Jürgen Quade · Michael Mächtel

Moderne Realzeitsysteme kompakt

Eine Einführung mit Embedded Linux

dpunkt.verlag

Was sind E-Books von dpunkt?

Unsere E-Books sind Publikationen im PDF- oder EPUB-Format, die es Ihnen erlauben, Inhalte am Bildschirm zu lesen, gezielt nach Informationen darin zu suchen und Seiten daraus auszudrucken. Sie benötigen zum Ansehen den Acrobat Reader oder ein anderes adäquates Programm bzw. einen E-Book-Reader.

E-Books können Bücher (oder Teile daraus) sein, die es auch in gedruckter Form gibt (bzw. gab und die inzwischen vergriffen sind). (Einen entsprechenden Hinweis auf eine gedruckte Ausgabe finden Sie auf der entsprechenden E-Book-Seite.)

Es können aber auch Originalpublikationen sein, die es ausschließlich in E-Book-Form gibt. Diese werden mit der gleichen Sorgfalt und in der gleichen Qualität veröffentlicht, die Sie bereits von gedruckten dpunkt.büchern her kennen.

Was darf ich mit dem E-Book tun?

Die Datei ist nicht kopiergeschützt, kann also für den eigenen Bedarf beliebig kopiert werden. Es ist jedoch nicht gestattet, die Datei weiterzugeben oder für andere zugänglich in Netzwerke zu stellen. Sie erwerben also eine Ein-Personen-Nutzungslizenz.

Wenn Sie mehrere Exemplare des gleichen E-Books kaufen, erwerben Sie damit die Lizenz für die entsprechende Anzahl von Nutzern.

Um Missbrauch zu reduzieren, haben wir die PDF-Datei mit einem Wasserzeichen (Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer) versehen.

Bitte beachten Sie, dass die Inhalte der Datei in jedem Fall dem Copyright des Verlages unterliegen.

Wie kann ich E-Books von dpunkt kaufen und bezahlen?

Legen Sie die E-Books in den Warenkorb. (Aus technischen Gründen, können im Warenkorb nur gedruckte Bücher ODER E-Books enthalten sein.)

Downloads und E-Books können sie bei dpunkt per Paypal bezahlen. Wenn Sie noch kein Paypal-Konto haben, können Sie dieses in Minutenschnelle einrichten (den entsprechenden Link erhalten Sie während des Bezahlvorgangs) und so über Ihre Kreditkarte oder per Überweisung bezahlen.

Wie erhalte ich das E-Book von dpunkt?

Sobald der Bestell- und Bezahlvorgang abgeschlossen ist, erhalten Sie an die von Ihnen angegebene Adresse eine Bestätigung von Paypal, sowie von dpunkt eine E-Mail mit den Downloadlinks für die gekauften Dokumente sowie einem Link zu einer PDF-Rechnung für die Bestellung.

Die Links sind zwei Wochen lang gültig. Die Dokumente selbst sind mit Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer als Wasserzeichen versehen.

Wenn es Probleme gibt?

Bitte wenden Sie sich bei Problemen an den dpunkt.verlag:
Frau Karin Riedinger (riedinger (at) dpunkt.de bzw. fon 06221-148350).

Jürgen Quade studierte Elektrotechnik an der TU München. Danach arbeitete er dort als Assistent am Lehrstuhl für Prozessrechner (heute Lehrstuhl für Realzeit-Computersysteme), promovierte und wechselte später in die Industrie, wo er im Bereich Prozessautomatisierung bei der Softing AG tätig war. Heute ist Jürgen Quade Professor an der Hochschule Niederrhein, wo er u.a. das Labor für Echtzeitsysteme betreut. Seine Schwerpunkte sind Echtzeitsysteme, Embedded Linux, Rechner- und Netzwerksicherheit sowie Open Source. Als Autor ist er vielen Lesern über das dpunkt-Buch »Linux-Treiber entwickeln« und die regelmäßig erscheinenden Artikel der Serie »Kern-Technik« im Linux-Magazin bekannt.

Michael Mächtel ist Professor für Betriebssysteme an der Hochschule für Technik, Gestaltung und Wirtschaft in Konstanz, wo er u.a. das Labor für Systemsoftware und Realzeitsysteme betreut. Nach dem Studium der Elektrotechnik an der TU München arbeitete er im Bereich Fahrzeugtechnik. In seiner Promotion untersuchte er die Ursachen für Latenzzeiten in Realzeit-Betriebssystemen. Seit mehr als 20 Jahren arbeitet Michael Mächtel sowohl an der Universität als auch in der Wirtschaft im Embedded-System-Umfeld. Seine engeren Fachgebiete sind Betriebssysteme und Systemsoftware, Realzeitsysteme sowie Embedded Systems mit Schwerpunkt Embedded Linux.

Jürgen Quade · Michael Mächtel

Moderne Realzeitsysteme kompakt

Eine Einführung mit Embedded Linux

dpunkt.verlag

Jürgen Quade
quade@hsnr.de

Michael Mächtel
maechtel@htwg-konstanz.de

Lektorat: René Schönfeldt
Copy Editing: Sandra Gottmann, Münster
Satz: data2type GmbH, Heidelberg
Herstellung: Nadine Thiele
Autorenfotos: privat (Quade), Franzis von Stechow (Mächtel)
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN:
Buch 978-3-89864-830-1
PDF 978-3-86491-217-7
ePub 978-3-86491-218-4

1. Auflage 2012

Ringstraße 19B
69115 Heidelberg

5 4 3 2 1 0

Vorwort

Der amerikanische Begriff Realtime System wird im Deutschen mit Echtzeitsystem oder Realzeitsystem übersetzt. Alle drei Begriffe bezeichnen im technischen Umfeld Systeme, die neben funktionalen auch zeitlichen Anforderungen genügen. Während der populäre Begriff *Echtzeit* ganz allgemein eingesetzt wird, wird der Begriff *Realzeit* eindeutig auf den technischen Bereich fokussiert – weswegen ihm in den folgenden Ausführungen der Vorzug gegeben wird.

Realzeitaspekte gehören seit vielen Jahren zur klassischen Informatiker- und Ingenieursausbildung. Die fortschreitende Miniaturisierung und Leistungssteigerung der Hardware auf der einen Seite und die technologischen Entwicklungen in der Software – allen voran bei Linux – auf der anderen Seite ermöglichen dem Entwickler heute, seine Systeme mit erheblich mehr Funktionalität auszustatten als es früher mit vertretbarem Aufwand möglich war. Allerdings: Die eingesetzten Entwurfstechniken stehen den klassischen Technologien zum Teil diametral entgegen. Wurden früher parallel stattfindende Abläufe trickreich in eine Singlethread-Lösung gepackt, werden heute sequenzielle Abläufe parallelisiert: Erst damit wird eine Multicore-CPU effizient genutzt.

Der Wandel im Bereich der Realzeitsysteme geht noch tiefer. Die Parallelisierung beispielsweise bringt neue Probleme, wie den Schutz kritischer Abschnitte, mit sich. Deren klassische Methoden, nämlich Interrupts zu sperren oder ein Semaphor einzusetzen, bringen das Zeitverhalten durcheinander. Oder: Die Vernetzung benötigt Schutzmechanismen, um Hackerangriffe abzuwehren.

Diese ausgewählten Beispiele stehen stellvertretend für unsere Motivation, die Grundlagen moderner Realzeitsysteme kompakt zu beschreiben und zu diskutieren. Die Auswahl der Inhalte stammt sowohl aus Vorlesungen, die wir in den Studiengängen Informatik, Elektrotechnik und Mechatronik halten, als auch aus Industrieprojekten. Uns ist bewusst, dass viele Aspekte wie beispielsweise XML oder Webservices unberücksichtigt geblieben sind. Trotz dieser Auslassungen liegt das Themenspektrum im Umfang deutlich über der ursprünglichen Planung.

Wir hoffen, dass der Leser unsere Auswahl und die Beschreibungen nützlich und hilfreich findet!

Inhaltsverzeichnis

1 Einleitung

Systeme, die neben den ohnehin vorhandenen funktionalen Anforderungen zusätzlich noch zeitlichen Anforderungen – typischerweise im Sekundenbereich und darunter – genügen müssen, werden als Realzeitsysteme bezeichnet.

Als wesentliche Komponente eines solchen Realzeitsystems gilt die Steuerung, die über Ein- und Ausgabeinterfaces, über Sensoren und Aktoren mit dem Benutzer und der Umwelt kommuniziert. Klassische Realzeitsteuerungen bestehen aus einer Singlecore-CPU, die häufig am leistungstechnischen Limit betrieben wird. Als Systemsoftware wird – wenn überhaupt – ein schlankes, vielfach selbst entwickeltes Realzeitbetriebssystem eingesetzt.

Dabei wird der Entwickler oft mit dem Problem konfrontiert, viele unterschiedliche Aufgaben in einer einzelnen Task zu implementieren. Der Zugriff auf Hardware erfolgt direkt, ohne Interaktion mit der Systemsoftware. Höhere Funktionalitäten wie Internetdienste sind in diesem Umfeld nur mit erheblichem Aufwand zu realisieren.

Neue Entwicklungen auf dem Gebiet der Prozessoren hin zu mehr Leistungsstärke, zu Energieeffizienz und zu Multicore eröffnen neue Möglichkeiten, moderne Realzeitsysteme zu verwirklichen. In modernen Realzeitsteuerungen löst der sogenannte Tickless-Betrieb die periodisch ablaufende Systemsoftware ab. Intuitive Mensch-Maschine-Schnittstellen und Vernetzung, insbesondere die Internetanbindung, stehen dem Entwickler über standardisierte Interfaces zur Verfügung. Anstelle proprietärer Systemsoftware bietet sich ein echtzeitfähiges Standardbetriebssystem an, bei dem beispielsweise ein einzelner Prozessorkern für Realzeitaufgaben reserviert werden kann.

Die Applikation passt sich an. Wo klassisch ein einzelner Thread ereignisgesteuert viele unterschiedliche Aufgaben abarbeitet, werden bei einem modernen Systementwurf unabhängige Aufgaben identifiziert und in separate Tasks implementiert. Die dabei notwendige Inter-Prozess-Kommunikation bringt allerdings kritische Abschnitte mit sich, die mit geeigneten Methoden zu schützen sind.

Für die Kodierung greift der Entwickler auf die vielfältigen, vorgefertigten und ebenfalls standardisierten Funktionen des Betriebssystems

zurück, um beispielsweise Tasks zu erzeugen, zu priorisieren oder auf einen Rechnerkern zu fixieren; auf Funktionen, um den Scheduler zu konfigurieren, zwischen seinen Tasks Daten auszutauschen oder um kritische Abschnitte zu schützen, und auf Funktionen, die für das Zeitmanagement benötigt werden.

Die Erstellung eines modernen Echtzeitsystems erfordert über die Kenntnis der technischen Möglichkeiten hinaus ein theoretisches Basiswissen. Neben der formalen Beschreibung seines Entwurfs muss der Entwickler aus der Aufgabenstellung die Echtzeitparameter für den Realzeitnachweis extrahieren und die zugehörigen Realzeitbedingungen aufstellen und anhand des von ihm erstellten Entwurfs überprüfen können. Für diesen notwendigen, in der Praxis häufig ausgelassenen Schritt benötigt der Entwickler Kenntnisse über das eingesetzte Scheduling-Verfahren und über die detallierte Verarbeitung kritischer Abschnitte.

Ziel dieses Buches ist es damit,

- Grundkenntnisse moderner Realzeitsysteme kompakt und praxisorientiert zu vermitteln,
- die Parameter vorzustellen, die die zeitlichen Anforderungen auf der einen Seite und das Zeitverhalten der zur Lösung eingesetzten Rechenprozesse auf der anderen Seite beschreiben,
- die Möglichkeiten zum Entwurf und zur Realisierung moderner Realzeitsysteme auf Basis von Single- oder Multicore-Hardware aufzuzeigen,
- die vielfältigen Einsatzmöglichkeiten von (Embedded) Linux im Realzeitumfeld zu verdeutlichen,
- die praxisrelevanten Funktionen zur Realisierung der Applikation anhand von Codebeispielen vorzustellen,
- in die Grundlagen der Betriebssicherheit (Safety) und der Angriffssicherheit (Security) einzuführen,
- einfache und weitestgehend intuitiv einzusetzende formale Beschreibungsmethoden vorzustellen und
- das Rüstzeug für einen Realzeitnachweis zu vermitteln, der auch Blockierzeiten berücksichtigt.

Scope

Das Buch gibt einen kompakten Überblick über die wesentlichen Aspekte von Realzeitsystemen. Es ist damit weder ein themenumfassendes Handbuch, noch ein auf Einzelaspekte fokussiertes Werk.

Das Buch richtet sich sowohl an Neulinge auf dem Gebiet der Realzeitsysteme, wie beispielsweise Studenten der Informatik, Elektrotechnik oder Mechatronik, als auch an den erfahrenen Entwickler. Es soll dabei den Systemarchitekten ebenso ansprechen wie den Programmierer. Da

softwarespezifische Aspekte im Vordergrund stehen, richtet es sich nicht an Hardwareentwickler.

Grundkenntnisse auf dem Gebiet der Betriebssysteme sind zum Verständnis von Vorteil, die Codebeispiele im Praxisteil sind in der Programmiersprache C geschrieben. Sie sind grundsätzlich unabhängig von einer spezifischen Systemversion, wir haben sie auf einem Ubuntu 12.04 LTS mit Kernel 3.2 getestet.

Das Buch ist aufgrund der Verbreitung und der herausragenden Eigenschaften Linux-zentriert, allerdings dabei unabhängig von einer spezifischen Plattform oder einer bestimmten CPU. In Beispielen referenzieren wir meist die verbreiteten x86- oder ARM-Architekturen.

Realzeitsysteme werden im amerikanischen Sprachraum sehr häufig mit eingebetteten Systemen (Embedded Systems) gleichgesetzt. Streng genommen ist das nicht korrekt, denn die integrierte, mikroelektronische Steuerung – so die Definition eines eingebetteten Systems – muss nicht zwangsweise zeitliche Anforderungen erfüllen. Dennoch besteht zwischen Realzeitsystemen und eingebetteten Systemen eine enge Verwandtschaft und ist daher auch Thema dieses Buches. Sogenannte Deeply Embedded Systems – Systeme, die auf einfachen Prozessoren und oft selbst geschriebener Systemsoftware beruhen – sind allerdings außerhalb des Scopes.

Aufbau des Buches

In Kapitel 2 werden zunächst die Beschreibungsgrößen eines Realzeitsystems definiert und die Bedingungen abgeleitet, die zur Erfüllung der zeitlichen Anforderungen einzuhalten sind.

Das dritte Kapitel stellt die für das Realzeitverhalten wesentlichen Komponenten der Systemsoftware vor. Dazu gehören beispielsweise die innerhalb der Realzeitsysteme auswählbaren und parametrierbaren Scheduling-Strategien und die Zeitverwaltung.

Aspekte der nebenläufigen Realzeitprogrammierung behandelt das vierte Kapitel anhand von vielen Codebeispielen. Hier werden die wichtigsten Systemcalls und Funktionen vorgestellt, mit denen Tasks erzeugt, Zeiten gelesen oder kritische Abschnitte geschützt werden.

Verschiedene, mit Linux gut realisierbare Realzeitarchitekturen, basierend auf Single- und Multicore-Hardware, werden im fünften Kapitel vorgestellt.

Realzeitsysteme werden häufig im sicherheitskritischen Umfeld eingesetzt. Daraus resultieren erweiterte Anforderungen an die Korrektheit, an die Ausfallsicherheit, aber auch an den Schutz vor unbefugtem Zugriff, lokal, aber auch über das Internet. Kapitel 6 behandelt Aspekte der Betriebs- und der Angriffssicherheit.

Kapitel 7 gibt dem Entwickler ausgewählte Methoden an die Hand, mit denen die Taskgebilde eines Realzeitentwurfs dargestellt und die Abläufe innerhalb der Tasks modelliert werden können. Vielen Lesern dürften die vorgestellten Verfahren bekannt sein; das hat uns motiviert, den Abschnitt weiter hinten im Buch zu platzieren.

Das letzte Kapitel widmet sich schließlich dem Realzeitnachweis, mit dem formal das Einhalten der zeitlichen Anforderungen unter Berücksichtigung verschiedener Systemparameter (zum Beispiel Scheduling) nachgewiesen werden kann.

Embedded Linux

Im Rahmen des Buches referenzieren wir immer wieder auf Linux. Tatsächlich ist Linux in den letzten Jahren zu einem Standardbetriebssystem mit Realzeiteigenschaften umgebaut worden. Dabei ist Linux – natürlich abhängig von der eingesetzten Hardware – in der Lage, harte Zeitgrenzen im Mikrosekundenbereich einzuhalten. Allerdings benötigt der Kernel hierzu zurzeit noch den sogenannten PREEMPT-RT-Patch ([http://www.kernel.org/pub/linux/kernel/projects/rt/]). Da Linus Torvalds mit jeder neuen Kernel-Version Teile des Patches übernimmt, wird das Patchen jedoch schon sehr bald unnötig sein.

Langzeitmessungen am Kernel belegen, dass das deterministische Zeitverhalten unabhängig von der eingesetzten Prozessorarchitektur und auch langzeitstabil ist [QuKu2012-63]. Das wird durch die sogenannten High Resolution Timer (hrtimer) und den Tickless-Betrieb erreicht. Das Zeitmanagement des Kernels basiert intern nicht mehr auf Timerticks, sondern auf einer Zeitbasis in Nanosekundenauflösung. Zeitaufträge werden also nicht mehr mit dem nächsten Timertick, sondern zu exakt den Zeitpunkten angestoßen, zu denen der Auftrag terminiert ist.

Das für ein Realzeitbetriebssystem typische prioritätengesteuerte Scheduling unterstützt Linux bereits seit vielen Jahren, ebenso wie Mutexe, die eine sogenannte Prioritätsvererbung ermöglichen.

Linux zeichnet aber noch mehr aus. Mithilfe von Control-Groups lassen sich Tasks gezielt auf einzelne Kerne einer Multicore-Maschine fixieren und per Threaded Interrupts werden Interrupt-Service-Routinen als priorisierbare Threads abgearbeitet.

Das Applikationsinterface orientiert sich für Realzeitanwendungen am POSIX-Standard.

Neben den Realzeiteigenschaften weist Linux weitere Merkmale auf, die es für den Einsatz als Embedded System prädestinieren. Wesentliche Eigenschaften sind

- Skalierbarkeit,
- Modularität,
- Portierbarkeit,
- Quelloffenheit,
- eine große Entwicklergemeinde und
- gute und umfangreiche Dokumentation.

Um ein Embedded System zusammenzustellen und zu bauen, können Sie auf diverse Hilfsmittel wie beispielsweise Buildroot (siehe Abschnitt 3.3) oder OpenEmbedded ([http://www.openembedded.org]) zurückgreifen. Buildroot bietet sich für kleinere Projekte an, auf OpenEmbedded greifen Sie erst zurück, wenn Sie sehr komplexe Systeme aufbauen müssen. Beide Programme bauen nicht nur das eigentliche System zusammen, sondern sorgen selbstständig dafür, dass die notwendigen Entwicklungswerkzeuge wie beispielsweise Crosscompiler zur Verfügung stehen.

2 Realzeitbetrieb

Realzeitsysteme sind Systeme, die neben den funktionalen Anforderungen auch zeitlichen Anforderungen genügen. Sie sind also für die fristgerechte Reaktion auf Ereignisse zuständig. Diese Ereignisse – wie beispielsweise Überdrucksignale oder aber auch die Tastatureingaben eines Computerspiels – wirken quasi von außen auf das Realzeitsystem ein. Somit gibt die Umgebung zeitliche Anforderungen vor, denen das Realzeitsystem, das softwaretechnisch aus einer Reihe von Rechenprozessen, den Tasks (siehe Abschnitt 3.2.2), besteht, genügen muss.

Insofern sind im Folgenden die von außen eintreffenden Anforderungen, die darauf reagierenden Rechenprozesse und die für das Zeitverhalten mitverantwortliche Systemsoftware (Betriebssystem) zeitlich zu charakterisieren. Aus der zeitlichen Charakterisierung lassen sich zwei Echtzeitbedingungen ableiten. Der letzte Abschnitt dieses Kapitels behandelt anschließend funktionale Anforderungen an die Systemsoftware und weitere Aspekte, die sich aus der Abarbeitung beziehungsweise dem Zusammenspiel der Rechenprozesse ergeben. Zentral ist dabei der Umgang mit Ressourcen.

