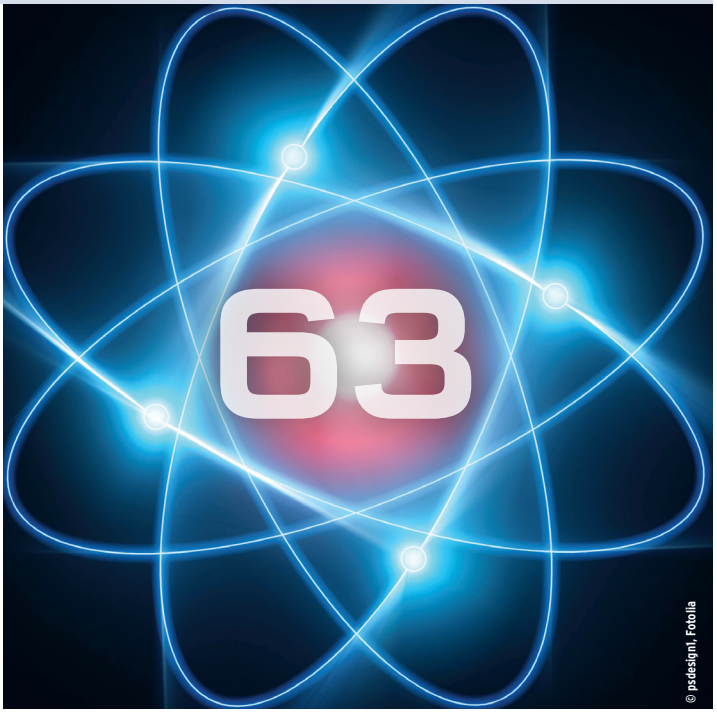
Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 63

Kern-Technik

Die Langzeitmessungen des Open Source Automation Development Lab zeigen erstmals: Mit Linux lassen sich Systeme für harte Echtzeitanforderungen realisieren. Die Kern-Technik stellt die Ergebnisse vor und erklärt, wie aus dem Standardbetriebssystem eines für Realzeitanwendungen wird. Jürgen Quade, Eva-Katharina Kunst



Vielerorts ist es noch einschlägige Lehrmeinung, dass Standard- und Realzeitbetriebssysteme zwei ganz verschiedene Paar Stiefel sind. Effizient und zugleich deterministisch, funktional und gleichzeitig sparsam im Umgang mit den Ressourcen, lauffähig auf einer PC-Plattform und auch auf einem Embedded-Prozessor – das sind die Zielkonflikte.

Deterministisch

Nun hat die Entwicklertruppe rund um Linus Torvalds das unmöglich Erscheinende möglich gemacht und Linux in den vergangenen Jahren ein deterministisches Zeitverhalten eingepflanzt. Mehr noch: Wenn ein Realzeitbetriebssystem vor zehn Jahren verlässlich Zeitschranken im zweistelligen Millisekundenbereich – abhängig von der eingesetzten Hardware – einhalten musste, schafft es Linux heute im Mikrosekundenbereich [1]. Allerdings benötigt der Kernel dazu noch das »PREEMPT_RT«-Patch [2]. Dieses fügt dem Standardkernel in der aktuellen Version gerade einmal 7000 Zeilen Code hinzu – Tendenz abnehmend, denn Torvalds übernimmt mit jeder neuen Release auch Teile des Patch in den Standardkernel. Dass der mit dem Patch versprochene zeitliche Determinismus nicht nur ein Marketing-Gag, sondern auch messbar ist, hat das Open Source Automation Development Lab (siehe Kasten „OSADL“) jüngst belegt. Es vergibt das Gütesiegel „Latest Stable Realtime“ an jene Kernel, die in permanenten Tests ein deterministisches Zeitverhalten bewiesen haben. Dazu betreibt das OSADL eine Rechnerfarm mit immerhin rund 50 Systemen, vorwiegend aus dem Embedded-Bereich.

