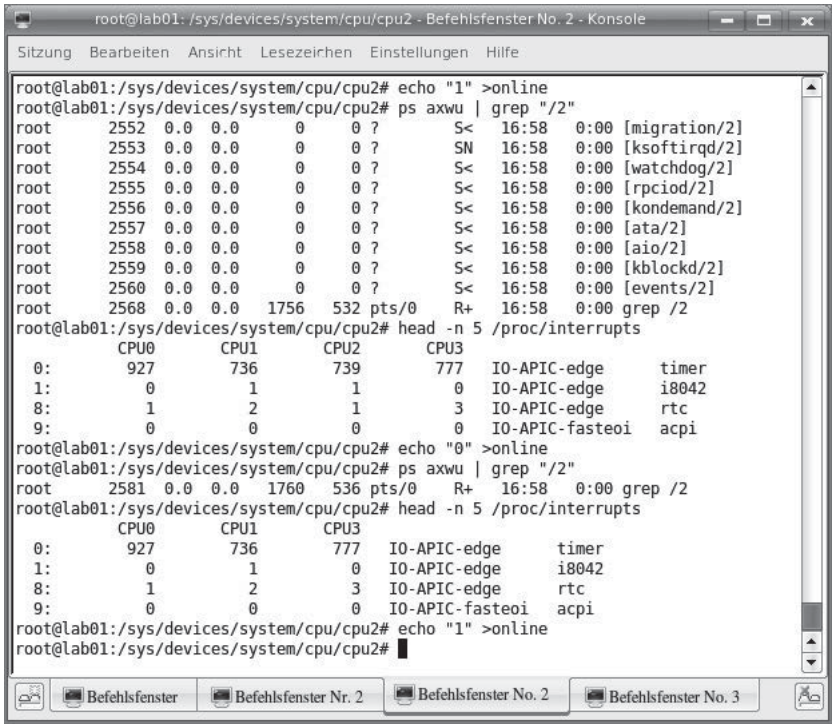
Die Verlagerung eines Rechenprozesses in eine Gruppe führt dazu, dass das Attribut `cpus_allowed` des verschobenen Prozesses mit den gruppenspezifischen Einstellungen (Affinity-Maske der Gruppe) überschrieben wird. Auf diese Weise lassen sich recht einfach (und dauerhaft) die zeitkritischen Prozesse von den weniger zeitkritischen trennen. Im einfachsten Fall legen Sie zwei Gruppen (*rt* und *non-rt*) an, weisen diesen die Speicherressourcen und die Prozessorkerne zu und verschieben sämtliche Prozesse in die *non-rt*-Gruppe. Anschließend picken Sie sich die zeitkritischen Tasks heraus und migrieren diese in die *rt*-Gruppe (siehe Beispiel 5-1).

Beispiel 5-1
Verteilung von Tasks auf Rechnerkerne

```
root@lab01:~# mkdir /dev/cpuset
root@lab01:~# mount -t cpuset -ocpuset cpuset /dev/cpuset
root@lab01:~# mkdir /dev/cpuset/rt
root@lab01:~# mkdir /dev/cpuset/non-rt
root@lab01:~# echo 0 > /dev/cpuset/rt/mems
root@lab01:~# echo 0 > /dev/cpuset/non-rt/mems
root@lab01:~# echo 2 >/dev/cpuset/rt/cpus
root@lab01:~# echo 1 >/dev/cpuset/rt/cpu_exclusive
root@lab01:~# echo 0-1 >/dev/cpuset/non-rt/cpus
root@lab01:~# echo 1 >/dev/cpuset/rt/cpu_exclusive
root@lab01:~# for pid in $(cat /dev/cpuset/tasks); \
> do /bin/echo $pid > /dev/cpuset/non-rt/tasks;\
> done
root@lab01:~# cd /dev/cpuset/rt
root@lab01:~# /bin/echo $$ >tasks
root@lab01:~# starte_rt_task
...
```

Wenn Sie die Verteilung einmal vorgenommen haben, sollten Sie davon absehen, nachträglich die Affinity-Maske einer Gruppe zu ändern. Das hat nämlich keine Auswirkung mehr auf den innerhalb der Gruppe befindlichen Prozess. Um eine veränderte Parametrierung wirksam werden zu lassen, müsste der Rechenprozess der entsprechenden Gruppe neu zugeordnet werden. Das Aus- oder Einbrechen aus dem Container-Gefängnis (cgroup) über die Funktion `sched_setaffinity()` ist aber dennoch nicht möglich. Nur wenn ein Prozessorkern zur Gruppe gehört, kann der Prozess dorthin migrieren.