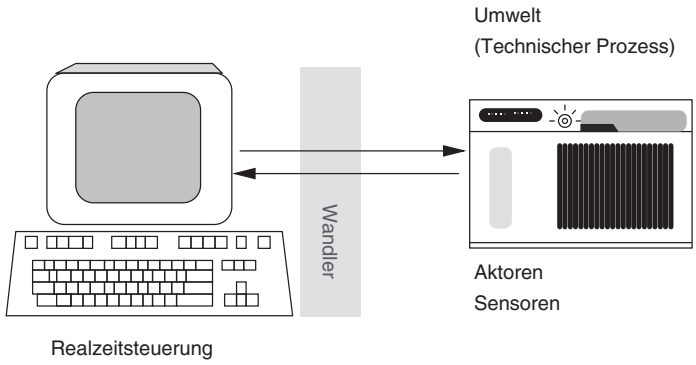


Abbildung 2-1
Realzeitsteuerungen verarbeiten Signale von und für die Umwelt.

Zur Erläuterung werden wir häufiger auf das Beispiel eines Fahrradcomputers zurückgreifen. Dieser bekommt mit jeder Umdrehung des Vorderrades über einen Magnetschalter ein Signal, aus dem unter Kenntnis der Vorderradgröße die Trittfrequenz, die Geschwindigkeit

und die zurückgelegte Strecke abgeleitet werden können. Das Ergebnis der Berechnungen wird per Touchscreen dem Radfahrer visualisiert. Zur Umwelt gehören demnach das Signal des Magnetschalters ebenso wie der Touchscreen, der sowohl für die Ausgaben zuständig ist als auch die Eingaben des Radfahrers entgegennimmt.

2.1 Zentrale Beschreibungsgrößen

Im Folgenden werden zunächst die zentralen Beschreibungsgrößen des technischen Prozesses, der Rechenprozesse und schließlich der Systemsoftware beschrieben.

2.1.1 Beschreibungsgrößen des technischen Prozesses

Der technische Prozess repräsentiert die Umwelt, die über Sensoren als Datenquellen Informationen zur Verfügung stellt beziehungsweise die über Aktoren (Datensenken) beeinflusst wird. Der Prozess wird formal über die von ihm gestellten Anforderungen an Rechenzeit und die zugehörigen Zeitparameter (Releasetime, Prozesszeit, Deadlines und Phase) beschrieben.

Rechenzeitanforderung

Löst der technische Prozess ein Ereignis aus, muss dieses in der zugehörigen Realzeitsteuerung (Computer) verarbeitet werden. Aus Sicht des Rechners stellt der technische Prozess eine *Rechenzeitanforderung*. Um die einzelnen Rechenzeitanforderungen voneinander unterscheiden zu können, werden sie mithilfe ihres Namens oder auch eines Buchstabens gekennzeichnet.

Beispiel 2-1 *Rechenzeitanforderungen an einen modernen Fahrradcomputer*

Aus Sicht des Fahrradcomputers stellt das Signal, das mit jeder Umdrehung des Vorderrades übermittelt wird, eine Rechenzeitanforderung dar, die vom Computer zu bearbeiten ist. Dieses Signal könnte mit einem *u* (für Umdrehung) gekennzeichnet werden. Eingaben über den Touchscreen sind ebenfalls Rechenzeitanforderungen des technischen Prozesses, die in diesem Fall vom Radfahrer stammen.

Releasetime

Der Zeitpunkt, zu dem eine Rechenzeitanforderung auftritt, wird als Releasetime $t_{Release}$ bezeichnet. Eine Rechenzeitanforderung u, die beispielsweise periodisch alle 200 ms auftritt, hat die folgenden Releasetimes: $t_{Release,u,1} = 0$ ms, $t_{Release,u,2} = 200$ ms, $t_{Release,u,3} = 400$ ms etc.

In Zeitdiagrammen wird der Zeitpunkt, an dem eine Rechenzeitanforderung auftritt, durch den zugehörigen Buchstaben gekennzeichnet (siehe Abbildung 2-2).

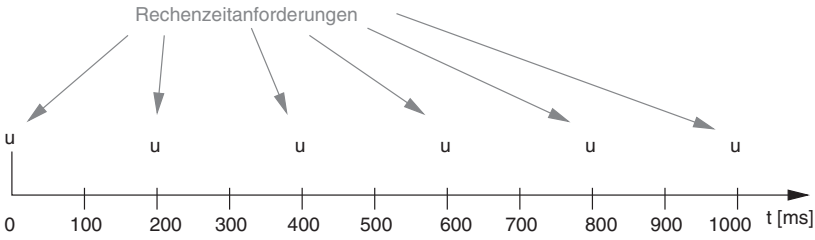


Abbildung 2-2
Rechenzeitanforderungen in Diagrammen

Prozesszeit

Der zeitliche Abstand zwischen zwei Rechenzeitanforderungen (Ereignissen) gleichen Typs ist die Prozesszeit, manchmal auch Periode t_P genannt. Einer Rechenzeitanforderung *i* ist eine Prozesszeit $t_{P,i}$ zugeordnet (siehe Abbildung 2-3).

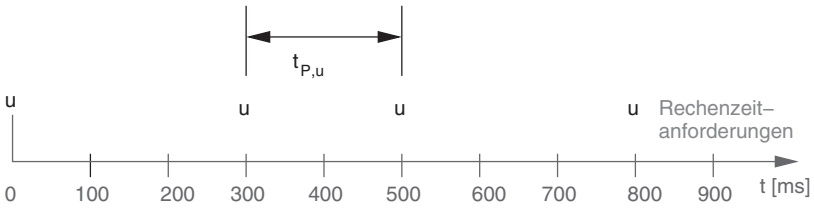


Abbildung 2-3
Prozesszeit

Prozesszeiten sind selten konstant. Sie schwanken zwischen einem minimalen Wert $t_{Pmin,i}$ (es vergeht wenig Zeit, bis die nächste Anforderung kommt) und einem maximalen $t_{Pmax,i}$. $t_{Pmax,i}$ repräsentiert das seltene Auftreten einer Rechenzeitanforderung und ist daher für Realzeitsysteme ohne Bedeutung.

Der Kehrwert der Prozesszeit $t_{P,i}$ ist die Rate r_i. Relevant ist vor allem die maximale Rate, die den Kehrwert der minimalen Prozesszeit darstellt.

$$r_{max,i} = \frac{1}{t_{Pmin,i}}$$

Der minimale zeitliche Abstand $t_{Pmin,u}$ für die Rechenzeitanforderung u des Fahrradcomputers, also des Signals, das mit jeder Radumdrehung ausgelöst wird, lässt sich auf Basis des Radumfanges und der maxima-

Beispiel 2-2
Bestimmung der minimalen Prozesszeit

len Fahrgeschwindigkeit berechnen. Hierzu wird die für eine Umdrehung minimal zurückgelegte Strecke (der minimale Radumfang) durch die maximale Geschwindigkeit dividiert. Das Vorderrad hat typischerweise einen Umfang zwischen 1200 mm bis 2300 mm. Als maximale Geschwindigkeit setzen wir 150 km/h an, also 41.667 m/s.

$$t_{Pmin,u} = \frac{d_{min}}{V_{max}} = \frac{1200mm}{150\frac{km}{h}} = \frac{1,2m}{41,667\frac{m}{s}} = 0,028799s \approx 28,8ms$$

$$r_{max,u} = \frac{1}{t_{Pmin,i}} = \frac{1}{0,028799s} = 34,7\frac{Umdrehungen}{s}$$

Übrigens ist die Bestimmung von $t_{Pmax,u}$ trivial. Der maximale zeitliche Abstand ergibt sich nämlich, wenn das Fahrrad steht, das Vorderrad sich also nicht dreht; $t_{Pmax,u}$ ist dann unendlich.

Zulässige Reaktionszeit (Deadline)

Der technische Prozess legt fest, ab welchem Zeitpunkt auf eine Rechenzeitanforderung i reagiert werden darf (minimale Deadline, $t_{Dmin,i}$) und bis wann auf eine Rechenzeitanforderung durch das Steuerungssystem reagiert worden sein muss (maximale Deadline, $t_{Dmax,i}$).

Mit $t_{Dmin,i}$ wird die untere Schranke bezeichnet, vor der keine Reaktion durch das Realzeitsystem erfolgen darf. In vielen technischen Prozessen darf direkt nach Auftreten der Rechenzeitanforderung reagiert werden. Die minimal zulässige Reaktionszeit ist in diesem Fall $t_{Dmin,i}$=0.

Beispielsweise darf die Reaktion bei einem klassischen Airbag direkt nach Detektion des Unfalls erfolgen, sodass $t_{Dmin,Airbag}$ = 0 ms beträgt. Bei einem modernen, zweistufigen Airbag jedoch darf je nach Unfallschwere die zweite Airbag-Stufe frühestens nach 5 ms gezündet werden. In diesem Fall ergibt sich eine minimal zulässige Reaktionszeit von $t_{Dmin,leicht}$ = 5 ms (siehe Abbildung 2-4).

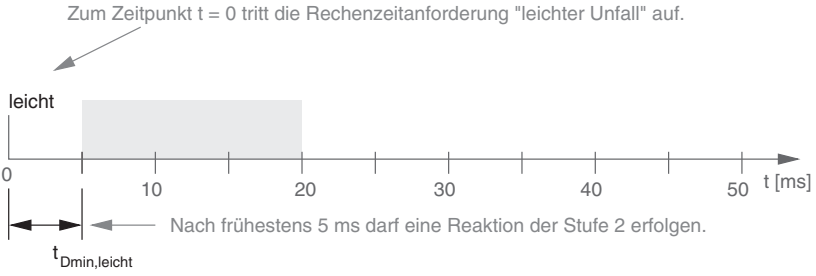


Abbildung 2-4
Minimal zulässige Reaktionszeit am Beispiel eines zweistufigen Airbags

Mit $t_{Dmax,i}$ wird die obere Schranke bezeichnet, bis zu der die Reaktion auf die Rechenzeitanforderung i erfolgt sein muss. Viele technische Prozesse erwarten eine *abgeschlossene* Reaktion auf eine Rechenzeitanforderung i spätestens bis die Rechenzeitanforderung i ein zweites Mal auftritt. In diesem Fall ist die maximal zulässige Reaktionszeit $t_{Dmax,i} \leq t_{Pmin,i}$.

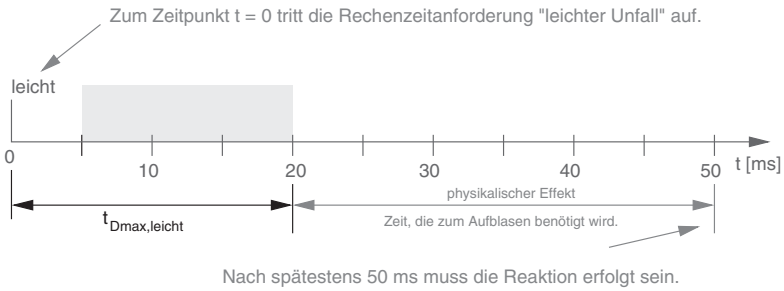


Abbildung 2-5
Maximal zulässige Reaktionszeit am Beispiel Airbag

Physikalische Effekte sind übrigens herausgerechnet, die zulässigen Reaktionszeiten geben die Zeitpunkte an, an denen das Echtzeitsystem reagiert, indem es das Ausgabesignal erzeugt. In einem Auto beispielsweise stehen nach einem Unfall etwa 50 ms zur Verfügung, bis der Airbag aufgeblasen sein muss, ansonsten ist er wirkungslos. Da jedoch das Aufblasen selbst 30 ms dauert, hat die zugehörige Rechenzeitanforderung (Unfall) eine maximale Deadline von $t_{Dmax,Unfall}$ = 50 ms - 30 ms = 20 ms.

Phase

Die Phase $t_{Ph,i}$ spiegelt den minimalen zeitlichen Abstand zwischen einer Rechenzeitanforderung *i* zum Bezugszeitpunkt t = 0 wider.

2.1.2 Beschreibungsgrößen der Rechenprozesse

Die Umwelt wird durch das eigentliche Echtzeitsystem – die Steuerung – beeinflusst. Dazu wird ein Rechner eingesetzt, auf dem für jede Rechenzeitanforderung eine Codesequenz abgearbeitet wird. Die Codesequenz ist meistens als eigenständige Task ausgeprägt, die durch die beiden Parameter Ausführungszeit und Reaktionszeit beschrieben wird.

Ausführungszeit

Die Ausführungszeit t_E einer Task ist die Zeit, die benötigt wird, um den Code der Task auf einem definierten Rechner abzuarbeiten. Somit stellt die Ausführungszeit die Summe der CPU-Zyklen dar, die für die Abarbeitung benötigt werden. Handelt es sich bei der Task um eine hundert-

prozentig CPU-intensive Task und ist diese Task die einzig aktive Task im System, so ergibt sich die Ausführungszeit als Differenz zwischen Endzeit und Startzeit.

Ungewollte und durch das System verursachte Verzögerungen (Latenzzeiten) sowie Zeiträume, in denen sich die Task gewollt schlafen legt, werden nicht zur Ausführungszeit gezählt.

Die Ausführungszeit schwankt zwischen einer Best Case Execution Time (BCET) t_{Emin} (siehe unten) und einer Worst Case Execution Time (WCET) t_{Emax} (siehe Seite 13). Ursachen dafür sind:

- Der implementierte Algorithmus: So werden beispielsweise abhängig von den Eingangsparametern unterschiedliche Codepfade durchlaufen.
- Caches (Daten- und Instruktionscaches, aber auch Translation Lookaside Buffer): Befindet sich beispielsweise der zur Task gehörende Code im Instructioncache, wird der Code mit sehr hoher Geschwindigkeit abgearbeitet. Muss die CPU den Code jedoch erst vom Hintergrundspeicher laden, wird erheblich mehr Zeit benötigt.

Andere Begriffe für die Ausführungszeit (Execution Time) sind Rechenzeit (Computation Time), Laufzeit oder auch Verarbeitungszeit.

Trägt man die Verarbeitungszeit t_E über die Zeit auf, erhält man die sogenannte Rechnerkernbelegung (Abbildung 2-6). Sie gibt Auskunft darüber, zu welchen Zeiten die Verarbeitungseinheit mit welcher Aufgabe betraut ist. Zur einfachen Erstellung der Rechnerkernbelegung ist es sinnvoll, in das Diagramm als Erstes die Zeitpunkte, an denen Rechenzeitanforderungen auftreten ($t_{Release}$, Abbildung 2-2), einzutragen.

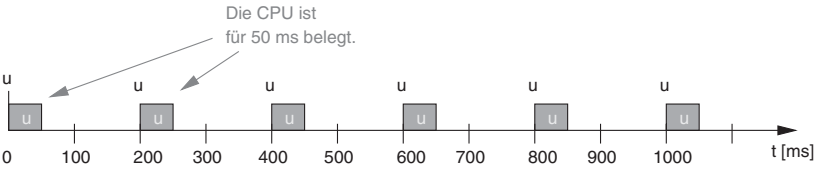


Abbildung 2-6 *Rechnerkernbelegung*

Abschätzung der Best Case Execution Time (BCET)

Die Best Case Execution Time einer Task muss bestimmt werden, falls eine Codesequenz nicht vor einer minimal zulässigen Reaktionszeit $t_{Dmin,i}$ abgearbeitet worden sein darf.

Wie bei der WCET gibt es zur Bestimmung den analytischen oder den experimentellen Ansatz. Beim analytischen Ansatz wird der vorhandene Code analysiert und der kürzeste Pfad gesucht. Aufgrund der Kenntnisse über die eingesetzte Hardware (CPU, Speicher) kann daraus die Abarbeitungszeit abgeleitet werden.

Beim experimentellen Ansatz wird der Code implementiert und der Entwickler lässt ihn auf der Zielhardware ablaufen. Die einzig vorgenommene Modifikation besteht darin, zu Beginn und zum Ende einen Zeitstempel zu nehmen, sodass per Differenzzeitmessung (siehe Abschnitt 4.4.3) $t_{Emin,i}$ bestimmt werden kann. Wichtig ist dabei, dass das System während der Messung lastfrei und nach Möglichkeit nur der auszumessende Code aktiv ist.

Abschätzung der Worst Case Execution Time (WCET)

Die maximale Verarbeitungszeit t_{Emax} – auch Worst Case Execution Time (WCET) genannt – einer Rechenzeitanforderung ist sehr schwer zu bestimmen. Die WCET hängt insbesondere vom Algorithmus, von der Implementierung des Algorithmus, von der verwendeten Hardware und von den sonstigen Aktivitäten des Systems (den anderen Rechenprozessen) ab. Daher kann die WCET allenfalls abgeschätzt werden.

Prinzipiell werden zwei Methoden zur Bestimmung der WCET unterschieden:

1. das Ausmessen der WCET und
2. die statische Analyse inklusive der daraus erfolgenden Ableitung der WCET.

In der Realität wird zur Abschätzung der WCET einer Task in vielen Fällen die oben genannte Messmethode verwendet. Die statische Analyse ist mehr im akademischen Umfeld verbreitet, rückt aber zunehmend aufgrund des Unsicherheitsfaktors der bisherigen Messmethode in den Fokus der Industrie. Für weitere Informationen siehe auch [LoMa11].

Ausmessen der WCET. Die WCET wird dadurch bestimmt, dass die zugehörige Codesequenz (im Regelfall auf der Zielplattform) abgearbeitet wird. Dabei wird die Zeit zwischen Eintreten der Rechenzeitanforderung und der zugehörigen Ausgabe bestimmt.

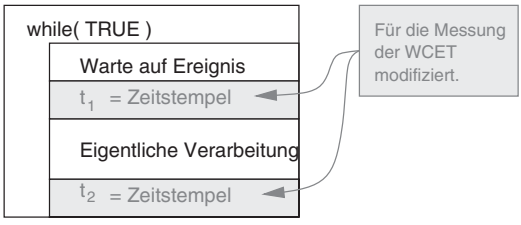


Abbildung 2-7
Messprinzip zur Bestimmung der WCET

Zur Messung dieser Zeit lassen sich prinzipiell zwei Verfahren unterscheiden:

1. Messung durch das auszumessende Codestück selbst mithilfe einer Differenzzeitmessung (siehe Abschnitt 4.4.3 und Abbildung 2-7).

Das Codestück wird so ergänzt, dass jedes Mal, wenn eine Rechenzeitanforderung auftritt und wenn die Reaktion erfolgt ist, ein Zeitstempel abgelegt wird. Die Differenz ergibt die aktuelle Ausführungszeit t_{Emax}.

Beim ersten Durchlauf wird diese als WCET abgelegt. Bei den nachfolgenden Durchläufen wird die neu berechnete Zeit mit der abgelegten Zeit verglichen. Ist die neu berechnete Zeit größer als die bisher gemessene WCET, überschreibt der neue Wert die bisherige WCET.

Eventuell kann anstelle der Modifikation des Codestücks auch ein Profiler eingesetzt werden.

2. Externe Messung von Ereignis und Reaktion (Ausgabe) zum Beispiel mithilfe eines Oszilloskops.

Damit eine WCET gemessen werden kann, muss die Codesequenz mit geeigneten Input-Werten versorgt werden. Für jeden Parameter wird der Wertebereich festgelegt. Die Messungen werden daraufhin mit verschiedenen Input-Daten durchlaufen: mit einem Wert, der für jeden Parameter einmal an den Rändern des Wertebereiches und einmal in der Mitte liegt. Darüber hinaus sollte aus Sicherheitsgründen auch eine Messung mit Werten außerhalb des identifizierten Wertebereiches vorgenommen werden.

Um die Ausführungszeiten im schlimmsten Fall zu bestimmen, ist darüber hinaus der Rechner unter Last zu setzen, um damit den Einfluss des Caches mit zu berücksichtigen. Allerdings kommt es durch die Lasterzeugung zu Kernel-Latenzzeiten, die im eigentlichen Sinne nicht Teil der Ausführungszeit sind. In der Praxis ist das jedoch unproblematisch.

Vorteile dieser Methode:

- Unabhängig von einer Programmiersprache.
- Vergleichsweise einfach realisierbar.

Nachteile der Messmethode:

- Die WCET einer Codesequenz kann nicht garantiert werden, da sie von zu vielen Einflussfaktoren (Vorgeschichte, Schleifendurchläufe, Caches, Verzweigungen usw.) abhängt.
- Das Messen ist zeitaufwendig und damit letztlich teuer. Das auszumessende Codestück muss theoretisch mit sämtlichen Inputdaten (in jeglicher Kombination) beschickt werden.
- Diese Methode kann korrekt nur mit dem produktiven Code auf der Zielplattform durchgeführt werden. Damit ist eine Abschätzung zu einem frühen Zeitpunkt nur bedingt möglich.
- Für die Durchführung der Messung muss eine Messumgebung, die die Inputdaten zur Verfügung stellt, geschaffen werden.

- Die auszumessende Codesequenz ist unter Umständen zum Ablegen von Zeitstempeln zu modifizieren.
- Durch die Messung unter Last kommt es zu ungenauen Ergebnissen, da Kernel-Latenzzeiten mitgemessen werden, die streng genommen nicht zur WCET gehören.