Diese Systeme setzen unterschiedliche Prozessoren und Systemversionen ein, sind für Realzeitanwendungen konfiguriert und durchlaufen einen Stresstest. Dabei werden sie ständig überwacht. Insbesondere misst und protokolliert das OSADL ununterbrochen die Latenzzeiten des Kernels, also die Zeiten zwischen dem Auftreten eines Ereignisses und dem Start (nicht dem Ende) der zugehörigen Codesequenz. Nach gut einem Jahr und rund 73 Milliarden Messwerten pro System ziehen die Betreiber eine Bilanz [3]: Das Zeitverhalten des Linux-Kernels ist deterministisch und lässt sich mit einer Obergrenze abschätzen. Ausreißer kommen auf fehlerfreien Systemen nicht vor. Die Task-Latenzzeiten bewegen sich im Worst Case abhängig von der untersuchten Plattform im zwei- oder dreistelligen Mikrosekundenbereich.

Dieses Verhalten – so die weitere Feststellung – ist Kernel-immanent, also nicht abhängig von der eingesetzten Prozessorarchitektur. Darüber hinaus ist das Realzeitverhalten langzeitstabil, verschlechtert sich also nicht mit steigender Uptime des Rechners: Es gibt Systeme, die seit über 400 Tagen ununterbrochen aktiv sind. Diese Aussage ist alles andere als selbstverständlich, beispielsweise kann schon eine zunehmende Fragmentierung des Speichers das Zeitverhalten verschlechtern.

Keine Ausreißer

Die vom OSADL veröffentlichten Messungen visualisieren die Ergebnisse in Form eines Häufigkeitsdiagramms (Abbildung 1). Auf der x-Achse sind die möglichen Latenzzeiten in Mikrosekundenauflösung dargestellt. Die y-Achse gibt an, wie oft im Laufe der Messungen die zugehörige Latenzzeit aufgetreten ist. Die logarithmische Skalierung der y-Achse ermöglicht die Darstellung von großen Werten ebenso wie eine Häufigkeit von 1. Das macht potenzielle Ausreißer auch bei den hier vorkommenden großen Datenmengen direkt sichtbar.

Die dritte Dimension in der Darstellung (z-Achse, mit „Repetitions“ gekennzeich-

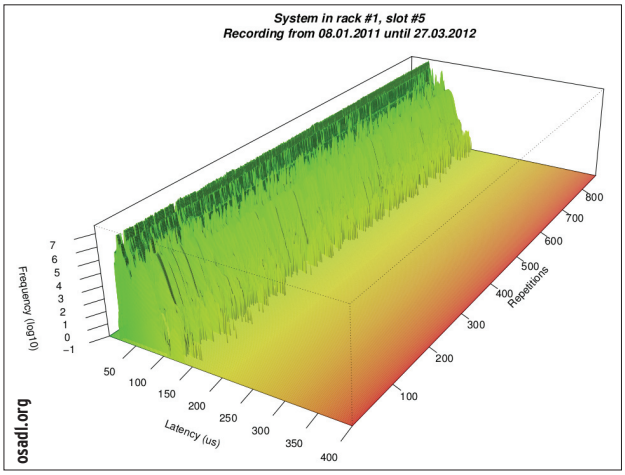


Abbildung 1: Häufigkeitsverteilung der Latenzzeit bei einem ARM-Prozessor (Texas Instruments OMAP 3517 mit 496 MHz). Die maximale Latenz von 87 Milliarden Messwerten liegt bei 158 Mikrosekunden.

net) entsteht dadurch, dass die Ergebnisse eines jeden Tages (jeweils zwei Plots) hintereinandergelegt werden. Eine Messung über 400 Tage führt zu 800 Repetitions beziehungsweise zu 80 Milliarden Messwerten. Die Gipfel stellen die unerwünschten Latenzzeiten dar. Systeme, die wie in der Abbildung keinen einzigen Gipfel in der Ebene aufweisen, sind potenziell für Realzeitsysteme geeignet.