Abbildung 5-5
Deaktivieren von Rechnerkernen



Durch das Verschieben der Prozesse in die non-rt-Gruppe wird der für Realzeitaufgaben vorgesehene Prozessor freigeschaufelt. Das lässt sich einfacher per CPU-Hotplugging bewerkstelligen. Per Software können einzelne Prozessoren (und damit auch Prozessorkerne) deaktiviert und später auch wieder aktiviert werden. Beim Deaktivieren werden dann die auf dem Prozessor ablaufenden Rechenprozesse auf die anderen Prozessoren migriert. Das Feature wird auch sinnvoll eingesetzt, wenn der Verdacht besteht, dass ein einzelner CPU-Kern defekt ist. Die CPU-0 spielt als Boot-Prozessor eine Sonderrolle und lässt sich auf vielen Systemen nicht deaktivieren. Das hängt damit zusammen, dass bei diesen Architekturen einige Interrupts fest an den ersten Prozessorkern gekoppelt sind. Abbildung 5-5 zeigt, wie Sie durch Zugriffe auf das Sys-Filesystem eine CPU aktivieren und deaktivieren. Sichtbar am Inhalt der Datei `/proc/interrupts` sind zunächst vier Kerne online. An der Prozesstabelle lassen sich die auf dieser CPU lokalisierten Kernel-Threads entdecken. Wird die CPU jetzt deaktiviert (Kommando `echo "0" >online`), taucht die CPU in der Datei `/proc/interrupts` nicht mehr auf. Auch die cpu-spezifischen Kernel-Threads sind verschwunden. Wenn Sie bei einer CPU die Datei `online` nicht vorfinden, bedeutet das nur, dass sich diese CPU auch nicht deaktivieren lässt.

Ähnlich wie Tasks besitzen auch Interrupts im Kernel ein ihnen zugewiesenes Attribut, welches die Prozessoren in Form der Bitmaske auf die Rechnerkerne festlegt, auf denen die zugehörige Interrupt-Service-Routine abgearbeitet werden darf. Dieses lautet im Kernel relativ einfach nur `mask`. Geht es um Hardware-Interrupts, kann der Kernel-Programmierer beim Einhängen der Interrupt-Service-Routine festlegen, dass die ISR nicht »balanced«, also auf unterschiedlichen Rechnerkernen abgearbeitet wird. Vom Userland aus kann das Attribut `mask` wieder einmal über das Proc-Filesystem manipuliert werden. Im folgenden Beispiel wird auf einer Vierkern-Maschine (smp_affinity=0x0f) die CPU-0 von der Verarbeitung des Interrupts 1 ausgenommen (smp_affinity=0x0e):

```
root@lab01:# cat irq/1/smp_affinity
0f
root@lab01:# echo "0x0e"> irq/1/smp_affinity
root@lab01:# cat irq/1/smp_affinity
0e
```

Die Abarbeitung von Threads und Interrupts durch die vorhandenen Prozessoren lässt sich mit den beschriebenen Methoden sehr gut kontrollieren. Softirqs, Tasklets und Timer, aber auch Workqueues sind hingegen anders gelagert. Typischerweise werden Softirqs und Tasklets auf der CPU abgearbeitet, auf der die Interrupt-Service-Routine angestoßen wurde. Hier muss der Systemarchitekt sorgfältig die IRQ-Affinität für einen mit harten Realzeitanforderungen versehenen Prozess auf die für Realzeitaufgaben reservierte CPU legen. Da Interrupts heutzutage allerdings gemeinsam (unterschiedliche Hardware löst den gleichen Interrupt aus) genutzt werden, muss der Konstrukteur dafür sorgen, dass die zugehörige Interrupt-Leitung exklusiv nur von dem einen Kern genutzt wird. In solch einem Fall ist ein Eingriff in das PCI-Handling erforderlich, was die Portabilität erschwert.

Diese moderne Realzeitarchitektur setzt eine Multicore-Hardware voraus. Sie ermöglicht im Userland, wieder abhängig von der Leistungsfähigkeit der eingesetzten Hardware, Task-Latenzzeiten im unteren Millisekunden- oder im dreistelligen Mikrosekundenbereich. Der Systementwurf, also die Einteilung in RT- und Nicht-RT-Prozessoren und die Verteilung der Threads auf die einzelnen Bereiche, ist komplex und zurzeit noch Handarbeit.

5.7 Multikernel-Architektur (RTAI/Xenomai)

Es ist sehr verbreitet, zur Realisierung hoher Anforderungen an das Zeitverhalten bei Einsatz eines Standardbetriebssystems dieses als einen Rechenprozess eines Realzeitkernels ablaufen zu lassen. Hierbei handelt