Statische Analyse. Hierbei wird der Code selbst analysiert. Dazu ist ein Analyseprogramm notwendig, welches meist mit dem C-Quellcode, manchmal aber auch mit dem Objektcode gefüttert wird. Außerdem ist eine Hardwarebeschreibung erforderlich. Die Analysewerkzeuge arbeiten so, dass aus dem Code zunächst ein Kontrollgraph erstellt wird. Die Laufzeit der einzelnen Codesequenzen des Kontrollgraphen können abgeschätzt werden. Im Anschluss muss die Anzahl der Schleifendurchläufe berücksichtigt werden.

Komplex wird die Analyse durch die notwendige Abschätzung von sogenannten Cache-Hits und Cache-Misses und des Prozessor-Pipelinings. Cache-Hits und Cache-Misses nennt man den Umstand, dass Code oder Daten bereits im Prozessor zwischengespeichert sind oder eben auch nicht. Sind sie nicht zwischengespeichert, müssen sie erst zeitaufwendig aus dem Hauptspeicher geladen werden. Das Pipelining ist eine Prozessoreigenschaft, die die Abarbeitung dadurch beschleunigt, dass Befehle vor der eigentlichen Ausführung bereits dekodiert werden. Insbesondere bei Verzweigungen ist diese Optimierung manchmal nicht nutzbar.

Unabhängig vom Verfahren gilt prinzipiell, den ausgemessenen Code nicht nachträglich zu verändern und außerdem noch mit einem Sicherheitsaufschlag zu versehen. Als Sicherheitsaufschlag arbeitet der Ingenieur manchmal mit der Zahl π.

$$t_{Emax} = t_{WCET} \cdot \pi$$

Die Bestimmung der WCET von Code, der auf einer Multicore-Maschine abgearbeitet wird, ist besonders schwierig. Hierbei kommen diverse Aspekte, wie beispielsweise die Zeit für die Taskmigration oder die unterschiedlichen Ausführungszeiten durch die Migration (Neubefüllen der Caches), zum Tragen. Erst zur Laufzeit werden aufgrund der Auslastung des Systems Tasks zwischen CPUs migriert. Wie oft dies beim Worst Case der Fall ist, hängt somit von der Auslastung des Gesamtsystems ab.

Reaktionszeit

Unter Reaktionszeit t_R wird die Zeit verstanden, die zwischen dem Auftreten einer Rechenzeitanforderung und dem *Ende* der Bearbeitung ver-

geht (siehe Abbildung 2-8). Durch die unterschiedlichen Wartezeiten kann sich die Reaktionszeit auch wie in Abbildung 2-9, dargestellt ergeben.

Abbildung 2-8
Reaktionszeit mit zusammenhängender Ausführungszeit

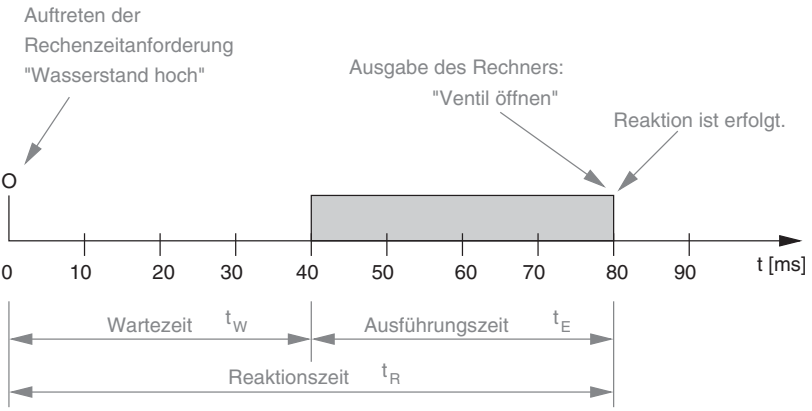
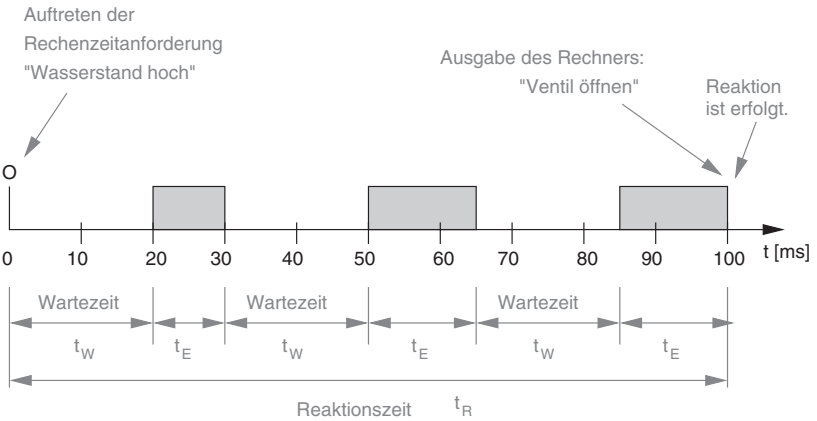


Abbildung 2-9
Reaktionszeit mit unterbrochener Verarbeitungszeit



Die Reaktionszeit wird sehr häufig mit der Interrupt- oder der Task-Latenzzeit verwechselt, die in der Reaktionszeit als Teil einer Wartezeit letztlich enthalten ist:

$$t_R = t_W + t_E$$

Die Wartezeit setzt sich zusammen aus Blockierzeiten und Latenzzeiten, die gemäß ihrer jeweiligen Ursache benannt sind (siehe Abschnitt 2.1.3).

Blockierzeiten t_B entstehen, wenn eine Task auf Daten oder ein Gerät zugreifen will, welches jedoch gerade exklusiv von einer anderen Task benutzt wird. Dann muss die Task auf die Freigabe der Daten oder des Gerätes warten. Während diese Blockierzeit ungewollt ist, kann t_B

auch gewollte Anteile enthalten, nämlich wenn sich die Task für eine definierte Zeit schlafen legt.

Je nach Auslastung des Systems ergibt sich eine minimale und eine maximale Reaktionszeit für die Rechenzeitanforderung i ($t_{Rmin,i}$ und $t_{Rmax,i}$).

2.1.3 Beschreibungsgrößen der Systemsoftware

Die Systemsoftware ist maßgeblich für das Zeitverhalten des Realzeitsystems mitverantwortlich. Sie wird unter anderem durch die Latenzzeiten (Verzögerungszeiten) gekennzeichnet. Diese können unterschiedliche Ursachen haben. Folgende Latenzzeiten werden unterschieden:

- Interrupt-Latenz
- Task-Latenz
- Kernel-Latenz
- Preemption Delay (Verdrängungszeit)

Die Interrupt-Latenzzeit t_{LIsr} ist die Zeit, die zwischen dem Auftreten eines Interrupts und dem Start der zugehörigen Interrupt-Service-Routine vergeht.

Unter der Task-Latenzzeit t_{LTask} versteht man die Zeit, die zwischen dem Auftreten eines Ereignisses – das kann ebenfalls ein Interrupt sein – und dem Start der zugehörigen Task vergeht.

Betriebssysteme werden bezüglich ihres Zeitverhaltens auf einer definierten Hardware primär über die Interrupt-Latenzzeit und die Task-Latenzzeit durch die Hersteller gekennzeichnet.

Kernel-Latenz $t_{LKernel}$ ist die Zeit, die eine Task vor oder während ihrer Ausführung warten muss, weil systembedingt und zumeist innerhalb des Kernels andere Codesequenzen bevorzugt abgearbeitet werden. Gründe für die Kernel-Latenz stellen somit Unterbrechungssperren und die vor parallelem Zugriff geschützten Bereiche innerhalb von Systemfunktionen respektive Systemcalls dar (siehe auch Abschnitt 4.3).

Verdrängungszeiten $t_{LPreempt}$, auch Preemptionzeiten genannt, entstehen, wenn ein wichtiges Ereignis eine gerade laufende Codesequenz unterbricht.

2.2 Realzeitbedingungen

Die fristgerechte Bearbeitung von Rechenzeitanforderungen ist dann garantiert, wenn die erste und die zweite Realzeitbedingung erfüllt sind: die Auslastungs- und die Rechtzeitigkeitsbedingung. Welche Auswirkun-

gen das Nichteinhalten der Rechtzeitigkeitsbedingung hat, wird in Abschnitt 2.2.3 erläutert.

2.2.1 Gleichzeitigkeit und Auslastung

Jeder im System vorhandene Rechnerkern stellt eine Rechenleistung zur Verfügung, bei der innerhalb eines Zeitfensters eine endliche Anzahl von Befehlen abgearbeitet werden kann. Wird eine Task aktiv, beansprucht sie einen Teil der Rechenleistung.

Abhängig von der Auftrittshäufigkeit einer Rechenzeitanforderung i und der damit anfallenden Ausführungszeit $t_{E,i}$ ist der Rechnerkern (Prozessor) ausgelastet. Die Auslastung ρ durch eine Rechenzeitanforderung ergibt sich damit als Quotient aus notwendiger Verarbeitungszeit und Prozesszeit:

$$\rho_i = \frac{t_{E,i}}{t_{P,i}}$$

Im Worst Case ist die Auslastung durch die Rechenzeitanforderung i durch Gleichung 2-2 gegeben.

Gleichung 2-2
Auslastung im Worst Case

$$\rho_{max,i} = \frac{t_{Emax,i}}{t_{Pmin,i}}$$

Ein Realzeitsystem muss in der Lage sein, die auftretenden Rechenzeitanforderungen in der Summe bearbeiten zu können (oft auch als Forderung nach *Gleichzeitigkeit*, Concurrency, bezeichnet). Mathematisch bedeutet dies, dass die Gesamtauslastung für jeden Rechnerkern kleiner als 100% (also 1) sein muss. Auf Mehrkernmaschinen kann jeder einzelne der *c* CPU-Kerne bis zu 100% belastet werden, sodass in diesem Fall die Gesamtauslastung kleiner als c mal 100% sein muss (Gleichung 2-3).

Die Gesamtauslastung (mit c = Anzahl der CPU-Kerne und n = Anzahl der Rechenzeitanforderungen) ergibt sich schließlich aus der Summe der Auslastungen der einzelnen Tasks (Gleichung 2-3).

Gleichung 2-3
Gesamtauslastung

$$\rho_{ges} = \sum_{j=1}^{n} \frac{t_{Emax,j}}{t_{Pmin,j}} \leq c$$

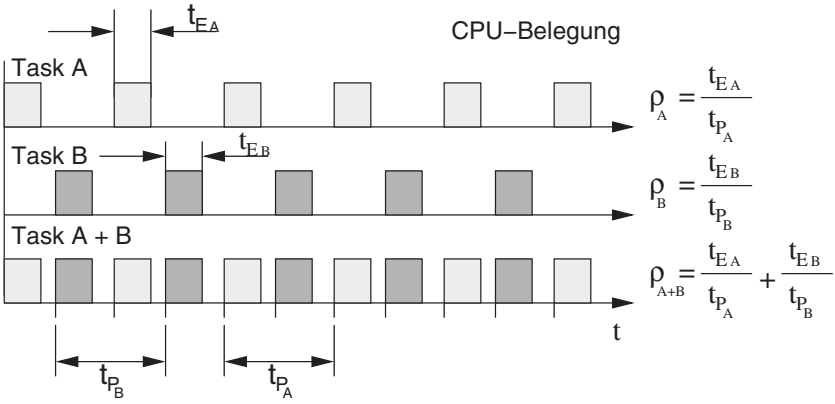


Abbildung 2-10
Auslastung

Abbildung 2-10 verdeutlicht die Auslastung für ein Singlecore-System grafisch. Ein Realzeitrechner bearbeitet zwei Rechenzeitanforderungen A und B mit jeweils einer eigenen Task: Task A und Task B. Sei die Verarbeitungszeit der Task A $t_{E,A}$ = 0.8 ms und die Prozesszeit $t_{P,A}$ = 2 ms ergibt sich eine Auslastung des Rechners durch die Task A von 40%.

Ist die Verarbeitungszeit der Task B ebenfalls $t_{E,B}$ = 0.8 ms und hat diese auch eine Prozesszeit von $t_{P,B}$ = 2 ms, ergibt sich die gleiche Auslastung wie die des Rechenprozesses A von ρ_B = 40 %.

Die Gesamtauslastung des Rechners der beide Rechenprozesse bearbeitet, beträgt $\rho_{ges} = \rho_A + \rho_B$ = 80 %.

Erste Realzeitbedingung

Die Auslastung ρ_{ges} eines Rechensystems muss kleiner oder gleich der Anzahl der Rechnerkerne sein.

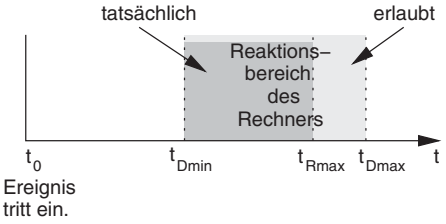
Die Auslastungsbedingung ist eine *notwendige*, aber keine *hinreichende* Bedingung. Für eine fristgerechte Bearbeitung ist es also notwendig (Grundvoraussetzung), dass die Auslastungsbedingung erfüllt ist; ohne sie geht es nicht. Andererseits reicht die Erfüllung der Auslastungsbedingung nicht aus, um die fristgerechte Bearbeitung garantieren zu können. Hierfür muss auch die zweite Realzeitbedingung erfüllt sein.

2.2.2 Rechtzeitigkeit

Neben der Korrektheit und der Vollständigkeit der Ausgangsdaten ist die Rechtzeitigkeit beziehungsweise Pünktlichkeit (Timeliness) der Daten von maßgeblicher Bedeutung. Stehen die Daten zu früh (vor $t_{Dmin,i}$) oder zu spät (nach $t_{Dmax,i}$) zur Verfügung, sind sie wertlos. Nur wenn sie dem technischen Prozess innerhalb des von diesem vorgegebenen Zeitfensters $t_{Dmin,i}$ und $t_{Dmax,i}$ zur Verfügung stehen, kann das gesamte Systemverhalten als korrekt angesehen werden.

Pünktlichkeit oder Rechtzeitigkeit darf nicht mit Schnelligkeit verwechselt werden. Allerdings lassen sich Anforderungen an Pünktlichkeit mit schnellen Systemen leichter erfüllen als mit langsamen. Prinzipiell muss ein Realzeitsystem eine Aufgabe in einem vorgegebenen Zeitfenster erfüllt haben – wie es das tut, ist dabei unwesentlich.

Abbildung 2-11
Reaktionsbereich



Gleichung 2-4
Zweite Realzeitbedingung (Rechtzeitigkeitsbedingung)

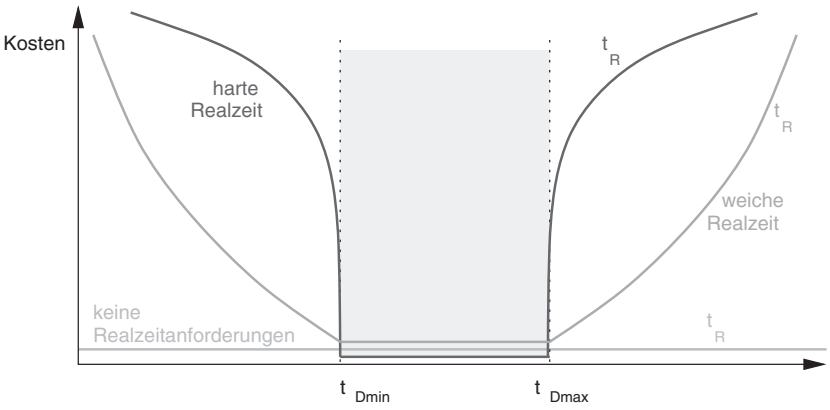
$$t_{Dmin,i} \leq t_{Rmin,i} \leq t_{Rmax,i} \leq t_{Dmax,i}$$

Zweite Realzeitbedingung

Um eine Rechenzeitanforderung i pünktlich (in Realzeit) zu erledigen, muss die Reaktionszeit größer oder gleich der minimal zulässigen Reaktionszeit, aber kleiner oder gleich der maximal zulässigen Reaktionszeit sein. Für alle Rechenzeitanforderungen *i* muss Gleichung 2-4 gelten.
Hierbei handelt es sich um eine notwendige und hinreichende Bedingung. Wenn diese Bedingung für alle Rechenzeitanforderungen i erfüllt ist, ist eine fristgerechte Bearbeitung gewährleistet.

2.2.3 Harte und weiche Realzeit

Abbildung 2-12
Harte und weiche Realzeit als Kostenfunktion



t_{Dmin} minimal zulässige Reaktionszeit

t_{Dmax} maximal zulässige Reaktionszeit

Die Forderung nach Pünktlichkeit ist nicht bei jedem System gleich stark. Während das nicht rechtzeitige Absprengen der Zusatztanks bei einer Rakete zum Verlust derselbigen führen kann, entstehen durch Verletzung der Rechtzeitigkeitsbedingung bei einem Multimedia-System allenfalls Komfortverluste. Vielfach findet man den Begriff *harte Realzeit*, wenn die Verletzung der Realzeitanforderungen katastrophale Folgen hat und daher nicht toleriert werden kann. Systeme, bei denen eine Deadline-Verletzung zunächst nur geringe Zusatzkosten aufwirft, bezeichnen wir als *weiche Realzeitsysteme*. Allerdings legt dieser Begriff nicht fest, wie stark oder wie häufig Deadline-Verletzungen toleriert werden dürfen.

Die Unterschiede bei der Forderung nach Pünktlichkeit werden oft anhand einer Kostenfunktion verdeutlicht. Bei den sogenannten *weichen Realzeitsystemen* bedeutet das Verletzen der Realzeitbedingung einen leichten Anstieg der Kosten. Bei den sogenannten *harten Realzeitsystemen* steigen jedoch die Kosten durch die Verletzung der Realzeitbedingung massiv an.

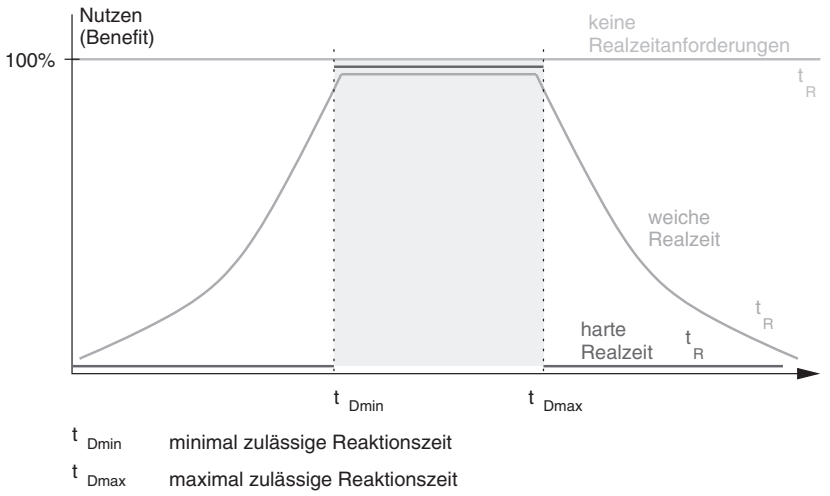


Abbildung 2-13
Nutzenfunktion

Eine andere Art der Darstellung ist die sogenannte Benefit Function (Nutzenfunktion). Hier wird der *Nutzen* der Reaktion des Realzeitsystems auf die Rechenzeitanforderung über der Zeit aufgetragen. Bei Systemen ohne Realzeitanforderungen ist der Nutzen unabhängig von dem Zeitpunkt der erfolgten Reaktion, also stets bei 100%. Bei einem harten Realzeitsystem ist nur dann ein Nutzen vorhanden, wenn die Reaktion innerhalb des durch t_{Dmin} und t_{Dmax} aufgestellten Zeitfensters erfolgt. Außerhalb des Zeitfensters ist der Nutzen null. Bei weichen Realzeitsystemen ist auch außerhalb des Zeitfensters ein Nutzen gegeben. Wie

stark dieser Nutzen ist, hängt jedoch vom jeweiligen System ab und kann nicht allgemein definiert werden.

2.3 Systemaspekte

Um Realzeitsysteme mit vernünftigem Aufwand realisieren zu können, muss das eingesetzte Betriebssystem einige Voraussetzungen erfüllen. Erstens muss es in der Lage sein, zusammengehörige Codesequenzen (Tasks) in mehrere Abschnitte zu zerteilen und Stück für Stück abzuarbeiten (Preemptibility). Zweitens muss die Systemsoftware Mechanismen zur Verfügung stellen, mit denen die Abarbeitungsreihenfolge unterschiedlicher Codesequenzen zeitlich festgelegt werden kann. Drittens schließlich muss die Systemsoftware Mechanismen zum Umgang mit Ressourcen anbieten. Durch Parallelverarbeitung kommt es nämlich schnell zu Ressourcenkonflikten und sogenannten kritischen Abschnitten.