Die anschaulichen Grafiken dürfen allerdings über eines nicht hinwegtäuschen: Relevant ist letztlich nur eine Zahl, nämlich die höchste aufgetretene Latenzzeit. Auf dem ARM-System in Abbildung 1 liegt diese bei 158 Mikrosekunden, das OSADL hat aber auch ein System mit einem Intel Core i7 (3,4-GHz-Takt) im Regal, das eine maximale Task-Latenz – also nicht nur Interrupt-Latenz – von unter 10 Mikrosekunden aufweist.

Um den „schlimmsten Fall“ zu provozieren, wechseln auf den unter Beobachtung stehenden Systemen Zeiten mit und ohne Last ab. Zur Erzeugung der Last kommen diverse Benchmarkprogramme zum Einsatz. Insbesondere erzeugen die Tester auch Lastfälle für das Netzwerk sowie für den Zugriff auf Peripherie (Festplatte) und auf den Hauptspeicher. Eine genaue Beschreibung findet sich unter [4].

Die Latenzzeit selbst misst das OSADL auf zwei Wegen. Zum einen setzen die Entwickler die Messsoftware Cyclictest [5] ein, zum anderen ein In-Kernel-Messverfahren, das Teil des »PREEMPT_RT«-Patch ist. Das In-Kernel-Messverfahren unterscheidet mögliche und effektive Latenzzeiten.

Mögliche Latenzzeiten ergeben sich durch kritische Abschnitte im Kernel. Die Dauer dieser Abschnitte lässt sich durch eine Differenzzeitmessung bestimmen. Dazu nimmt man zu Beginn des kritischen Abschnitts – der Kernel reserviert ein Mutex – und am Ende – der Kernel gibt das Mutex wieder frei – jeweils einen Zeitstempel (Abbildung 2).

Die Differenz der beiden Zeitstempel dient als Feldindex, die ausgewählte Feldadresse wird inkrementiert. Damit weiß man nicht nur, dass eine Latenz in der Größenordnung der Differenzzeit aufgetreten ist, sondern auch wie häufig. Im Fall einer Interruptsperre beziehungsweise -freigabe geschieht das Gleiche.

Latenzen messen

Zum Erfassen der effektiven Latenzzeit dient ebenfalls eine Differenzzeitmessung. Legt sich ein Job für eine definierte Zeit schlafen, speichert die Messsoftware den gewünschten Aufweckzeitpunkt als ersten Zeitstempel. Den zweiten nimmt sie, wenn der Job wieder aktiviert wird. Das Auswerten und Abspeichern funktioniert ebenfalls wie in Abbildung 2 beschrieben.

Um die In-Kernel-Messung auf einem mit »PREEMPT_RT« versehenen Kernel zu starten, schreibt man jeweils eine 1 in die Konfigurationsdateien »enable/wakeup« für die möglichen Latenzzeiten sowie »enable/missed_timer_offsets« für die effektiven Latenzzeiten, die im Verzeichnis »/sys/kernel/debug/tracing/la-

OSADL

Die Genossenschaft Open Source Automation Development Lab (OSADL) hat das Ziel, den Einsatz von Open-Source-Software, insbesondere von Linux, in der Automatisierungstechnik und in eingebetteten Systemen zu fördern und zu unterstützen.

Dazu bietet das OSADL seinen knapp 60 Mitgliedern rechtlichen Rat, technischen Support und Schulungen, es führt Konferenzen und Seminare durch und vermarktet Open Source aktiv. Außerdem bildet es eine Plattform für gemeinsame Entwicklungen und hostet eine Rechnerfarm mit Schwerpunkt bei eingebetteten Systemen, auf denen die hier vorgestellten Messungen gelaufen sind [3].

Die Mitglieder, von denen etwa drei Viertel aus dem industriellen und ein Viertel aus dem akademischen Umfeld stammen, können eigene Systeme in das Rack einstellen und vom OSADL monitoren lassen.