2.3.1 Unterbrechbarkeit

Unter Unterbrechbarkeit (Preemptibility) versteht man die Eigenschaft beziehungsweise Anforderung an eine Codesequenz, die Abarbeitung in mehrere Abschnitte aufteilen zu können. Diese Abschnitte werden dann in der korrekten Reihenfolge, aber eben mit Unterbrechungen, in denen andere Verarbeitungen stattfinden, ausgeführt. Hierzu ein Beispiel:

Ein Messwerterfassungssystem, realisiert auf einer Singlecore-Hardware, soll im Abstand von 1 ms kontinuierlich Messwerte aufnehmen. Dazu benötigt die zugehörige Task eine Rechenzeit von 500 µs. Jeweils 100 Messwerte ergeben einen Datensatz, der vorverarbeitet und zur Archivierung weitergeleitet wird. Dazu ist eine Rechenzeit von 40 ms notwendig. Die Auslastung ergibt sich zu 0,5 + 0,4 = 0,9 = 90% Die Auslastungsbedingung (erste Realzeitbedingung) ist also erfüllt.

Bei linearer Abarbeitung der Aufgabe ohne Unterbrechbarkeit ergibt sich die in Abbildung 2-14 dargestellte Rechnerkernbelegung. Der zum Zeitpunkt 101 ms auftretende Messwert kann bereits nicht mehr erfasst werden. Die zweite Realzeitbedingung ist nicht erfüllt: Es handelt sich um kein Realzeitsystem.

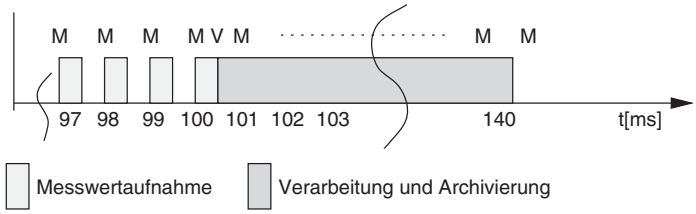


Abbildung 2-14
Messwerterfassung ohne Preemption

Splittet man jedoch die Aufgabe auf in einen Teil, der die eigentliche Messwerterfassung durchführt, und einen Teil, der für die Verarbeitung zuständig ist, und ermöglicht man des Weiteren die Abarbeitung einer Aufgabe in mehreren Schritten, die unterbrochen werden können, ist eine fristgerechte Abarbeitung möglich. Dies zeigt Abbildung 2-15. Um Realzeitaufgaben mit vertretbarem Aufwand lösen zu können, wird daher die Unterbrechbarkeit von Aufgaben, die im Folgenden als Tasks bezeichnet werden sollen, gefordert.

Man spricht von vollständiger Unterbrechbarkeit, wenn eine Task zu jedem beliebigen Zeitpunkt unterbrochen werden kann und keine Abhängigkeit – zum Beispiel über Ressourcen – zu anderen Tasks besteht. Die vollständige Unterbrechbarkeit kommt in der Realität nur selten vor.

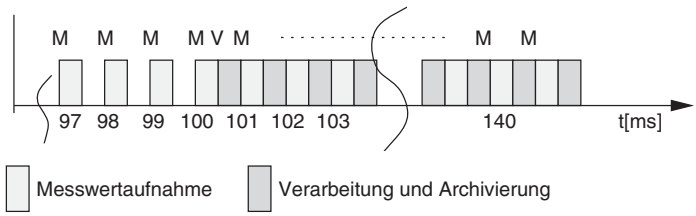


Abbildung 2-15
Messwerterfassung mit Preemption

Auch die Unterbrechbarkeit des Betriebssystemcodes entscheidet darüber, wie schnell die Reaktion auf ein Ereignis gestartet werden kann (Interrupt-Latenzzeit). Lange Interruptsperren im Betriebssystemcode verzögern das Starten der Reaktion. Die Unterbrechbarkeit des Betriebssystems selbst stellt somit ebenfalls einen wichtigen Aspekt dar.

2.3.2 Prioritäten

Zur Lösung von Realzeitaufgaben reicht die Forderung nach Unterbrechbarkeit nicht aus. Um die zeitlichen Anforderungen einhalten zu können, muss eine Systemsoftware zusätzlich die Möglichkeit bieten, die Ablaufreihenfolge der Tasks festzulegen. Hierzu gibt es verschiedene Verfahren, die in Klassifizierung der Scheduling-Verfahren (Seite 44) vorgestellt werden. Im Realzeitumfeld weit verbreitet und zugleich ein-

fach nachzuvollziehen ist die Festlegung der Ablaufreihenfolge über die Vergabe von Prioritäten. Es gibt keine einheitliche Definition darüber, wie eine höhere Priorität repräsentiert wird. In manchen Systemen stehen niedrige Zahlen für eine hohe Priorität, in anderen dagegen verhält es sich genau umgekehrt. Im Folgenden wird grundsätzlich eine hohe Priorität über eine niedrige Zahl (1) und eine niedrige Priorität über eine hohe Zahl repräsentiert (siehe hierzu auch Abschnitt 4.3.6).

2.3.3 Ressourcenmanagement

In Realzeitsystemen werden Rechenzeitanforderungen auf Tasks abgebildet, die auf der CPU ablaufen. Aus Gründen der Effizienz (Stromverbrauch) und aus Gründen der übersichtlicheren Entwicklung werden die Tasks gleichzeitig (real oder quasi-parallel) abgearbeitet. Dabei konkurrieren sie um die Ressourcen.

Ressourcen kann man in Hardware- und Software-Ressourcen unterteilen. Der Zugriff auf Hardware-Ressourcen wie z.B. CPU, Speicher und E/A-Geräte wird vom Betriebssystem geregelt. Software-Ressourcen sind typischerweise Synchronisationsprimitive wie Locks und Semaphore.

Unter einer Kooperation ist die Festlegung einer logischen Reihenfolge zu verstehen, in der die Tasks abzuarbeiten sind. Nachfolgende Tasks müssen teilweise auf die Beendigung der vorausgegangenen Task warten. So kann z.B. die Bearbeitung einer Task auf dem Ergebnis einer vorangegangenen Task aufbauen. Diese Art der Abhängigkeit ist in Abbildung 2-16 dargestellt.

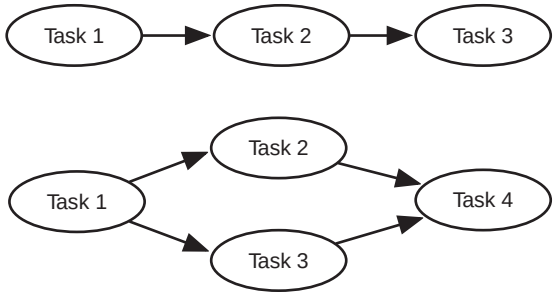


Abbildung 2-16
Task-Abhängigkeiten

Konkurrenz tritt auf, wenn Tasks versuchen, gleichzeitig auf gemeinsame Ressourcen zuzugreifen, wie zum Beispiel auf einen gemeinsamen Speicher (Shared Memory). Typischerweise dürfen diese Zugriffe jedoch nicht parallel erfolgen, da es sonst insbesondere bei Schreibzugriffen zu Dateninkonsistenzen kommen kann.

Beispiel 2-3
Inkonsistenz durch parallele Zugriffe

Die fehlerhafte Implementierung einer `echo()`-Funktion liest ein Zeichen von der Standardeingabe und gibt dieses auf die Standardausgabe aus:

```
char ch;

void echo()
{
   ch = getchar(); // Zeichen einlesen
   putchar(ch);    // Zeichen ausgeben
}
```

Angenommen: Zwei Tasks (genauer: Threads) verwenden diese Funktion und im Eingabepuffer befinden sich die Zeichen »AB«. Arbeitet die CPU die erste Zeile Code (`getchar()`) im Kontext der ersten Task ab, steht in der globalen Variablen `ch` der Wert »A«. Wird jetzt die erste Zeile im Kontext der zweiten Task abgearbeitet, überschreibt dies den Wert »A« mit dem »B«. Schließlich handelt es sich bei `ch` um eine globale Variable. Folglich wird später auf dem Bildschirm das falsche Ergebnis »BB« erscheinen.

Die Codesequenzen, die den Zugriff auf gemeinsame Ressourcen implementieren, werden kritische Abschnitte genannt. Damit es durch die kritischen Abschnitte nicht zu Inkonsistenzen kommt, muss der Programmierer Schutzmechanismen einsetzen. Solche Schutzmechanismen sind beispielsweise Semaphore, Mutexe, Spinlocks oder auch Monitore, wie sie in Abschnitt 4.3 vorgestellt werden. Im Regelfall sorgen diese Mechanismen für einen sogenannten gegenseitigen Ausschluss (Mutual Exclusion), also dafür, dass zu einem Zeitpunkt immer nur eine der konkurrierenden Codesequenzen abgearbeitet wird. Durch diese Serialisierung kommt es zu ungewollten Blockierzeiten.

Ressourcen werden mit dem Buchstaben R (Ressource) oder S (Semaphor) bezeichnet, also R_1, R_2 ... R_N oder S_1, S_2 ... S_N. Die Operationen, mit denen der Programmierer einen kritischen Abschnitt zu Beginn und zum Ende kennzeichnet, werden mit Lock $L(R_n)$ und Unlock $U(R_n)$ beziehungsweise mit $P(S_n)$ und $V(S_n)$ gekennzeichnet.

Die zeitliche Länge eines kritischen Abschnitts wird mit t_{CS} gekennzeichnet. In Abbildung 2-17 beispielsweise wird von T_1 die Ressource R_1 für zwei Zeiteinheiten und die Ressource R_2 für drei Zeiteinheiten reserviert. Damit ist $t_{CS,1}$ = 2 ms und $t_{CS,2}$ = 3 ms.

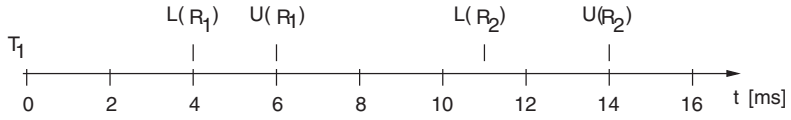
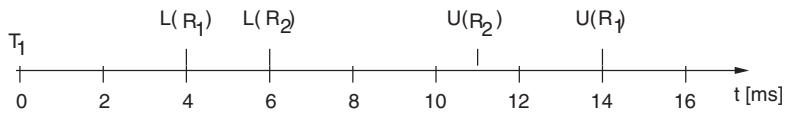


Abbildung 2-17
Kritischer Bereich im Code

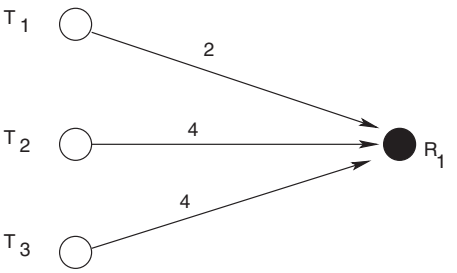
Noch komplizierter wird die Situation dadurch, dass Ressourcen oft verschränkt benutzt werden. In Abbildung 2-18 reserviert T_1 zunächst die Ressource R_1 und danach R_2. Für die verschränkte Benutzung der Ressourcen ist darauf zu achten, dass die Länge des kritischen Bereichs für R_1 die Länge für R_2 beinhaltet. Von daher ergibt sich aus Abbildung 2-18 die Länge für $t_{CS,R1}$ = 10 ms und für $t_{CS,R2}$ = 5 ms.

Abbildung 2-18
Verschränkte Benutzung von Ressourcen



Ressourcengraph

Abbildung 2-19
Ressourcengraph



In Abbildung 2-19 ist eine mögliche Darstellung der Abhängigkeiten von Tasks zu einer gemeinsamen Ressource R_1 dargestellt. Die Zahl am Pfeil gibt die Länge des kritischen Bereichs an. Die Länge der kritischen Bereiche ist somit:

- T_1: $t_{CS,R1} = 2$
- T_2: $t_{CS,R1} = 4$
- T_3: $t_{CS,R1} = 4$

Wir erweitern unser bisheriges Modell um folgende Parameter:

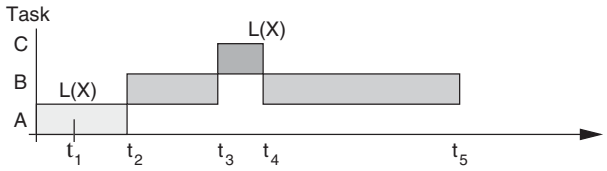
- $t_{CS,k,i}$: Länge des kritischen Bereichs der Benutzung einer Ressource k durch eine Task i
- π_i: Priorität der Task i
- $\Pi(R_k)$: Priorität der Ressource R_k
- Π_S: Priorität des Gesamtsystems

Die letzten beiden Parameter benötigen weitere Erläuterungen. Zum Berechnen der Blockierzeiten (siehe Abschnitt 8.3.1) werden auch die Ressourcen priorisiert, dargestellt mittels des Parameters $\Pi(R_k)$. Die Priorität einer Ressource wird durch die Task mit höchster Priorität be-

stimmt, die diese Ressource benötigt. Greifen also die Tasks T_1, T_2 und T_3 auf die Ressource R_1 zu und hat T_1 die Priorität eins und somit aus der Menge der drei Tasks die höchste Priorität, so bekommt die Ressource R_1 auch die Priorität eins zugeordnet. Die Priorität des Gesamtsystems Π_S wird von der im System benutzten Ressource mit höchster Priorität bestimmt.

2.3.3.1 Prioritätsinversion

Unter Prioritätsinversion versteht man die Situation, bei der eine niedrigpriore Task eine Ressource (über ein Semaphor) alloziert hat, die von einer hochprioren Task benötigt wird. Dadurch wird die Bearbeitung der hochprioren Task so lange verzögert, bis die niedrigpriore Task die Ressource freigibt. Das kann aber unzumutbar lang dauern, wenn im System eine Reihe Tasks lauffähig sind, die mittlere Priorität haben (siehe Abbildung 2-20).



t_1	Task A belegt die Ressource X.
t_2	Task B (mit höherer Priorität) wird rechenbereit.
t_3	Task C (mit der höchsten Priorität) wird rechenbereit.
t_4	Task C versucht, die Ressource X zu belegen, und wird schlafen gelegt.
t_5	Task B hat von den rechenbereiten Prozessen die höchste Priorität.

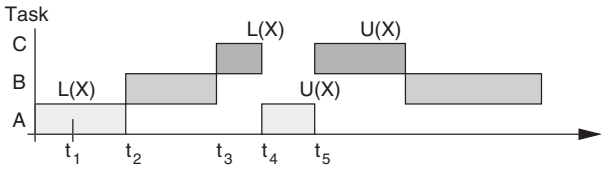
Abbildung 2-20
Prioritäten und Synchronisation

Zwei unterschiedliche Methoden stehen zur Verfügung, um die Situation zu entschärfen:

1. der Einsatz einer Unterbrechungssperre oder
2. der Einsatz von Ressourcen-Zuteilungsprotokollen

Einfachere Systeme schützen die wiederholte Ausführung von kritischem Code durch eine Unterbrechungssperre oder durch eine Task-Wechselsperre. Erst wenn die Task den kritischen Bereich verlassen hat, wird eine Unterbrechung oder ein Kontextwechsel wieder möglich. Dadurch ist beim Zugriff auf kritische Bereiche und Ressourcen auf einem Singlecore-System die Datenkonsistenz gewahrt.

Abbildung 2-21
Prioritätsinversion



t_1	Task A belegt die Ressource X (Lock).
t_2	Task B (mit höherer Priorität) wird rechenbereit.
t_3	Task C (mit der höchsten Priorität) wird rechenbereit.
t_4	Task C versucht, die Ressource X zu belegen, und wird schlafen gelegt.
t_4	Task A erbt die Priorität von Task C und wird rechenbereit.
t_5	Task A gibt X wieder frei und bekommt die ursprüngliche Priorität.

Um die Dauer der Prioritätsinversion zu begrenzen, können in Realzeitsystemen auch Protokolle für die Ressourcenzuteilung eingesetzt werden. Dadurch lassen sich Blockierzeiten begrenzen und die maximale Blockierzeit jeder Task berechnen. Völlig vermeiden lässt sich eine Prioritätsinversion in Systemen, die auf Prioritäten basieren, nicht.

Es gibt unterschiedliche Protokolle zur Prioritätsvererbung, zum Beispiel das

1. Priority Inheritance Protocol (PIP) oder das
2. Priority Ceiling Protocol (PCP).

Diese beiden Protokolle werden im Anschluss an die Diskussion über die Unterbrechungssperre vorgestellt. Die programmtechnische Behandlung der Prioritätsinversion wird in Abschnitt 4.3.2 thematisiert.

2.3.3.2 Unterbrechungssperre

Eine einfache Methode, um den gleichzeitigen Zugriff auf eine Ressource zu verhindern, stellt die Unterbrechungssperre (NPCS, Non Preemptive Critical Section) dar. Sie ist vor allem für Singlecore-Systeme geeignet. Auf Multicore-Systemen unterscheidet man die lokale von der globalen Unterbrechungssperre. Die lokale Unterbrechungssperre verhindert quasi-parallele Zugriffe auf der lokalen CPU, also der CPU, auf der sie angewendet wurde. Zugriffe, die zur gleichen Zeit über eine andere CPU stattfinden (real-parallele Zugriffe), werden dadurch aber nicht verhindert. Hierfür ist eine globale Unterbrechungssperre notwendig. In der Realisierung führt eine globale Unterbrechungssperre jedoch zu sehr hohen Latenzzeiten und wird in modernen Systemen (zum Beispiel Linux) daher nicht mehr eingesetzt.

Eine Unterbrechungssperre sorgt für eine atomare (ungeteilte) Abarbeitung einer Codesequenz. Sie existiert in unterschiedlichen Ausprägungen. In Linux gibt es für jede Ebene des Betriebssystems eine eigene

Unterbrechungssperre (Abschnitt 4.3.6). Die bekanntesten Unterbrechungssperren sind die Interruptsperre und die Task-Wechselsperre.

Solange sie wirksam ist, verzögert die Interruptsperre das Auftreten eines Interrupts. Die Task-Wechselsperre (Preemption Disable, Preemption Lock) verhindert – solange sie wirksam ist – dass eine andere Task auf der lokalen CPU aktiv wird. Sie ist damit eine Hauptursache für den Preemption Delay.

2.3.3.3 Priority Inheritance Protocol

Greift eine höherpriore Task auf eine Ressource R zu, die von einer niedrigprioren Task verwendet wird, so vererbt die höherpriore Task ihre Priorität auf die Task, die die Ressource derzeit besitzt, also auf die niedrigpriore Task. Gibt die niedrigpriore Task die Ressource frei, erhält sie die ursprüngliche Priorität zurück. In Abbildung 2-22 beispielsweise teilen sich die Tasks A und C eine Ressource. Task A wird zum Zeitpunkt 40 ms lauffähig und möchte auf die Ressource zugreifen. Zu diesem Zweck ruft die Task A die Funktion L(R) auf. Da die Ressource aber bereits durch Task C gehalten wird, wird Task A schlafen gelegt, obwohl Task A eine höhere Priorität hat. Beim Priority Inheritance Protocol wird allerdings die hohe Priorität auf C vererbt. Damit wird C als Nächstes abgearbeitet.

Die Zuteilung der Ressourcen geschieht bei PIP nach folgenden Regeln:

- Ist R frei, erhält die Task T bei einem L(R) die Ressource und behält diese so lange, bis sie R wieder über einen U(R) freigibt.
- Ist R nicht verfügbar, wird die Task T durch L(R) blockiert und erst wieder rechenbereit gesetzt, wenn R verfügbar ist.

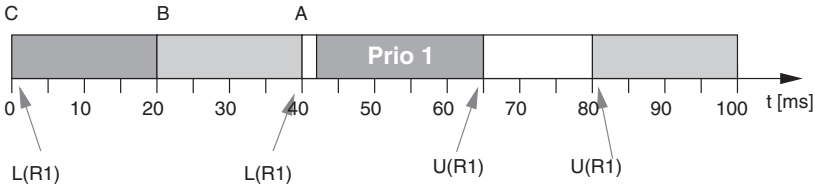


Abbildung 2-22
Priority Inheritance

0: C (niedrige Priorität) reserviert R1.
20: B (mittlere Priorität) unterbricht C.
40: A (hohe Priorität) versucht, R1 zu reservieren, A vererbt die Priorität an C.
42: C ist wieder lauffähig, für den Ressourcenzugriff mit hoher Priorität.
75: C gibt die Ressource frei, bekommt die ursprüngliche Priorität zurück.
80: A gibt die Ressource frei und beendet sich.

Die Prioritätsvererbung ist transitiv. Falls die Task T_O bereits die Priorität von T_R geerbt hat und eine weitere Task T_S, deren Priorität größer als die von T_R ist, lauffähig wird, erbt T_O die Priorität von T_S.

2.3.3.4 Priority Ceiling Protocol

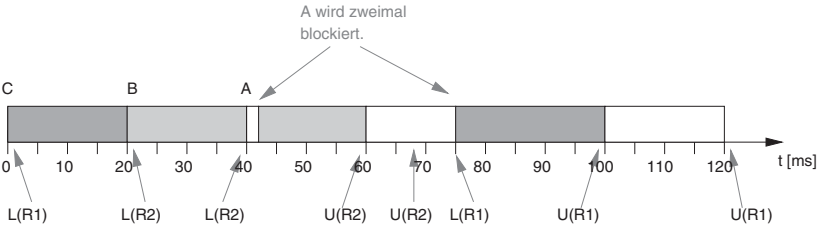


Abbildung 2-23
Eine hochpriore Task wartet u. U. mehrfach auf Ressourcen.

0: C (niedrige Priorität) reserviert R1.
20: B (mittlere Priorität) reserviert R2.
40: A (hohe Priorität) versucht, R2 zu reservieren – wird blockiert.
60: B gibt R2 frei, damit wird A aktiv.
75: A versucht, R1 zu reservieren – wird wieder blockiert, C wird dadurch aktiv.
100: C gibt R1 frei, A wird aktiv.

Verwendet eine (hochpriore) Task mehrere Ressourcen, verhindert das Priority Ceiling Protocol (PCP), dass diese Task mehrfach auf Ressourcen warten muss (Abbildung 2-23) – sie wartet damit maximal auf eine Ressource; nachfolgende Anfragen werden ohne Blockierzeit bedient. Damit verhindert PCP Deadlocks und die Gefahren, die verschachtelte kritische Abschnitte mit sich bringen, sind reduziert. Nachteilig ist, dass der Programmierer beim System die Benutzung seiner Ressourcen vorher anmelden muss. Mit dieser Information bestimmt das Protokoll bei jeder Ressourcenanforderung oder -freigabe die sogenannte Priority Ceiling des Systems (Π_s). Π_s kann z.B. als eine Prioritätsobergrenze des Systems übersetzt werden. Die Bestimmung von Π_s geschieht folgendermaßen:

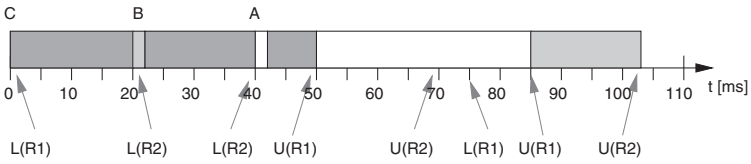
- Intern im System werden den Ressourcen Prioritäten zugeordnet. Die Task mit der höchsten Priorität, die eine Ressource R benutzt, bestimmt die Priorität für diese Ressource. Wenn zum Beispiel die Tasks T_1, T_2 und T_3 eine Ressource R benutzen und T_1 die Task mit höchster Priorität ist, so bekommt die Ressource R die Priorität 1 zugeordnet.
- Die aktuelle Priority Ceiling Π_s des Systems entspricht der Priorität der Ressource, die aus der Menge der benutzten Ressourcen die höchste ist. Werden zum Beispiel die Ressourcen R_2, R_3 und R_4 im System zum Zeitpunkt t von verschiedenen Tasks benutzt und die

Ressource R_2 hat von dieser Menge die höchste zugeordnete Priorität, so hat die Π_s die Priorität der Ressource R_2.

- Wird keine Ressource benutzt, so wird Π_s auf die niedrigste Priorität des Systems gesetzt, was typischerweise mit 0 gekennzeichnet wird.

Über die Priority Ceiling des Systems (Π_s) wird die Ressourcenzuteilung beim PCP geändert. Wenn eine Task T_i zurzeit t eine Ressource anfordert:

- und R nicht verfügbar ist, wird die Anforderung abgelehnt und T_i blockiert. Das war bei PIP genauso. Neu ist aber nun, dass
- wenn R frei ist:
  - T_i die Ressource nur dann erhält, wenn die Priorität der Task i höher als $\Pi_s(t)$ ist.
  - Falls die Priorität der Task i gleich oder niedriger als $\Pi_s(t)$ ist, erhält T_i die Ressource R nur, wenn T die Task ist, durch deren Ressourcenzugriffe die derzeitige $\Pi_s(t)$ ausgelöst wurde. Ansonsten wird T weiterhin blockiert.



0: C (mittlere Priorität) reserviert R1, die System Priority Ceiling ist jetzt 1.
20: B (mittlere Priorität) versucht, R2 zu reservieren, wird blockiert, C erbt die Priorität von B. (Die Priorität von B ist nicht größer als die System Priority Ceiling.)
40: A (hohe Priorität) versucht, R2 zu reservieren; wird blockiert, C erbt die Priorität von A. (Die Priorität von A ist nicht größer als die System Priority Ceiling.)
50: C gibt R1 frei, A wird aktiv.
85: B wird aktiv, da A beendet ist.

Abbildung 2-24
Priority Ceiling

Abbildung 2-24 veranschaulicht die Abläufe. Unter Umständen wird dem L(R) einer Task also nicht stattgegeben, obwohl die Ressource frei ist (t = 20 und t = 40). Bei jedem Ressourcenzugriff oder bei jeder Ressourcenfreigabe wird die Priority Ceiling des Systems entsprechend angepasst.

Die Prioritäten-Vererbungsregel gilt weiterhin: Wenn die Task T_i, die die Ressource angefordert hat, blockiert wird, so erbt die Task T_j, die zur Blockierung von T_i geführt hat, vorübergehend die aktuelle Priorität $\pi(t)$ von T. Die Task T_i wird nun mit dieser Priorität ausgeführt, bis sie jede Ressource abgibt, deren Priority Ceiling gleich oder höher als die derzeitige Priorität $\pi(t)$ von T_j ist. Die Task T_j erhält dann wieder die Priorität, die sie vor dem Zugriff auf die Ressource hatte.

Aufbauend auf PCP gibt es weitere Protokolle wie z.B. das Stack Based Priority Ceiling Protocol (SPCP). Das Prinzip von SPCP ist einfach: Eine Task wird nur dann gestartet, wenn alle Ressourcen für die Task frei vorzufinden sind. Nachdem eine Task gestartet ist, wird ihre Ausführung so lange blockiert, bis die Priorität der Task höher ist als die Priority Ceiling des Systems.

Beim PCP- und SPCP-Protokoll wird aufgrund der Priority Ceiling entschieden, ob eine freie Ressource zugeteilt werden darf oder nicht. Über diese Priority Ceiling wird ein geordneter Zugriff auf die Ressourcen implementiert. Da nun die Tasks auf die Ressourcen nur nach einem Ordnungssystem zugreifen können, wird ein Deadlock im System verhindert.

Protokolle, die auf PCP aufbauen, erreichen in der Regel deutlich kürzere Blockierzeiten für die Tasks als z.B. das PIP-Protokoll – und das, obwohl sie freie Ressourcen unter Umständen einer Task nicht sofort zuteilen.

3 Systemsoftware

Der Aufbau von Realzeitsystemen vereinfacht sich durch den Einsatz von Systemsoftware. Diese führt die Basisinitialisierung der Hardware durch, sie übernimmt die Aufgabe eines Bootloaders und lädt Software in den Speicher, ermöglicht das Debugging des Systems und erleichtert durch vorgefertigte Funktionen den Zugriff auf die Hardware. Hersteller von Systemsoftware liefern diese häufig unter dem Namen Board-Support-Package aus und integrieren dabei noch die notwendigen Entwicklungswerkzeuge (Compiler, Linker, Bibliotheken).

Welche Systemsoftware eingesetzt werden kann, ist abhängig von der verwendeten Hardware und der Komplexität der geforderten Funktionalität. Dabei geht der Trend ganz eindeutig zu vernetzten Systemen und in Richtung komplexerer Realzeitbetriebssysteme, allen voran Linux als in weiten Teilen realzeitfähiges Standardbetriebssystem.

Der Entwickler eines Realzeitsystems muss nicht nur die technischen Möglichkeiten der Systemsoftware kennen, sondern zusätzlich mit dem Aufbau vertraut sein. Nur so kann er die Systemsoftware um allgemeine Funktionalitäten und notwendige Treiber erweitern. Außerdem entscheidet er durch Auswahl und Parametrierung des Systemkerns über dessen Zeitverhalten.

3.1 Firmware

Jeder Rechner benötigt eine sogenannte Firmware, die für die Basisinitialisierung (Inbetriebnahme) der Hardware notwendig ist. Diese Software befindet sich typischerweise im nicht flüchtigen Speicher (ROM, Flash) und wird aktiviert, sobald der Rechner angeschaltet wird. Die Firmware hat die folgenden Aufgaben:

1. Grundinitialisierung der Hard- und Software, insbesondere von
   - CPU
   - MMU (Speicherverwaltung)
   - Timer/Uhr
2. Diagnose, Funktionstest wichtiger HW-Baugruppen wie beispielsweise des Speichers (Power On Self Test).

3. Betriebsinitialisierung der Hard- und Software, insbesondere von
   - Interrupt
   - MMU
   - Timer/Uhr
   - Watchdog
   - Serielle Schnittstelle
   - Ethernet
   - Display
4. Laden und Aktivieren von Code
5. Runtime-Services (Basisfunktionen), die einer durch die Firmware aktivierten Software (Applikation) zur Verfügung stehen, beispielsweise
   - Funktionen zum Zugriff auf Flash-Speicher
   - Zeit auslesen und setzen
   - Watchdog-Konfiguration

In Realzeitsystemen beziehungsweise eingebetteten Systemen, die eine übersichtliche Funktionalität aufweisen, wird die Firmware häufig direkt mit der eigentlichen Applikation kombiniert und als ein Binary im Speicher abgelegt, so beispielsweise bei einem Waschmaschinen-Steuergerät (siehe Abbildung 3-1).

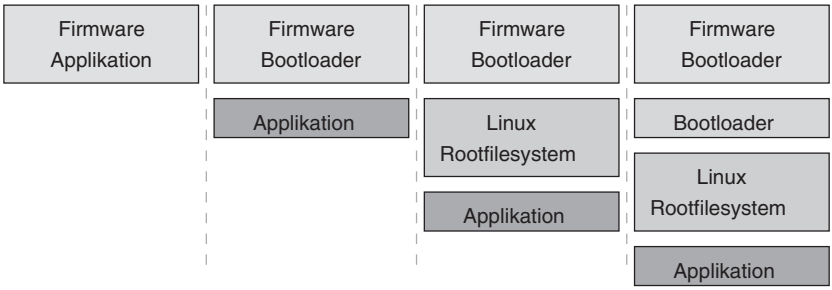


Abbildung 3-1
Systemsoftware - Architekturvarianten

Die Installation beziehungsweise der Austausch einer Firmware ist allerdings nicht trivial. Heute liegt diese häufig in einem fest eingebauten Flash-Speicher; Zugriff gibt es nur über spezielle Hardware-Interfaces, beispielsweise JTAG. Um in der Entwicklung flexibler zu sein, werden Firmware und Applikation entkoppelt. Die Firmware wird um einen Lademechanismus für eine Applikation erweitert. Sie enthält zusätzlich Routinen, mit denen der Flash-Speicher beschrieben werden kann, die Programmkontrolle an die geladene Software übergeben wird und die Debugging ermöglicht. In dieser Fassung spricht man häufig auch von der Monitor-Software und beim Lademechanismus vom Bootloader.

Häufig stellt eine Monitor-Software neben den Boot-Services noch Runtime-Services zur Verfügung. Applikationen können auf Funktionen zurückgreifen, die beispielsweise die aktuelle Uhrzeit zurückliefern oder den Zugriff auf den Flash-Speicher ermöglichen.

Wird mehr Funktionalität gefordert, lädt der Bootloader nicht direkt die Applikation, sondern zunächst ein Betriebssystem, welches die parallele Verarbeitung mehrerer Tasks und den bequemen Zugriff auf die Hardware ermöglicht. Das Betriebssystem startet schließlich die Applikation.

Im PC-Umfeld wird die Firmware BIOS (Basic Input Output System) genannt. Da der BIOS-Bootloader aus historischen Gründen nicht besonders ausgefeilt ist, wird ein weiterer, intelligenterer Bootloader benötigt. Im Bereich Linux wird hierfür zurzeit der sehr mächtige `grub2` eingesetzt ([http://www.gnu.org/software/grub/]). Kommt Linux auf einer PC-Hardware im Bereich eingebetteter Systeme zum Einsatz, wird häufig auch `syslinux` verwendet ([http://www.syslinux.org/]). Da das klassische PC-BIOS mit seinem historischen Ballast an seine technologischen Grenzen gestoßen ist, ersetzt UEFI (Unified Extensible Firmware Interface) auf modernen, 64-Bit-orientierten PC-Systemen das BIOS. UEFI unterscheidet zwischen den Boot-Services und den Runtime-Services. Im Rahmen der Boot-Services bringt es einen ausgeklügelteren Bootloader mit. Runtime-Services bieten über Treiberfunktionen für die geladene Systemsoftware plattformunabhängigen Zugriff auf Hardware.

Auf einer alternativen Plattform, die beispielsweise auf einer ARM-CPU basiert, setzen Entwickler gerne die Monitor-Software `Das U-Boot` oder `barebox` ein. Beide bieten sehr viele Funktionen, inklusive einem Bootloader, der das Booten über Netzwerk unterstützt. Außerdem sind alle genannten Bootloader in der Lage, nicht nur den Kernel, sondern auch das sogenannte Userland in Form eines Rootfilesystems in den Hauptspeicher zu laden.

3.2 Realzeitbetriebssysteme

Definition Betriebssystem

Aus Systemsicht ist ein Betriebssystem die Bezeichnung für alle Softwarekomponenten, die

- die Ausführung der Benutzerprogramme,
- die Verteilung der Betriebsmittel (z.B. Speicher, Prozessor, Dateien)

ermöglichen, steuern und überwachen.

Das Betriebssystem stellt dem Benutzer die Sicht eines *virtuellen Rechnerkerns* zur Verfügung, der einfacher zu benutzen ist als die reale Hardware. So steht aus Sicht eines Benutzers der Rechnerkern (die CPU) ihm allein zur Verfügung und er erhält einfachen Zugriff auf Ressourcen wie Speicher, Geräte und Dateien.

Das Betriebssystem besteht aus einem Betriebssystemkern und der Applikationsebene, auch Userland genannt, welches beispielsweise Dienstprogramme (z.B. eine Shell zur Eingabe von Kommandos) repräsentiert.

Tabelle 3-1 zeigt einige Anforderungen, die an ein Realzeitbetriebssystem gestellt werden.

Tabelle 3-1 *Anforderungen an Realzeitbetriebssysteme*

Anforderung	Bemerkungen
Zeitverhalten	Das Zeitverhalten muss deterministisch (berechenbar) sein. Das Betriebssystem muss möglichst kurze Reaktionszeiten ermöglichen.
Ressourcen-verbrauch	Anforderungen an Speicher (Hauptspeicher, Hintergrundspeicher) oder auch Rechenleistung sollten möglichst gering sein.
Zuverlässigkeit und Stabilität	Programmfehler dürfen das Betriebssystem und andere Programme nicht beeinflussen.
Sicherheit	Das Betriebssystem muss Möglichkeiten zum Zugangsschutz und Dateischutz bieten. Das ist nicht nur bei vernetzten Geräten notwendig.
Flexibilität und Kompatibilität	Es ist von Vorteil, wenn bereits existierende Software mit dem System zusammenarbeitet. Dazu muss das System einheitliche Standards unterstützen (z.B. POSIX). Außerdem sollte das System erweiterbar sein.
Portabilität	Das System selbst (nicht nur aufsetzende Applikationen) muss sich leicht auf unterschiedliche Hardware portieren lassen.
Skalierbarkeit	Durch das Weglassen oder Hinzufügen von Komponenten sollte das System skalierbar sein, sodass es auf unterschiedlich leistungsfähiger Hardware eingesetzt werden kann.

Abbildung 3-2 stellt den prinzipiellen Aufbau eines Betriebssystems dar. Die Treiberschicht abstrahiert Zugriffe auf die Hardware und stellt eine betriebssysteminterne, standardisierte Schnittstelle zur Verfügung, um in das System neue Hardwarekomponenten systemkonform zu integrieren. Gerade in Realzeitsystemen ist die Treiberschicht von zentraler Bedeutung. Denn nur wenn zusätzliche (proprietäre) Hardware systemkon-

form in das System integriert wird, kann der versprochene Determinismus des Betriebssystems auch gewährleistet werden. Anders formuliert: Ein schlecht geschriebener Gerätetreiber kann das Realzeitverhalten des gesamten Systems kompromittieren.

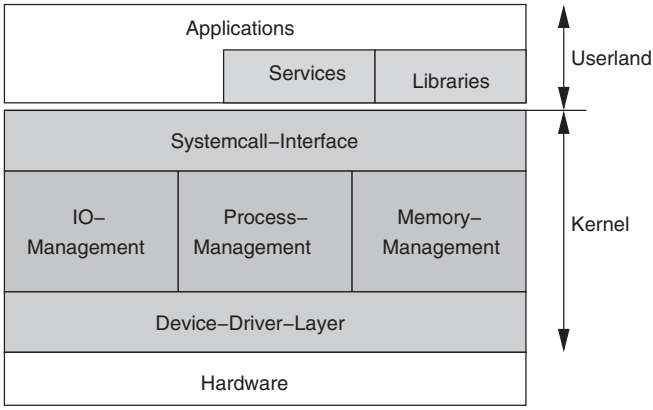


Abbildung 3-2
Betriebssystem-Architektur

Die Integration von Hardware in das System über die Treiberschicht ermöglicht darüber hinaus auch den systemkonformen und standardisierten Zugriff auf Hardware aus der Applikation heraus. Hierfür sorgt das I/O-Subsystem eines Betriebssystems. Bestimmte Gruppen von Hardware (z.B. Netzwerkkarten, SCSI, Filesysteme, PCI) benötigen ähnliche Funktionalitäten. Diese zur Verfügung zu stellen, ist ebenfalls Aufgabe des I/O-Subsystems.

Eine wichtige Aufgabe für Betriebssysteme ist die Verteilung der Ressource CPU (Rechnerkern) auf mehrere Tasks, das sogenannte Scheduling. Diese Aufgabe nimmt das *Prozess-Management* wahr.

Im Speichermanagement (Adressumsetzung und Speicherschutz) besteht eine weitere wesentliche Aufgabe des Betriebssystems. Es darf nicht möglich sein, dass eine normale Applikation ein komplettes Rechnersystem zum Absturz bringt. Dazu müssen aber die Speicherräume der unterschiedlichen Applikationen gegeneinander abgeschottet sein, ein Speicherschutzmanagement ist notwendig. Die entsprechende Komponente heißt Memory Management Unit (MMU). Speicherschutz wird durch die Hardware unterstützt.

Über sogenannte *Software-Interrupts* können Applikationen Dienste des Betriebssystems in Anspruch nehmen. Man spricht hierbei von *Systemcalls*. Die Schnittstelle innerhalb des Betriebssytems, die gemäß dem ausgelösten Systemcall den richtigen Dienst ausführt, wird als Systemcall-Interface bezeichnet.

Die bisher genannten Blöcke gehören alle zum sogenannten Betriebssystemkern. Auf dem dem Kern gegenüberstehenden Userland be-

finden sich die Applikationen, Dienstprogramme und Libraries. Dienstprogramme sind zum Betrieb des Rechnersystems notwendig. Beispielsweise werden für die Konfiguration des Systems (Zuteilung von Netzwerkadressen u.Ä.) und auch zum Betrieb der Netzwerkdienste Dienstprogramme benötigt. Die Programme selbst greifen in den seltensten Fällen direkt über die Systemcall-Schnittstelle auf die Dienste des Betriebssystems zu. Im Regelfall sind diese in einer Bibliothek gekapselt, die Standardfunktionen (beispielsweise `open()`, `close()`, `read()` und `write()`) zur Verfügung stellt. Innerhalb der Library ist der Systemcall selbst auscodiert.

Natürlich sind die Dienstprogramme des Betriebssystems im strengen Sinne auch Applikationen. Es werden hier nur deshalb zwei Begriffe verwendet, um die zum Betriebssystem gehörigen Applikationen – die Daemonen, Services oder einfach Dienstprogramme – von den *selbst geschriebenen* Applikationen unterscheiden zu können.

3.2.1 Systemcalls

Der Betriebssystemkern stellt für die Applikationen Dienste, die sogenannten Systemcalls, zur Verfügung. Die Art und die Anzahl der Systemcalls entscheiden über die Möglichkeiten, die der Kernel bietet. Ein aktuelles Betriebssystem weist mehr als 300 Systemcalls aus. Beispielhaft finden Sie einige Systemcalls in Tabelle 3-2.