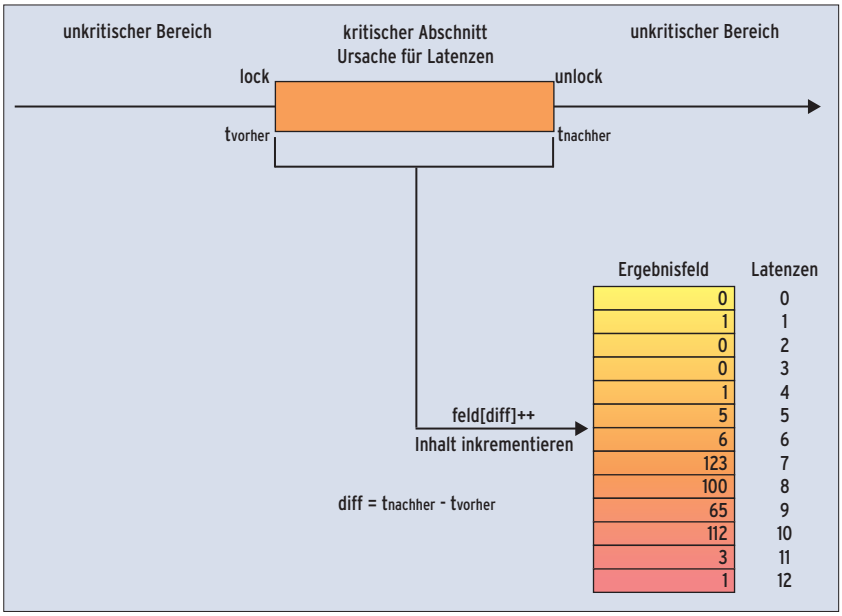


Abbildung 2: Das In-Kernel-Messverfahren registriert die Zeit vor und nach einem kritischen Abschnitt.

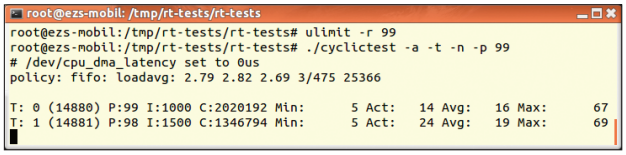


Abbildung 3: Die Ausgaben des Messprogramms Cyclictest, gestartet auf einem Dualcore-System (Intel T7500 mit 2,2 GHz).

tency_hist/« liegen. Das Generieren eines gepatchten Kernels hat die vorige Folge der Kern-Technik [6] beschrieben. Die Messergebnisse finden sich – ausgehend vom Hauptverzeichnis »latency_hist« – in den Verzeichnissen »wakeup«, »missed_timer_offsets« und »timerandwakeup«.

Realistische Daten

Die genutzte Messsoftware dagegen benötigt keine Unterstützung durch den Kernel (siehe Kasten „Latenzzeitmessungen mit Cyclictest"). Sie arbeitet ebenfalls mit der Zeitdifferenz zwischen einer gewünschten Aufweckzeit und der tatsächlichen Aktivierung. Diese Messung findet im Userland statt und ist damit aussagekräftiger und realistischer.

Ein ganzer Reigen von Kommandozeilenparametern ermöglicht unterschiedliche Messungen, beispielsweise getrennt für jeden Kern einer Multicore-Architektur (Abbildung 3).

Ein wesentlicher Baustein der Realzeitfähigkeit des Linux-Kernels besteht im High-Resolution-Timer-Subsystem, das eine auf Nanosekunden beruhende Zeitbasis eingeführt hat. Dadurch lassen sich zeitgesteuerte Dienste mit sehr hoher Genauigkeit zum gewünschten Zeitpunkt aktivieren. Des Weiteren haben die Entwickler beizeiten die Kernel-Preemption eingeführt, die Systemcalls und Kernelthreads unterbricht, falls höher priorisierte Codesequenzen abzuarbeiten sind. Im Gegenzug – und das ist „Work in Progress" – spüren die Entwickler kritische, ununterbrechbare Bereiche auf und sorgen dafür, dass diese möglichst kurz sind [7].

Prioritäten und Scheduling

Damit aus einem Standard- ein Realzeitbetriebssystem wird, ist aber noch mehr notwendig als nur kurze Latenzzeiten. Von einem Realzeitbetriebssystem erwartet man ein deterministisches, typischerweise auf Prioritäten basierendes Scheduling. Bei Linux arbeiten mehrere Scheduler Hand in Hand.