Tabelle 3-2 *Ausgewählte Systemcalls*

Systemcall	Funktion
`clone`	Neue Task erzeugen
`exit`	Eine Task beenden
`gettimeofday`	Zeit auslesen
`read`	Daten einlesen
`kill`	Anderen Tasks ein Signal schicken

Aktiviert werden die Systemcalls über das Systemcall-Interface, welches manchmal auch abgekürzt nur Syscall-Interface genannt wird. Typischerweise legt die Applikation dazu die für die Ausführung des Systemcalls benötigten Parameter in die Prozessorregister oder auf den Stack ab und löst dann einen Software-Interrupt aus. Dadurch wechselt der Prozessor in den priviligierten Modus und führt die zum Interrupt gehörende Interrupt-Service-Routine aus. Diese entnimmt die in den Registern oder auf dem Stack befindlichen Parameter. Ein zentraler Parameter ist die Nummer des Systemcalls – zum Beispiel vier für `write` –, der ausgeführt werden soll. Der Kernel aktiviert im Rahmen der Interrupt-Service-Routine die zugehörige Bearbeitungsfunktion.

Beispiel 3-1 zeigt, wie ein Systemcall programmtechnisch unter Linux realisiert wird. Auf einer X86-Plattform werden die Parameter in Register abgelegt, woraufhin mithilfe des Befehls `int` (Interrupt) der Software-Interrupt mit der Nummer `0x80` aufgerufen wird. Allerdings implementieren moderne Prozessoren inzwischen einen eigenen Befehl (`sysenter`), um Systemcalls noch schneller aktivieren zu können.

Beispiel 3-1 *Assembler-Programm zum Aufruf des Systemcalls write*

```
.text
.globl write_hello_world
write_hello_world:
    movl $4,%eax        ; //code fuer "write" systemcall
    movl $1,%ebx        ; //file descriptor fd (1=stdout)
    movl $message,%ecx  ; //Adresse des Textes (buffer)
    movl $12,%edx       ; //Laenge des auszugebenden Textes
    int $0x80           ; //SW-Interrupt, Auftrag an das BS
    ret
.data
message:
    .ascii "Hello World\n"
```

Für die meisten Systemcalls gibt es in der Standard-C-Bibliothek Wrapper-Funktionen, die den Assemblercode implementieren. Manchmal jedoch stehen Bibliotheken nicht zur Verfügung. Dann bleibt nur noch übrig, den Aufruf des Systemcalls selbst zu realisieren.

3.2.2 Taskmanagement

Zentrale Aufgabe des Taskmanagements ist die Verteilung der Ressource CPU. Das Taskmanagement ermöglicht damit die quasi-parallele und auf Mehrkernmaschinen auch die real-parallele Verarbeitung mehrerer Rechenprozesse.

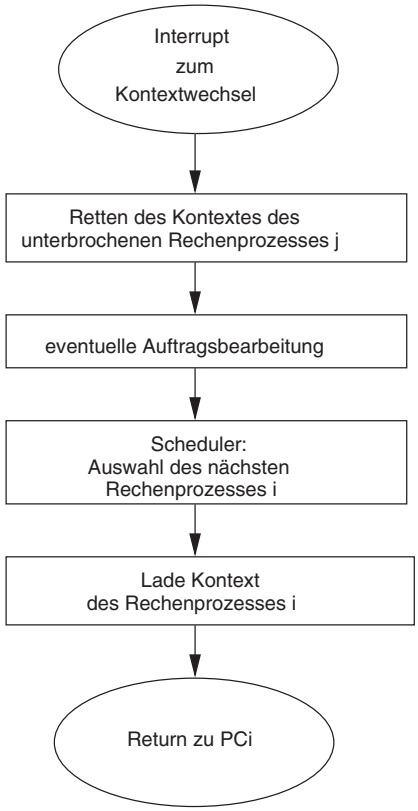


Abbildung 3-3
Interrupt-Service-Routine in einem Realzeitbetriebssystem

Die Fähigkeit der CPU, den normalen Programmablauf zu unterbrechen, wenn ein Interrupt auftritt, wird genutzt, um die quasi-parallele Bearbeitung mehrerer Programme (Tasks) zu ermöglichen. Dazu wird bei jedem Interrupt zunächst eine vom Betriebssystem zur Verfügung gestellte ISR (Abbildung 3-3) aufgerufen. In dieser ISR werden die Prozessorregister und die durch die CPU auf den Stack abgelegten Werte (Rücksprungadresse und Prozessorflags) in eine zum gerade aktiven Programm gehörige Datenstruktur (Task Control Block, Abbildung 3-4) abgelegt (gerettet). Danach kann die eigentliche Ursache des Interrupts bearbeitet werden. Gegen Ende der Interruptbearbeitung wird ein nächstes Programm ausgewählt (Scheduling), das nach Abschluss der ISR fortgesetzt werden soll. Dazu werden die Prozessorregister mit den Kopien der Register zum Zeitpunkt der letzten Unterbrechung der Task geladen. Flags und insbesondere die Rücksprungadresse (Programmcounter) werden so auf dem Stack abgelegt, dass die vom Prozessor bei Eintritt des Interrupts abgelegten Werte überschrieben werden und es wird schließlich der Befehl `Return Interrupt` ausgeführt. Dabei wird der Befehlszähler mit der *neuen* Rücksprungadresse geladen und die Bearbeitung der *neuen* Task fortgesetzt. Die Aktivierung der durch den Sche-

duler ausgewählten Task nennt man Contextswitch oder auch Dispatching.

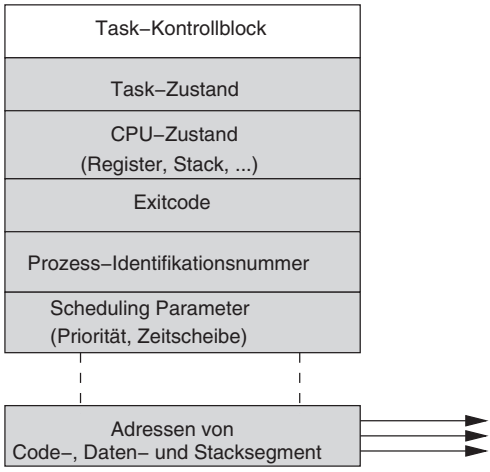


Abbildung 3-4
Task Control Block (TCB)

Der Unterschied zu einer gewöhnlichen ISR besteht darin, dass

1. die Register nicht auf den Stack gerettet werden, sondern in die Speicherzellen des Task Control Block (TCB) der gerade aktiven Task. Die Adresse des TCB der gerade aktiven Task ist im Regelfall in einer globalen Variablen (`current`) abgespeichert.
2. beim Contextswitch die Register des Prozessors nicht zwangsläufig mit den Werten geladen werden, die vor der Unterbrechung geladen waren. Stattdessen werden die Register aus dem TCB genommen, den der Scheduler ausgewählt hat.
3. die Rücksprungadresse auf dem Stack *manipuliert* wird, das heißt, der durch den Prozessor dort bei der Unterbrechung abgelegte Wert wird mit dem Wert für den PC überschrieben, der sich in dem TCB der neuen aktiven Task befindet.

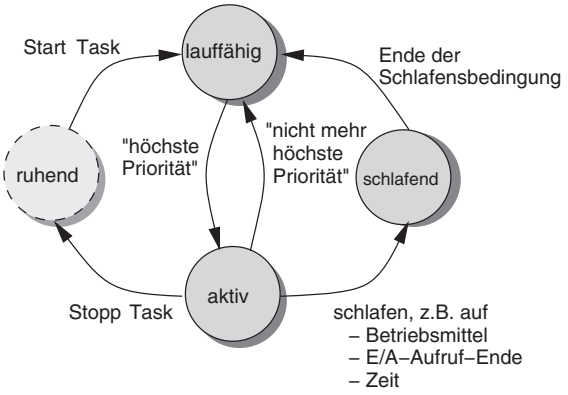
Die den Zustand einer Task (also des im System instanzierten Programms) genau beschreibende Datenstruktur *Task Control Block* speichert unter anderem die folgenden Informationen:

- Speicherplätze, in denen jeweils alle Prozessorzustände (Registerinhalte) abgelegt werden, mit welchen diese Task zum letzten Mal unterbrochen wurde.
- Angaben über den Zustand der Task, gegebenenfalls über die Bedingungen, auf die die Task gerade wartet.
- Angaben über die Priorität der Task.

Der Task Control Block spezifiziert damit eine Task eindeutig.

Anhand der Informationen im TCB entscheidet der Scheduler, ob eine Task die CPU zugeteilt bekommt oder nicht.

Abbildung 3-5
Task-Zustände



Eine der wichtigsten Informationen im TCB zur Auswahl des nächsten Rechenprozesses ist der Zustand der Task. Jede Task befindet sich in einem von vier möglichen Zuständen, wie in Abbildung 3-5 ersichtlich.

Lauffähig (auch rechenbereit genannt). Der Scheduler wählt aus der Liste der Prozesse denjenigen lauffähigen Prozess als Nächstes zur Bearbeitung aus, der die höchste Priorität hat. Mehrere Tasks im System können sich im Zustand lauffähig befinden.

Aktiv. Immer nur so viele Tasks im System können sich im Zustand *aktiv* befinden, wie es Rechnerkerne (Cores) gibt. Der aktiven Task wird ein CPU-Core zugeteilt und sie wird ausgeführt, bis sie entweder

- sich selbst beendet (in den Zustand *ruhend*) versetzt oder
- auf ein Betriebsmittel schlafen muss (z.B. auf das Ende eines I/O-Aufrufes) oder
- nicht mehr die höchste Priorität hat, da beispielsweise die Wartebedingung eines höherprioren Prozesses erfüllt wurde.

Schlafend. Eine Task wird in den Zustand *schlafend* versetzt, wenn nicht mehr alle Bedingungen zur direkten Ausführung erfüllt sind. Eine Task kann dabei auf unterschiedliche Bedingungen warten, beispielsweise auf das Ende von I/O-Aufrufen, auf den Ablauf einer definierten Zeitspanne oder auf das Freiwerden sonstiger Betriebsmittel.

Ruhend oder terminiert. Bei *ruhend* handelt es sich im eigentlichen Sinn nicht um einen Zustand, sondern vielmehr um einen *Metazustand*. Bevor ein Rechenprozess überhaupt bearbeitet wird, befindet er sich im Metazustand *ruhend*. Aus diesem Zustand kommt die Task nur, wenn sie durch das Betriebssystem (welches im Regelfall durch einen anderen Rechenprozess dazu aufgefordert wurde) *gestartet* wird. Im Metazu-

stand *ruhend* existiert im Kernel noch kein TCB für die Task. Der Metazustand *ruhend* wird oft auch als Zustand *terminiert* bezeichnet.

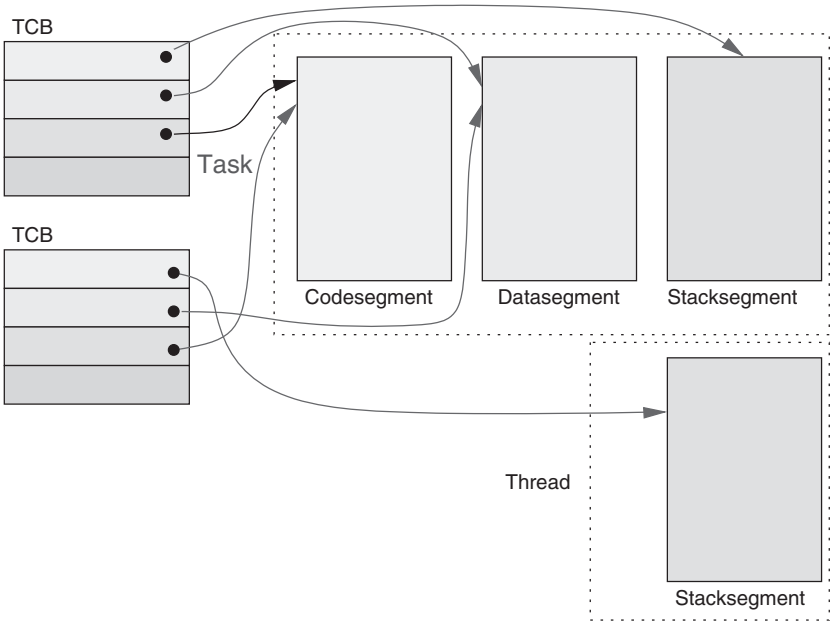


Abbildung 3-6
Speicherbereiche einer Task

Der zu einer Task gehörende Speicher ist typischerweise in mehrere Bereiche, in jeweils mindestens ein Code-, ein Daten- und ein Stacksegment, eingeteilt (Abbildung 3-6). Während Stacksegmente für jede Task exklusiv vorliegen müssen, können Daten- und Codesegmente von zwei oder mehr Tasks auch gemeinsam genutzt werden. Eine besondere Rolle für die Klassifizierung von Tasks spielt dabei das Datensegment. Nutzt eine Task exklusiv, also nur für sich, ein Datensegment, nennt man diese Task *Prozess*. Wird demgegenüber ein Datensegment von zwei oder mehr Tasks gemeinsam genutzt, nennt man diese Tasks Threads, die eine Threadgruppe bilden. Threads einer Threadgruppe nutzen auch das Codesegment gemeinsam.

Tabelle 3-3
Vergleich von Prozess und Thread

Prozess	Thread
Eigener TCB	Eigener TCB
Eigenes oder gemeinsam genutztes Codesegment	Gemeinsam genutztes Codesegment
Eigenes Datensegment	Gemeinsam genutztes Datensegment
Eigenes Stacksegment	Eigenes Stacksegment

Threads weisen gegenüber den Prozessen den Vorteil auf, schneller erzeugt zu werden – es muss neben dem TCB nur ein neues Stacksegment

angelegt werden. Daher nennt man sie auch *leichtgewichtige Tasks*. Außerdem ermöglichen sie eine einfache Inter-Prozess-Kommunikation über das gemeinsam genutzte Datensegment. Prozesse haben den Vorteil, mehr Sicherheit zu bieten. Ein amoklaufender Thread kann das Datensegment überschreiben und damit auch die übrigen Threads der Threadgruppe in einen inkonsistenten Zustand bringen.

Für das Erzeugen von Threads und das Erzeugen von Prozessen stellt das Betriebssystem Systemcalls zur Verfügung (siehe Abschnitt 4.2). Diese Systemcalls legen einen neuen TCB an und kopieren – falls notwendig – die zugehörigen Speicherbereiche. Bei Prozessen wird das Datensegment kopiert und nur ein Stacksegment angelegt, bei Threads wird nur ein neues Stacksegment angelegt.

3.2.2.1 Klassifizierung der Scheduling-Verfahren

Scheduling nennt man das Verfahren, den Tasks Prozessoren und Ressourcen zuzuordnen. Den Plan, welcher die genaue zeitliche Zuordnung der Tasks zu den Prozessoren und Ressourcen festlegt, nennt man Scheduling-Plan oder Scheduling-Tabelle. Aus diesem Plan muss hervorgehen, wann eine betrachtete Task auf einem Prozessor ausgeführt wird. In einem Realzeitsystem ist es Ziel, die Tasks so zu verteilen, dass alle maximalen Deadlines eingehalten werden. Das System arbeitet dann korrekt (korrekter Scheduling-Plan).

Ob für eine Menge von Tasks ein korrekter Scheduling-Plan existiert, wird mit einem Einplanbarkeitstest untersucht. Während der Schedulability-Test lediglich die Möglichkeit einer fristgerechten Planung eröffnet, sollte ein Scheduling-Plan die gesamte Information repräsentieren, die für eine Zuordnung von Tasks zu Prozessoren notwendig ist. Der Scheduling-Plan definiert den Zeitbereich von 0 bis zu einem gewissen Zeitpunkt t, ab dem sich der Scheduling-Plan wiederholt. Diese Periode [0,t] wird Hyperperiode genannt.

Damit die Tasks überhaupt gescheduled werden können, muss die Auslastung für jeden CPU-Kern kleiner 100 Prozent sein (Scheduling-Test, Gleichung 3-1):

Gleichung 3-1
Auslastung

$$\rho_{ges} = \sum_{j=1}^{n} \frac{t_{Emax,j}}{t_{Pmin,j}} \leq c$$

Die Einzelauslastungen der Tasks (Quotient aus Ausführungszeit und Periode) werden summiert und die Summe darf nicht größer als die vom System zur Verfügung gestellte Leistung c sein. Bei einem Singlecore-System gilt c = 1. Ist die Summe größer als c, ist das System überlastet.

In einem überlasteten System kann kein korrekter Scheduling-Plan existieren.

Ein Scheduling-Verfahren wird als *optimal* bezeichnet, wenn es in jeder Situation einen korrekten Scheduling-Plan findet – vorausgesetzt, es existiert mindestens einer. Das impliziert, dass das System nicht überlastet ist. Ein Scheduling-Verfahren ist also *nicht optimal*, wenn es keinen korrekten Scheduling-Plan findet, obwohl mindestens einer existiert.

Fokussiert auf die Zuordnung von Tasks auf Prozessoren, lassen sich Scheduling-Verfahren klassifizieren (Abbildung 3-7).

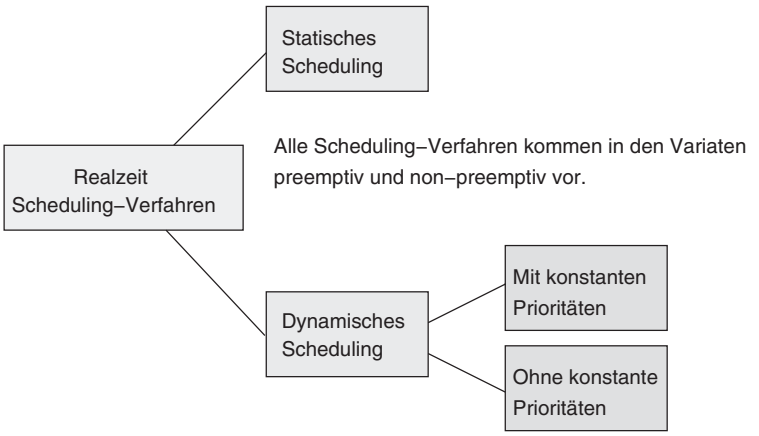


Abbildung 3-7
Klassifizierungsmöglichkeiten von Scheduling-Verfahren

- Beim *statischen Scheduling (Offline Scheduling)* wird der komplette Scheduling-Plan berechnet, bevor das System startet. Entsprechend aufwendig kann man die Berechnung, die offline erfolgt, gestalten. Notwendig ist die Kenntnis verschiedener temporaler Parameter des Systems wie die Prozesszeit, die Ausführungszeit, die minimale und die maximale Deadline und die von der Task benötigten Ressourcen zu jeder Zeit. Statisches Scheduling führt zu deterministischen Systemen.
- Beim *dynamischen Scheduling* wird bei Aufruf des Schedulers die nächste Task bestimmt, die ausgeführt werden soll. Der Scheduling-Plan wird nicht im Vorhinein erstellt, sondern entsteht zur Laufzeit anhand der aktuellen Lastsituation. Ein dynamisches Scheduling-Verfahren analysiert bei jedem Aufruf die aktuelle Lastsituation anhand der zur Verfügung stehenden Parameter. Das Verfahren ist gegenüber einem statischen Scheduling viel flexibler, wenn auch nicht immer optimal. Schließlich können die zukünftigen Anforderungen nicht in die Erstellung des aktuellen Scheduling-Plans mit einbezogen werden (da ja nicht bekannt!). Prinzipiell lassen sich zwei Arten

des dynamischen Schedulings unterscheiden, je nachdem, ob sie auf konstanten Prioritäten basieren oder eben nicht.

- Mit konstanten Prioritäten: Die zu den Rechenzeitanforderungen gehörenden Tasks bekommen jeweils eine feste Priorität, die sich typischerweise nicht ändert (Ausnahme Prioritätsvererbung).
- Ohne konstante Prioritäten: Die Wahl der nächsten Task wird aufgrund eines anderen Kriteriums, beispielsweise der jeweils zugehörigen maximalen Deadline, durchgeführt. Implementierungstechnisch, aber auch formaltechnisch wird dieses andere Kriterium zuweilen auf Prioritäten abgebildet. In diesem Fall variiert die Priorität der Task aber während der Laufzeit - unabhängig von einer möglichen Prioritätsvererbung. Daher spricht man zuweilen auch von Scheduling-Verfahren mit variablen Prioritäten.

- *Non-preemptive*: Hier wird die gestartete Task bis zur Vollendung ausgeführt. Die Scheduling-Entscheidung erfolgt, wenn sich die Task beendet, schlafen legt oder freiwillig den Scheduler aktiviert. Man spricht auch von *kooperativem* Scheduling, da andere Tasks nur dann ausgeführt werden, wenn die aktive Task von sich aus die CPU wieder freigibt.
- *Preemptive*: Bei präemptiven Algorithmen kann die laufende Task jederzeit unterbrochen werden, um den Prozessor einer anderen Task zuzuordnen. Dies erfolgt gemäß den durch den Algorithmus definierten Regeln.

3.2.2.2 Statisches Singlecore-Scheduling

Statisches Scheduling wird bei Systemen mit harten Realzeitanforderungen und meist einfachem, deterministischem Laufzeitsystem eingesetzt. Systeme dieser Art werden auch zeitgesteuerte Systeme genannt, da der Scheduling-Plan im Vorhinein aufgestellt und als Tabelle dem System bekannt gegeben wird. Im System selbst ist lediglich ein Dispatcher implementiert, der entsprechend zu den Zeitpunkten in der Tabelle die Tasks ausführt. Die Tasks besitzen in diesen Systemen keine Prioritäten. Stattdessen werden die Tasks zu bestimmten Zeitpunkten aktiviert und sofort ausgeführt. Läuft zum Zeitpunkt der Aktivierung einer Tasks noch eine andere Task, so wird diese unterbrochen.

Für den Scheduling-Plan gibt es zwei Ausprägungen:

1. den zeitgesteuerten Scheduling-Plan
2. ein auf konstanten Zeitslots beruhender Scheduling-Plan

Task	t_{Pmin}	t_{Dmin}	t_{Dmax}	t_{Emin}	t_{Emax}
V	20 ms	0 ms	20 ms	1 ms	5 ms
G	40 ms	0 ms	40 ms	1 ms	10 ms
M	30 ms	0 ms	30 ms	1 ms	10 ms

Tabelle 3-4
Beispieldaten für das Aufstellen eines statischen Scheduling-Plans

Zeitgesteuerter Scheduling-Plan

Für dieses Verfahren wird ein Scheduling-Plan aufgestellt, der die Startzeitpunkte zusammen mit den zu startenden Tasks enthält. Der Plan umfasst genau eine Hyperperiode. Das Verfahren ist aufwendig, weil der Scheduler mit jedem Eintrag im Plan den Timer neu programmieren muss. Der Systemarchitekt kann den Plan selbstständig (quasi per Hand) oder mithilfe eines Algorithmus erstellen. In Abbildung 3-8, ist eine mögliche Rechnerkernbelegung für das Taskset aus Tabelle 3-4, dargestellt. Der zugehörige Scheduling-Plan findet sich in Tabelle 3-5.

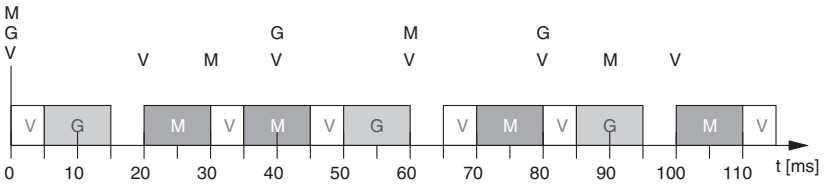


Abbildung 3-8
Rechnerkernbelegung: statisches, zeitgesteuertes Scheduling

Releasetime	Task
0	V
5	G
20	M
30	V
35	M
45	V
50	G
65	V
70	M
80	V
85	G
100	M
110	V

Tabelle 3-5
Scheduling-Plan für zeitgesteuertes Scheduling

Zeitgesteuerter Scheduling-Plan mit konstanten Zeitslots

Ein periodisch auftretender Interrupt stößt im System die Abarbeitung der Tasks an. Der Zeitraum zwischen zwei Interrupts (Zeitslot) wird dabei als Frame oder auch Minor Cycle bezeichnet. Die Dauer eines Fra-

mes und die Zuordnung der Tasks auf die Frames (Scheduling-Plan) wird vom Systemarchitekten festgelegt. Innerhalb eines Frames können – falls deren Verarbeitungszeiten in Summe in den Frame passen – auch mehrere Tasks nacheinander abgearbeitet werden. Es wird davon ausgegangen, dass die abzuarbeitenden Tasks periodisch (also mit konstantem t_P) sind. Durch die Periodizität wiederholen sich irgendwann die Vorgänge. Die Zeit, in der sich die Vorgänge wiederholen, wird Hyperperiode beziehungsweise Major Cycle genannt. Die Scheduling-Tabelle deckt genau eine Hyperperiode ab. Existieren neben den periodischen Tasks auch aperiodische Tasks (also welche mit variabler t_P), werden diese in freien Zeitintervallen abgearbeitet (eventuell auch mit Unterbrechung, Preemption).

Zur Bestimmung der Scheduling-Parameter (Frame-Länge und Scheduling-Tabelle) wird zunächst die Hyperperiode bestimmt. Frame-Längen, die eventuell eine fristgerechte Abarbeitung ermöglichen, können dann über die folgenden Randbedingungen eingegrenzt werden:

1. Zur Vermeidung von Preemption sollte der Frame lang genug sein, sodass auch die Task mit der größten Verarbeitungszeit innerhalb eines Frames abgearbeitet werden kann. Jedoch darf der Frame nicht so groß werden, dass er über das Minimum der geforderten Deadlines aller Tasks hinausgeht.

Gleichung 3-2
Untere und obere Frame-Grenze bestimmen

$$f \geq \max_{1 \leq j \leq n} (t_{Emax,j})$$
$$f \leq \min_{1 \leq j \leq n} (t_{Dmax,j})$$

2. Zur Begrenzung der Länge der Scheduling-Tabelle muss f ein ganzzahliger Teiler der Hyperperiode H sein. Das ist dann gegeben, wenn f ein ganzzahliger Teiler der Periode (t_{Pmin}) einer der beteiligten Tasks ist.

Gleichung 3-3
Begrenzung der Länge der Frame-Tabelle

$$\exists i : mod(t_{p,i}, f) = 0$$

3. *f* muss klein genug gewählt werden, damit zwischen der Release-Time und der maximalen Deadline einer jeden Task zumindest ein Frame liegt. Denn:
   - Liegt die Releasetime einer Task innerhalb eines Frames, so wird diese Task erst zu Beginn des nächsten Frames released.
   - Hat eine Task ihre maximale Deadline im Frame k+1, muss sie notwendigerweise im Frame k ausgeführt werden.

$$2 * f - \gcd(t_{Pmin,i}, f) \leq t_{Dmax,i}$$

Gleichung 3-4
Mögliche f für alle Tasks prüfen

> Die Funktion `gcd()` steht für *Greatest Common Divisor*, also den größten gemeinsamen Teiler.

Eine Frame-Länge, die die fristgerechte Abarbeitung aller Tasks ermöglicht, muss sämtliche Randbedingungen für alle Tasks erfüllen. Allerdings ist das Erfüllen sämtlicher Bedingungen noch kein Garant dafür, dass es mit dieser Frame-Länge auch einen gültigen Scheduling-Plan gibt.

Die Bestimmung der Frame-Länge und das Aufstellen eines zugehörigen Scheduling-Plans soll anhand des in Tabelle 3-4 gelisteten Tasksets exemplarisch vorgeführt werden.

Bedingung 1. Die Frame-Länge f muss größer oder gleich dem Maximum der drei Verarbeitungszeiten sein: max ($t_{Emax,V}$, $t_{Emax,G}$, $t_{Emax,M}$) = 10 ms. Aus der Menge der maximalen Deadlines gibt Task V die obere Frame-Grenze von t = 20 ms vor. Die Frame-Größen bewegen sich somit im Intervall [10, 20] ms.

Bedingung 2. f muss ein ganzzahliger Teiler von einer der drei minimalen Prozesszeiten (Periode) sein:

$t_{Pmin,V}$ = 20 ms; ganzzahlige Teiler sind 1, 2, 4, 5, 10 und 20 (jeweils in ms).

$t_{Pmin,G}$ = 40 ms; ganzzahlige Teiler sind 1, 2, 4, 5, 8, 10, 20 und 40 (jeweils in ms).

$t_{Pmin,M}$ = 30 ms; ganzzahlige Teiler sind 1, 2, 3, 5, 6, 10 und 15 (jeweils in ms).

Gemäß dieser Bedingung muss f 1, 2, 3, 4, 5, 8, 10, 15, 20 oder 40 ms lang sein.

Bedingung 3. Für jede der drei Tasks muss die Gleichung der Bedingung drei für jedes f durchgerechnet und überprüft werden. Die Bedingungen 1 und 2 schränken jedoch die möglichen Werte für f auf 10 ms, 15 ms oder 20 ms ein.

$$2 \cdot f - ggt(t_{Pmin,i}, f) \leq t_{Dmax,i}$$

$$V\ mit\ f = 10ms: \quad 2{\cdot}10ms - ggt(20ms, 10ms) \leq 20ms;\ \ 20ms - 10ms \leq 20ms\ (OK)$$

$$V\ mit\ f = 15ms: \quad 2 \cdot 15ms - ggt(20ms, 15ms) \leq 20ms;\ \ 30ms - 5ms > 20ms$$

$$V\ mit\ f = 20ms: \quad 2{\cdot}20ms - ggt(20ms, 20ms) \leq 20ms;\ \ 40ms - 20ms \leq 20ms\ (OK)$$

$$G\ mit\ f = 10ms: \quad 2{\cdot}10ms - ggt(40ms, 10ms) \leq 40ms;\ \ 20ms - 10ms \leq 40ms\ (OK)$$

$$G\ mit\ f = 15ms: \quad 2{\cdot}15ms - ggt(40ms, 15ms) \leq 40ms;\ \ 30ms - 5ms \leq 40ms\ (OK)$$

$$G\ mit\ f = 20ms: \quad 2{\cdot}20ms - ggt(40ms, 20ms) \leq 40ms;\ \ 40ms - 20ms \leq 40ms\ (OK)$$

$$M\ mit\ f = 10ms: \quad 2{\cdot}10ms - ggt(30ms, 10ms) \leq 30ms;\ \ 20ms - 10ms \leq 30ms\ (OK)$$

$$M\ mit\ f = 15ms: \quad 2{\cdot}15ms - ggt(30ms, 15ms) \leq 30ms;\ \ 30ms - 15ms \leq 30ms\ (OK)$$

$$M\ mit\ f = 20ms: \quad 2{\cdot}20ms - ggt(30ms, 20ms) \leq 30ms;\ \ 40ms - 10ms \leq 30ms\ (OK)$$

Sämtliche Bedingungen sind nur für Frame-Längen von f = 10 ms und f = 20 ms erfüllt. Mit diesen Frame-Längen kann jetzt versucht werden, einen Scheduling-Plan aufzustellen. Grafisch kann dies beispielsweise durch das Erstellen einer Rechnerkernbelegung erfolgen. Zunächst ist die Zeitachse mit Kennzeichnung der Frames für eine Hyperperiode (hier 120 ms) zu zeichnen, danach die Release-Times der einzelnen Rechenzeitanforderungen.

Im vorliegenden Fall gibt es auch mit einer Frame-Länge von f = 10 ms und f = 20 ms keinen gültigen Scheduling-Plan. In dieser Situation kann der Systemarchitekt versuchen, einen Rechenprozess in mehrere Teile (sogenannte Slices) aufzusplitten. Dies bietet sich als potenzielle Realisierungsvariante an, wenn die oben genannten drei Constraints nicht einzuhalten sind.

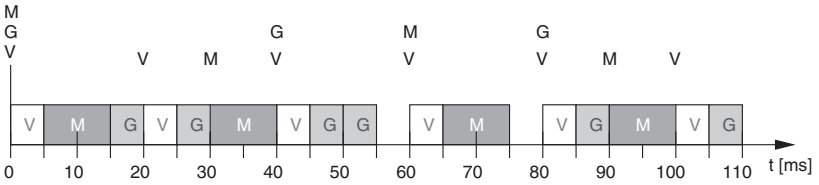


Abbildung 3-9 *Rechnerkernbelegung: statisches, periodisches Scheduling*

Im vorliegenden Fall bietet es sich an, die Task G in zwei Slices zu teilen, die nacheinander abgearbeitet werden. Die Verarbeitungszeit von Slice 1 beträgt $t_{E,G1}$ = 5 ms, die von Slice 2 ebenfalls $t_{E,G2}$ = 5 ms. Damit ergibt sich die in Abbildung 3-9 dargestellte Rechnerkernbelegung. Andere Rechnerkernbelegungen sind ebenfalls möglich. Der zur Rechnerkernbelegung gehörende Scheduling-Plan ist in Tabelle 3-6 zu finden.

Wie zu sehen ist, besteht der Major Cycle aus sechs Minor-Zyklen (Frames).

Tabelle 3-6
Scheduling-Plan

Frame-Nummer	Tasks
0	V, M, G1
1	V, G2, M
2	V, G1, G2
3	V, M
4	V, G1, M
5	V, G2

Was wird geschehen, wenn eine Task aufgrund von Soft- oder Hardwarefehlern doch länger benötigt als eingeplant? Diesen Fall einer Deadline-Verletzung nennt man in zeitgesteuerten Systemen auch Frame Overrun. Folgende Ansätze bieten sich an:

Mit der Ausführung fortfahren
Eine Möglichkeit besteht darin, die Task weiter auszuführen, wodurch sich die Ausführungen der Tasks im nächsten Frame jedoch verzögern. Diese Möglichkeit bietet sich nur dann an, wenn das "verspätete" Ergebnis der Task sehr wichtig ist und auch verspätet einen Nutzen hat.

Abbrechen der Instanz der Task
Das Abbrechen der Task führt zu einer rechtzeitigen Ausführung der Tasks in den folgenden Frames. Diese Möglichkeit bietet sich für Tasks an, deren verspätetes Ergebnis keinen Nutzen hat.

Fehlerroutine aufrufen
Zwischen dem weiteren Ausführen der Task und dem Abbrechen gibt es die Möglichkeit, eine spezielle Routine der Task bei einer Deadline-Verletzung aufzurufen. Manche Systeme (z.B. OSEK-Time) sind so konfigurierbar, dass nach dieser Routine das Betriebssystem sicherheitshalber neu gestartet wird. Eine Fehlerbehandlung wird somit ausgeführt, aber zum Synchronisieren aller Tasks auf die Frame-Grenzen wird das System neu gestartet.

Freie Zeitbereiche der folgenden Frames nutzen
Eine wiederholte Terminierung einer Task oder das Neustarten des Systems birgt die Gefahr, dass sich das Gesamtsystem dauerhaft in einem Fehlerzustand befindet. Die nicht genutzte Zeit in den folgenden Frames kann für die Ausführung des restlichen Codes der Task oder auch der Fehlerroutine der Task genutzt werden. Eine Reaktion findet somit statt, aber zu einem nicht definierbaren späteren Zeitpunkt.

Die vorgestellten Möglichkeiten belegen, dass es keine optimale Lösung für die Deadline-Verletzung einer Task gibt. Dies zeigt umso deutlicher, wie wichtig eine korrekte Bestimmung des Scheduling-Plans in einem zeitgesteuerten System ist.

OSEK/VDX

Das OSEK/VDX-Gremium besteht aus Automobilherstellern, Softwarefirmen und Automobilzulieferern. Das Gremium hat eine Reihe von Standards für den Automobilbereich geschaffen, wobei mit OSEK-Time und OSEK-OS Standards für zeitgesteuerte und ereignisgesteuerte Betriebssysteme existieren.

OSEK-Time stellt das Konzept eines zeitgesteuerten Betriebssystems dar (siehe Statisches Singlecore-Scheduling (Seite 46)). Das OSEK-Time-Task-Konzept sieht vor, dass Tasks sequenziell abgearbeitet werden. Da keine Events, Alarme, Counter und Ressourcen vorhanden sind, kann in einer Task nicht auf externe Ereignisse gewartet werden. Die zeitliche und sofortige Aktivierung von Tasks ermöglicht, Aufgaben hochgradig zeitsynchron durchzuführen.

Eine Besonderheit der Interruptverwaltung von OSEK-Time ist die Definition von Zeitintervallen für Interrupts. Interrupts werden nach ihrem Auftreten für dieses Zeitintervall gesperrt. Das zweimalige Verarbeiten eines Interrupts innerhalb seiner konfigurierten Zeitspanne (vgl. Frame in Statisches Singlecore-Scheduling (Seite 46)) wird damit verhindert.

OSEK-OS stellt das Konzept eines ereignisgesteuerten Betriebssystems dar (Dynamisches Singlecore-Scheduling (Seite 53)). OSEK-OS hat gegenüber OSEK-Time einen prioritätenbasierten Scheduler implementiert. Es unterstützt zwei Arten von Tasks:

- Basic Tasks haben die Eigenschaft, dass sie den Prozessor nur freigeben, wenn sie terminieren oder das Betriebssystem sie unterbricht, um einen Task oder eine Interrupt-Service-Routine mit höherer Priorität auszuführen.
- Extended Tasks sind eine Erweiterung der Basic Tasks und können gegenüber einer Basic Task auf ein Ereignis warten. Solange die Task sich im Zustand Waiting befindet, führt der Prozessor Tasks mit niedriger Priorität aus. Extended Tasks haben somit die Möglichkeit, mit Ressourcen zu arbeiten, die auch von anderen Tasks genutzt werden.

Tasks können full-preemptiv oder non-preemptiv sein. Eine full-preemptiv Task wird von höherprioren Tasks jederzeit unterbrochen. Non-preemptiv bedeutet, dass die Task selbst die CPU abgeben muss (zum Beispiel durch einen `wait()`-Aufruf), der Scheduler unterbricht die Task nie. Der Programmierer kann im Code der Task Abschnitte definieren, die full-preemptiv oder non-preemptiv sind.

OSEK-OS hat das Priority Ceiling Protocol für Tasks implementiert und für Interrupts erweitert. Um die Ceiling-Priorität einer Ressource festzulegen, werden dazu den Interrupts virtuelle Prioritäten zugewiesen (vgl. Priority Ceiling Protocol (Seite 30)).

Die OSEK-Time-Spezifikation erlaubt es übrigens, ein gemischtes System aus OSEK-Time und OSEK-OS zu implementieren.

3.2.2.3 Dynamisches Singlecore-Scheduling

Im Folgenden werden Singlecore-Scheduling-Verfahren aus dem Bereich des dynamischen Schedulings vorgestellt. Diese Verfahren findet man typischerweise in ereignisgesteuerten Betriebssystemen. Dabei gibt es zunächst vier Grundverfahren, die in realen Implementierungen in unterschiedlichen Ausprägungen vorkommen. Außerdem wird noch ein kombiniertes Scheduling-Verfahren vorgestellt, wie es heute häufig eingesetzt wird.

Anhand des Beispiels Fahrradcomputer wird in den folgenden Beschreibungen der bei den verschiedenen Singlecore-Scheduling-Verfahren entstehende Scheduling-Plan in Form einer Rechnerkernbelegung dargestellt. Auf dem Fahrradcomputer sollen drei Tasks berücksichtigt werden: Task V wird mit jeder Umdrehung des Vorderrades aufgerufen und ist für die Berechnung der aktuellen Parameter (Geschwindigkeit, zurückgelegte Strecke, Strecken- und Geschwindigkeitsprofil) zuständig. Die rechenintensive Task GUI steuert den Touchscreen an, auf dem die Ausgaben auf der einen und die Inputs des Users auf der anderen Seite zu verarbeiten sind. Die Task MONITORING schließlich überwacht die Systemzustände, bei einem Pedelec insbesondere auch Strom, Spannung, Ladezeiten und Restkilometer. Die Zeitparameter finden sich zusammengefasst in Tabelle 3-7.

Task	t_{Pmin}	t_{Dmin}	t_{Dmax}	t_{Emin}	t_{Emax}
V	20 ms	0 ms	20 ms	1 ms	5 ms
GUI	40 ms	0 ms	40 ms	1 ms	15 ms
MONITORING	30 ms	0 ms	30 ms	1 ms	10 ms

Tabelle 3-7 *Beispieldaten Taskgebilde Fahrradcomputer*

First Come First Serve (FCFS, FIFO)

Alle lauffähigen Tasks werden gemäß ihres Auftrittszeitpunktes in einer Queue (Warteschlange) eingehängt. Die Task, die vorn als Erste in der Queue ist, wird vom Scheduler ausgewählt (damit der Context-Switch die Task aktiviert). Diese benutzt die CPU, bis a) die Task sich beendet oder b) die Task sich schlafen legt. Wird die Task wieder aufgeweckt, unterbricht sie typischerweise nicht die gerade aktive Task, wird aber am *Anfang* der Queue eingehängt.

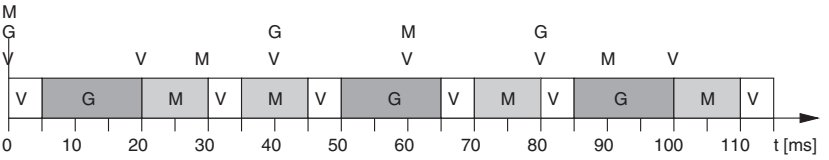
FCFS gehört zur Gruppe der kooperativen Scheduling-Verfahren. Bei diesen muss die Anwendung selbst kooperativ sein und durch Abgabe des Rechnerkerns das Scheduling unterstützen. Dazu dient ein Systemcall, der häufig `yield()` lautet.