Übergeordnet gibt es einen Multicore-Scheduler, der die Last auf die Rechnerkerne verteilt. Auf jedem Rechnerkern läuft dann der Singlecore-Scheduler, der auf Prioritäten basiert. Linux bietet insgesamt 140 Prioritätsebenen, die – um den Spagat zwischen Standard- und Echtzeitbetriebssystem zu schaffen – in einen Bereich mit 40 und einen mit 100 Ebenen unterteilt sind. Der Bereich mit 100 Prioritätsebenen ist für Realzeitapplikationen gedacht.

Sollte es mehrere Jobs geben, die die gleiche Priorität haben, sich also auf der gleichen Ebene befinden, legt der Entwickler per Posix-Interface fest, ob der Kernel sie nach einem First-Come-First-Serve- oder einem Zeitscheiben-(Round Robin)-Scheduling-Verfahren abarbeitet. Im Round-Robin-Fall ist es möglich, die Länge einer Zeitscheibe zu konfigurieren.

Die unteren 40 Prioritätsebenen schließlich sind für den Teil „Standardbetriebssystem" bestimmt. Wenn es keine Jobs mit Realtime-Priorität gibt, wählt der Scheduler nach dem Completely Fair Scheduling (CFS) genannten Verfahren den nächsten Job auf der jeweiligen CPU. Wie der Name sagt, wird die Rechenzeit möglichst fair verteilt, um ein interaktives System zu gewährleisten.

Wie jedes Standardbetriebssystem bietet auch Linux zum Schutz kritischer Abschnitte eine Reihe von Lock-Mechanismen wie Semaphore, Mutexe und Spinlocks ([8], [9]). Um die Realzeit-Klientel zu bedienen, gibt es Realzeit-Mutexe, die insbesondere Prioritätsinversion unterstützen (siehe Kasten „Prioritätsinversion und Prioritätsvererbung"). Weitere Mechanismen, die für Standardbetriebssysteme obligatorisch sind, etwa Paging oder Swapping, lassen sich für die Echtzeitapplikationen per Memory-Locking gezielt deaktivieren.

Pflicht und Kür

Neben diesen Pflichtübungen beherrscht Linux aber auch die Kür und gibt Systemarchitekten erweiterte Möglichkeiten an die Hand. Mit Hilfe so genannter Control Groups beispielsweise lassen sich Jobs gezielt auf spezifischen Kernen einer Multicore-Maschine fixieren. Dies reserviert einen Teil der Hardware für die

Latenzzeitmessungen mit Cyclictest

Das OSADL bietet diverse Werkzeuge an, darunter auch das erwähnte »cyclictest« zum Messen des Realzeitverhaltens. Um Cyclictest auf Ihrem eigenen System einzusetzen, gehen Sie wie folgt vor: Sie benötigen das Versionsverwaltungssystem Git. Legen Sie ein neues Verzeichnis an und laden per Git den Quellcode herunter. Im dabei neu angelegten Verzeichnis rufen Sie direkt »make« auf. Tritt beim Kompilieren die Fehlermeldung auf, dass der Compiler die Datei »numa.h« nicht findet, fügen Sie in Zeile 22 des Makefile die Zeile »NUMA := 0« ein.

Zum Aufrufen von »cyclictest« müssen Sie Superuser sein. Eventuell sollten Sie auch die Userlimits für die Vergabe von Realzeitprioritäten mit dem Kommando »ulimit -r« anpassen. Die wesentlichen Befehle in der Übersicht:

```
# Quellcode herunterladen
mkdir rt-tests
cd rt-tests
KERNELGIT=git://git.kernel.org/pub/scm/
linux/kernel/git
git clone $KERNELGIT/clrkwllms/rt-tests.git
cd rt-tests
# Makefile editieren
NUMA := 0 # Zeile 22
# Generieren
make
# Programm starten
ulimit -r 99
./cyclictest -a -t -n -p 99
```

Sollten Sie in den Programmausgaben (Abbildung 3) auf unerwartete Latenzzeiten stoßen, können diese mehrere Ursachen haben [10]:

- Hardware-Bug, zum Beispiel beim Chipsatz
- System Management Interrupts (SMI), verursacht beispielsweise durch das Bios
- Dynamische Taktanpassung der CPU (Speedstep)
- CPU-Sleep-States
- Kernelkonfiguration, zum Beispiel Debug-Einstellungen
- Software-Bug

Standard- und den anderen Teil für Realzeit-Aufgaben. Je mehr die Multicore-Technologie in die Welt der eingebetteten Systeme vordringt, desto mehr gewinnt diese Technik an Bedeutung.

Oder Threaded Interrupts: Allein die Übergabe des Bootparameters »threadirqs« (auf einem Ubuntu-System beispielsweise in der Datei »/etc/default/grub« als »GRUB_CMDLINE_LINUX = "threadirqs"« eingetragen) splittet Interrupt-Service-Routinen in zwei Teile auf. Der Teil mit dem eigentlichen, also dem produktiven Code wird dann als Kernelthread abgearbeitet, der übrige Teil dient lediglich dazu, den Kernelthread zu aktivieren. Damit sind Interrupt-Service-Routinen plötzlich priorisierbar, und wichtige Interrupts, etwa vom Echtzeit-Ethernet-Controller können sogar bevorzugt vor weniger wichtigen Interrupts, beispielsweise von Tastatur oder Maus, abgearbeitet werden. Wenn Echtzeit konfiguriert ist, ist dies automatisch eingeschaltet, und der erwähnte Bootparameter muss nicht noch zusätzlich angegeben werden.

Dank dieser Maßnahmen gelingt Linux der Spagat, zugleich Standard- und Echtzeitbetriebssystem zu sein. Vielleicht noch dieses Jahr, spätestens aber im nächsten, so schätzt Carsten Emde, Geschäftsführer des OSADL, sei dazu nicht einmal mehr ein Patch notwendig. Wenn also eine (geeignete) 32- oder 64-Bit-CPU, insbesondere mit MMU-Unterstützung, zur Verfügung steht, fehlt Linux nur noch eine Zertifizierung für den Einsatz im sicherheitskritischen Umfeld. *(mhu)* ■

Infos

[1] Quade, Kunst, „Kern-Technik", Folge 31: Linux-Magazin 01/07, S. 110
[2] »PREEMPT_RT-Patch«: [http://www.kernel.org/pub/linux/kernel/projects/rt/]
[3] Carsten Emde, „OSADL celebrates one-year anniversary of the QA Farm": [https://www.osadl.org/Single-View.111+M56495907e88.0.html]
[4] OSADL, „Latency Plots": [https://www.osadl.org/?id=962]
[5] Cyclictest: [https://www.osadl.org/?id=209]
[6] Quade, Kunst: „Kern-Technik", Folge 62: Linux-Magazin 05/12, S. 84
[7] Quade, Kunst, „Kern-Technik", Folge 34: Linux-Magazin 07/07 S. 102
[8] Quade, Kunst, „Kern-Technik", Folge 44: Linux-Magazin 03/09, S. 88
[9] Quade, Kunst, „Kern-Technik", Folge 51: Linux-Magazin 05/10, S. 92
[10] Carsten Emde, „Mein RTOS ist nicht so echtzeitfähig, wie es sein soll – Was kann ich tun?": Embedded Software Engineering Kongress, 5. bis 9.12.2011, Sindelfingen

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. In der Zwischenzeit ist die dritte Auflage ihres Buches „Linux Treiber entwickeln" erschienen.

Prioritätsinversion und Prioritätsvererbung

Wenn ein Job mit mittlerer Priorität einen Job höherer Priorität aufhält, nennt man das „Prioritätsinversion" (Abbildung 4). Typischerweise sind daran mindestens drei Jobs beteiligt: Einer mit hoher (A), einer mit mittlerer (B) und eben einer mit niedriger Priorität (C). Außerdem teilen sich Job A und C ein Betriebsmittel. Sie haben also einen kritischen Abschnitt gemeinsam, den ein Lock (Semaphor, Mutex, Spinlock oder Futex) schützt.

Zu einer Prioritätsinversion kommt es, wenn C den kritischen Abschnitt betreten hat, in den A eintreten will. In diesem Fall blockiert A so lange, bis C die Unlock-Funktion aufruft. Wird in dieser Situation Job B lauffähig, verzögert er die Weiterverarbeitung von C, denn der hat ja niedrigere Priorität. Indirekt bremst er so aber auch A, obwohl die Verarbeitung von A aufgrund der Priorität dringlicher wäre.

Zur Entschärfung der Prioritätsinversion setzen Entwickler auf die so genannte Prioritätsvererbung. Sobald der hochpriorisierte Job A die Lock-Funktion beim Betreten des kritischen Abschnitts aufruft und das Semaphor oder das Mutex durch den niedrig priorisierten Rechenprozess C belegt ist, erbt C die hohe Priorität von A (Abbildung 5).

So ausgestattet lässt C sich nicht mehr durch den mittelpriorisierte Job B verdrängen. Gibt er das Lock frei, nimmt er wieder seine ursprüngliche Priorität an und der hochpriorisierte Job betritt – mit signifikant kürzerer Latenzzeit – den kritischen Abschnitt.

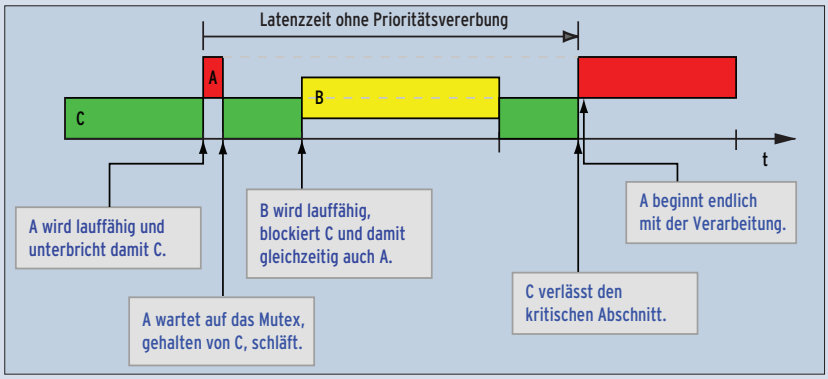


Abbildung 4: Eine Prioritätsinversion mit drei Jobs – A (rot, hohe Priorität), B (gelb, mittlere Priorität) und C (grün, niedrige Priorität) – entsteht, wenn der hochrangige Job A auf zwei nachrangige Jobs wartet.

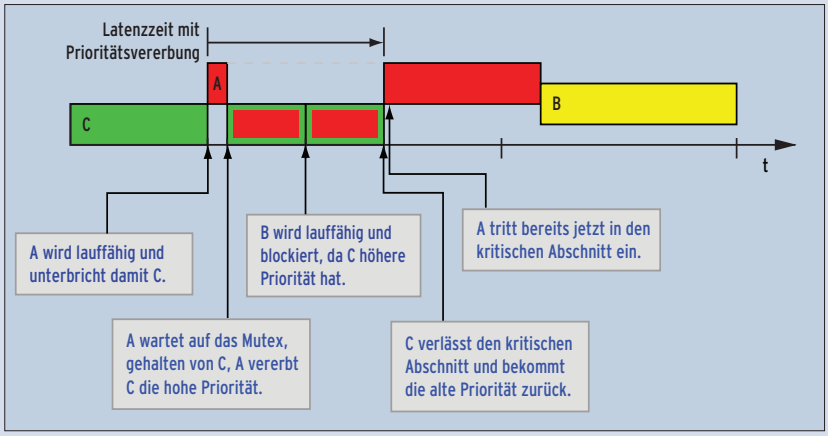


Abbildung 5: Prioritätsvererbung klärt die Situation: Der langsame Job C erbt die hohe Priorität von A, solange dieser auf Einlass wartet. Job B wartet. Die Latenz von A ist insgesamt deutlich geringer.