Bewertung für Realzeitsysteme: Als eigenständiges Scheduling-Verfahren ist es für den Einsatz in Realzeitsystemen nicht geeignet. Eine einzige Task beziehungsweise ein Fehler innerhalb der Task kann das gesamte System blockieren.

Beispiel 3-2
First Come First Serve

Im Worst Case werden alle drei Rechenprozesse des Fahrradcomputers gleichzeitig lauffähig. Gemäß dem Schema werden sie dann wie in Abbildung 3-10 nacheinander abgearbeitet.

Abbildung 3-10
First Come First Serve



Zeitscheibenverfahren

Ähnlich wie bei FCFS werden alle lauffähigen Tasks gemäß ihres Auftrittszeitpunktes in eine Queue (Warteschlange) eingereiht. Die Task, die sich an erster Position in der Queue befindet, wird vom Scheduler ausgewählt und darauffolgend vom Contextswitch aktiviert. Diese Task benutzt die CPU, bis a) sich die Task beendet, b) sich die Task schlafen legt oder c) ihre Zeitscheibe abgelaufen ist.

Wird eine Task wieder aufgeweckt, reiht sie sich an das Ende der Warteschlange ein. Da alle Tasks eine gleich lange Zeitscheibe bekommen, handelt es sich um ein sehr faires Scheduling-Verfahren.

Die Länge einer Zeitscheibe bewegt sich typischerweise im Bereich von 10 ms bis 200 ms und kann häufig vom Systemarchitekten konfiguriert werden. Sie entscheidet auf der einen Seite über die Effizienz und auf der anderen Seite über die erreichbaren Reaktionszeiten. Je kürzer die Zeitscheiben gewählt werden, desto kürzer sind die Reaktionszeiten. Dadurch wird auf der anderen Seite der Scheduler häufiger aktiv, sodass das Verhältnis zwischen der Rechenzeit, die das System benötigt, und der Rechenzeit, die den Tasks zur Verfügung steht, ungünstig wird.

Beispiel 3-3
Zeitscheiben-verfahren

Die Länge einer Zeitscheibe betrage maximal 10 ms. Im Worst Case werden alle drei Rechenprozesse des Fahrradcomputers gleichzeitig lauffähig. Gemäß Schema werden sie wie in Abbildung 3-11 abgearbeitet.

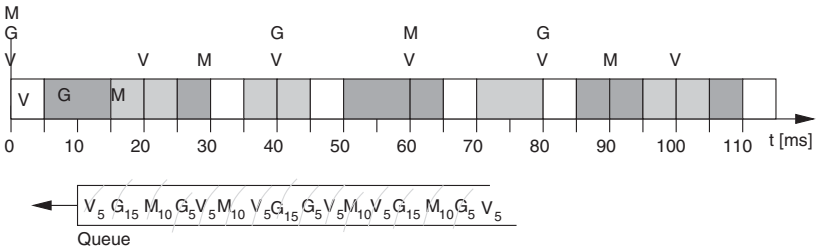


Abbildung 3-11
Zeitscheiben-verfahren

Um die Rechnerkernbelegung händisch zu bestimmen, simuliert man am einfachsten die Warteschlange. Sobald eine Task ihre Zeitscheiben genutzt hat, wird sie gestrichen. Ist sie nicht komplett abgearbeitet, wird sie mit der Restzeit wieder hinten in die Warteschlange eingehängt. Nutzt eine Task ihre Zeitscheibe nicht ganz (beispielsweise gleich zu Beginn Task V), darf die nächste lauffähige Task direkt mit ihrer Zeitscheibe beginnen. Bei dynamischem Scheduling wird typischerweise keine Rechenzeit verschenkt. Zum Zeitpunkt 20 ms wird Task M unterbrochen, da für Task V eine neue Rechenzeitanforderung eintrifft. Neue Rechenzeitanforderungen führen grundsätzlich zu einer Unterbrechung. Allerdings stellt der Scheduler fest, dass Task M die Zeitscheibe noch nicht vollständig genutzt hat, daher wird Task M weiter ausgeführt.

Prioritätengesteuertes Scheduling

Jeder im System befindlichen Task wird eine Priorität zugeteilt. Der Scheduler wählt aus der Liste der rechenbereiten Tasks diejenige aus, die die höchste Priorität hat. Diese Task darf die CPU benutzen, bis sie sich a) beendet, b) schlafen legt oder c) bis eine andere Task mit höherer Priorität lauffähig wird.

Prioritätengesteuertes Scheduling mit konstanten Prioritäten ist das klassische Verfahren für Realzeitsysteme.

Oft repräsentieren kleine Zahlen hohe Priorität. Bei POSIX beispielsweise verhält es sich genau umgekehrt. Von daher sollte der Entwickler vor der Verteilung der Prioritäten grundsätzlich die Dokumentation seines Systems zurate ziehen.

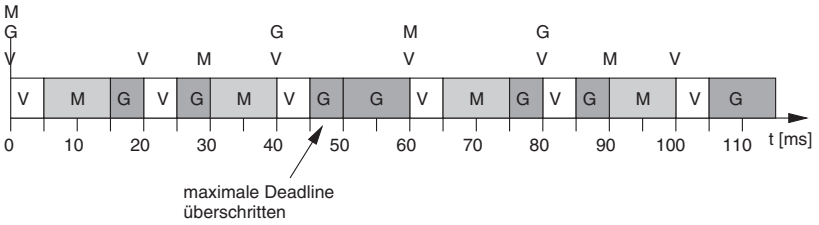
Die Wahl beziehungsweise Zuordnung einer geeigneten Priorität zu einer Task ist nicht immer trivial. In vielen Fällen können hierfür jedoch die Zeitparameter herangezogen werden. Daher bekommen die Tasks, die eine kurze Verarbeitungszeit und gleichzeitig(!) eine kurze maximale Deadline haben, eine hohe Priorität. Umgekehrt erhalten Tasks mit einer langen Verarbeitungszeit und gleichzeitig langer Deadline eine niedrige Priorität. Ist die maximale Deadline nicht bekannt, wird alternativ die Periode genommen.

Ist die Korrelation nicht gegeben, kann die Auslastung einer Task als ein Kriterium zur Prioritätenverteilung herangezogen werden: Je geringer die Auslastung, desto höher die Priorität. Anschließend ist mithilfe des Realzeitnachweises die Einhaltung der Realzeitbedingungen zu verifizieren. Sollten diese nicht eingehalten werden, wird die Verteilung der Prioritäten modifiziert. Dieser Vorgang wird so lange wiederholt, bis entweder eine funktionierende Verteilung gefunden werden konnte oder aber alle Variationen durchprobiert wurden. Im letzteren Fall gibt es keine geeignete Verteilung und das Problem ist mit einem prioritätengesteuerten Scheduling nicht lösbar.

Beispiel 3-4
Prioritätengesteuertes Scheduling

Zunächst müssen den drei Rechenprozessen Prioritäten zugeteilt werden. Dieses kann auf Basis der Korrelation zwischen Verarbeitungszeit und maximaler Deadline (beziehungsweise Prozesszeit) durchgeführt werden. Task V hat die kürzeste Verarbeitungszeit und gleichzeitig die kürzeste maximale Deadline. Daher bekommt V die Priorität 1. Task M bekommt die Priorität 2 und Task G die Priorität 3. Im Worst Case werden alle drei Tasks des Fahrradcomputers gleichzeitig lauffähig. Gemäß dem Schema werden sie wie in Abbildung 3-12 dargestellt abgearbeitet.

Abbildung 3-12
Prioritätengesteuertes Scheduling



Zur Erstellung der Rechnerkernbelegung kann man folgendermaßen vorgehen: Nach Eintragung der Rechenzeitanforderungen wird zunächst die Rechnerkernbelegung für die Task mit der höchsten Priorität eingezeichnet. Diese wird direkt ausgeführt, wenn die Rechenzeitanforderung auftritt. Danach kommt die Task mit der nächst niedrigeren Priorität an die Reihe. Diese wird an die nächste freie Stelle nach dem Auftrittszeitpunkt eingetragen. Das Prozedere wird so lange fortgesetzt, bis auch die Task mit der niedrigsten Priorität eingezeichnet wurde.
Im vorliegenden Fall kommt es zu einer Verletzung der maximalen Deadline. Die maximale Reaktionszeit für die Rechenzeitanforderung für Task G zum Zeitpunkt t = 0 beträgt 50 ms, wobei nur 40 ms erlaubt sind.

Deadline-Scheduling

Beim *Earliest Deadline First (EDF)*, kurz Deadline-Scheduling, wird aus der Liste der rechenbereiten Tasks diejenige ausgewählt, deren maximale Deadline dem Momentanzeitpunkt am nächsten ist. Anders ausgedrückt: Die Task wird ausgewählt und anschließend vom Context-Switch aktiviert, die als Erstes fertig sein muss. Eine Task darf so lange die CPU verwenden, bis sie sich a) beendet, b) schlafen legt oder c) bis eine andere Task lauffähig (rechenbereit) wird, die eine kürzere maximale Deadline aufweist (früher fertig sein muss).

Beim EDF handelt es sich um ein optimales Scheduling-Verfahren. Falls ein Problem auf einer gegebenen Hardware schritthaltend verarbeitet werden kann, dann ist dies mit EDF möglich. Demnach müsste es in der Praxis das vorherrschende Verfahren sein. Da jedoch in vielen Fällen Deadlines nicht bekannt sind und viele Hersteller der Systemsoftware den Algorithmus nicht implementiert haben, ist es bisher leider nur vereinzelt in echten Realzeitsystemen anzutreffen.

Beispiel 3-5 *Deadline-Scheduling*

Im Worst Case werden alle drei Rechenprozesse des Fahrradcomputers gleichzeitig lauffähig. Gemäß der Vorgabe, dass »die Task, die als Erstes fertig sein muss, ausgeführt wird«, werden die Tasks wie in Abbildung 3-13 abgearbeitet.

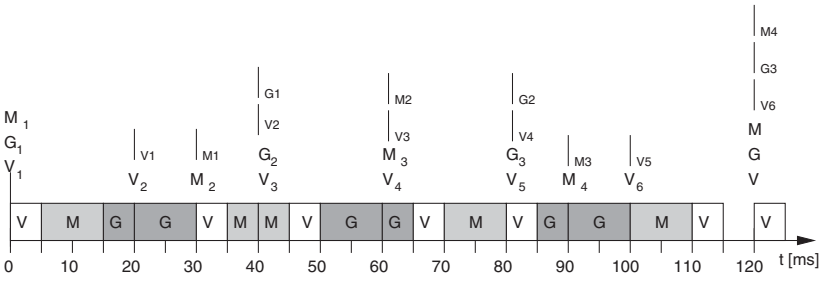


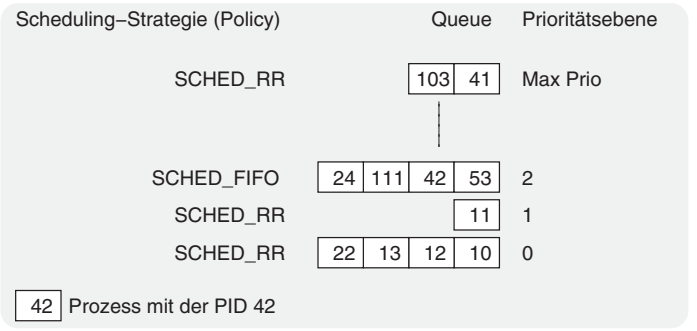
Abbildung 3-13 *Deadline-Scheduling*

Zur Erstellung der Rechnerkernbelegung werden wieder als Erstes die Rechenzeitanforderungen eingetragen. Zur besseren Orientierung werden die Anforderungen nummeriert. Außerdem werden die maximalen Deadlines (beispielsweise als senkrechte kurze Linien, versehen mit der Bezeichnung der zugehörigen Rechenzeitanforderung) eingezeichnet. Haben zwei oder mehr Tasks die gleiche maximale Deadline, ist es im Prinzip egal, welche ausgeführt wird. Typischerweise wählt der Scheduler allerdings diejenige aus, die als Letztes aktiv war, da hier die Wahrscheinlichkeit am größten ist, dass die Prozessorcaches noch mit den zugehörigen Daten gefüllt sind.

Kombinierte Scheduling-Verfahren

Nur in seltenen Fällen findet man die vorgestellten Grundvarianten des Schedulings alleinstehend implementiert. Vielmehr kombinieren moderne Betriebssysteme, wie im Folgenden beschrieben, die vorgestellten Verfahren.

Abbildung 3-14
POSIX-Scheduling



Als Basis für das Scheduling wird ein prioritätengesteuertes Scheduling, welches auf Prioritätsebenen basiert, eingesetzt: Mehrere Tasks können dieselbe Priorität haben. Innerhalb einer solchen Prioritätsebene findet das Scheduling dann nach einem FCFS oder einem Zeitscheibenverfahren statt. Beim sogenannten POSIX-Scheduling (POSIX 1003.1b) kann das innerhalb der Prioritätsebene verwendete Scheduling-Verfahren gewählt werden, ebenso wie die beim Zeitscheibenverfahren verwendete Größe der Zeitscheibe (Quantum, siehe Abbildung 3-14). Das Zeitscheibenverfahren wird bei POSIX unter dem Define `SCHED_RR` (Round Robin), das FCFS-Verfahren unter `SCHED_FIFO` geführt. Die niedrigste Priorität wird beim POSIX-Scheduling durch die 0 präsentiert.

Darüber hinaus fassen Betriebssysteme wie Linux oder Windows mehrere Prioritätsebenen zu Prioritätsbereichen zusammen. Innerhalb dieser Bereiche führt der Scheduler Verschiebungen der Prioritäten einzelner Tasks aufgrund unterschiedlicher Kriterien durch. Ein Kriterium ist beispielsweise, ob eine Task CPU-lastig (cpu-bound) oder IO-lastig (io-bound) ist. Dazu wird innerhalb der einzelnen Prioritätsebenen des Prioritätsbereiches ein Zeitscheibenverfahren eingesetzt. Wird eine Zeitscheibe nicht aufgebraucht, ist die Task IO-lastig und wird in ihrer Priorität bevorzugt. Braucht eine Task ihre Zeitscheibe auf, ist sie CPU-lastig und wird herabgestuft. Durch die daraus resultierenden kürzeren Reaktionszeiten verbessert sich die Interaktivität.

3.2.2.4 Multicore-Scheduling

Auf Multicore-Systemen gilt es, das Problem zu lösen, eine bestimme Anzahl Tasks auf einer bestimmten Anzahl CPUs so zu verteilen, dass alle Tasks die geforderten Reaktionszeiten (Deadlines) einhalten. Die Auswirkungen von Multicore- Systemen auf das Timing, insbesondere das Scheduling von Tasks in Multicore-Systemen, ist ein aktuelles Forschungsthema. Während die in Dynamisches Singlecore-Scheduling (Seite 53) vorgestellten Verfahren nachweisbar sehr gut auf Singlecore-Systemen funktionieren, so zeigen sie, insbesondere EDF, Performanceschwächen auf Multicore-Systemen. Scheduling-Verfahren auf Multicore-Systemen können in drei Gruppen unterteilt werden:

1. Partitioniertes Scheduling
2. Globales Scheduling
3. Semi-partitioniertes Scheduling

Partitioniertes Scheduling

Auf Realzeitsystemen ist in der Praxis das partitionierte Multicore-Scheduling am meisten verbreitet. Jede CPU hat ihren eigenen Scheduler. Von Vorteil ist die Offline-Verteilung der Tasks auf die zur Verfügung stehenden CPUs. Dafür werden Algorithmen zur Lösung von Optimierungsproblemen verwendet, wie z.B. der »First-Fit-Decreasing-Utilization Partitioning Algorithm«. Der Algorithmus verteilt die Tasks auf die Cores unter Beachtung einer von der jeweiligen Scheduling-Strategie auf dem Core abhängigen Auslastungsgrenze.

Eine weitere Optimierung besteht darin, möglichst alle Threads einer Threadgruppe auf einen Core zu verteilen. Falls einzelne Threads aufgrund der einzuhaltenden Auslastungsgrenze einer Threadgruppe auf mehrere CPUs verteilt werden müssen, sollten auch diese Threads möglichst zusammen auf einen weiteren Core verteilt werden. Dies optimiert die Ausführungszeit der Tasks, da Threads einer Threadgruppe auf gleiche Code- und Datenbereiche zugreifen und die Ressourcen innerhalb desselben Adressraums (Cache) benutzen.

Globales Scheduling

Bei diesem Verfahren verteilt der Scheduler die Tasks auf die Kerne dynamisch, weshalb Tasks auf unterschiedlichen Kernen ablaufen können. Eine feste Zuordnung einer Task auf einen Kern existiert nicht. Das Verfahren ist nicht auf spezifische Singlecore-Scheduling-Algorithmen beschränkt. Es eignet sich beispielsweise sowohl für prioritätenbasiertes als auch für Deadline-Scheduling. Prinzipiell nutzen globale Scheduling-Verfahren die von den CPUs angebotene Leistung (Verarbeitungszeit) besser aus als partitioniertes Scheduling. Das Verfahren migriert zur

besseren Auslastung der einzelnen CPUs die Task während ihrer Ausführungszeit sehr oft. Theoretisch ist dadurch zwar eine 100%ige Auslastung der CPUs möglich, in der Praxis verlängern die durch die Migration entstehenden Cache-Misses aber die Ausführungszeiten. Seit einigen Jahren steht globales Scheduling daher im Fokus wissenschaftlicher Arbeiten.

Semi-partitioniertes Scheduling

Hierbei handelt es sich um eine Mischform der beiden oberen Varianten. Gegenüber der reinen Partitionierung ist bei dieser Art die Partitionierung selbst Teil des Schedulers. Das bedeutet, dass der Scheduler neben der Verteilung der Tasks auf die einzelnen Prozessoren auch eine Aufspaltung von Tasks zur Laufzeit vornehmen kann (vergleiche [Bertogna2009]). Eine Task wird vom Scheduler gesplittet und die einzelnen Teile hintereinander auf verschiedenen Prozessoren in korrekter Reihenfolge ausgeführt.

Praktische Einschränkungen

Nach [Bertogna2009] ist die Partitionierung immer dann die mit hoher Wahrscheinlichkeit beste Lösung, wenn das Taskset vorab bekannt ist – was bei den meisten Realzeitsystemen der Fall ist. Auch wenn der globale Ansatz Reserven der CPU-Auslastung durch Migration der Tasks nutzt, die bei der Partitionierung unberücksichtigt bleiben, ist die Auswirkung der Migration selbst auf die Ausführungszeit der Task schwierig zu bestimmen.

Obwohl die Auslastung aller Prozessoren bei dem globalen Ansatz besser skaliert, gibt es insgesamt mehr Vorteile bei einem partitionierten Verfahren [Lopez2004] – deswegen, weil durch die Partitionierung der Realzeitnachweis mit bekannten Methoden für jede CPU einzeln durchgeführt werden kann. Die Semi-Partitionierungsalgorithmen werden ihre Vorteile erst auf Manycore-Systemen, also Systemen, die aus hundert oder auch mehr Prozessorkernen bestehen, ausspielen können [Bertogna2009]. Doch bis Manycore-Systeme auch im Embedded-Realzeit-Umfeld anzutreffen sind, werden noch einige Jahre vergehen.

Das globale Scheduling wird sehr erfolgreich in Standardbetriebssystemen eingesetzt, die Migration der Tasks stellt jedoch in Realzeitsystemen derzeit ein Problem dar. Die Modelle beziehen die Migration insgesamt zu ungenau in ihre Analysen mit ein. Daher kommt es zu sehr hohen Abweichungen der im Modell bestimmten Zeiten von den in der Realität gemessenen Zeiten.

Das Problem beim globalen Scheduling sind die Laufzeitvarianzen, die sich beim Ausführen einer Task auf verschiedenen CPUs statt nur einer CPU ergeben. Je nach CPU, auf welcher die Task eine Zeitlang läuft, kann diese auf Informationen im Cache zurückgreifen oder muss aus dem langsameren Hauptspeicher Informationen in den Cache nachladen. Dadurch verlängern sich die Ausführungszeiten im Vergleich zu Singlecore-Systemen. Da für Realzeitsysteme eine Worst-Case-Analyse durchgeführt werden muss, stellt die Bestimmung der Worst-Case-Ausführungszeit (WCET) bisher ein Problem dar, weil hierfür keine verlässlichen Verfahren für Multicore-Systeme existieren. Eine Realzeitanalyse kann derzeit nicht für globale Scheduling-Verfahren durchgeführt werden.

Globales Scheduling im Linux-Kernel

Beim Booten legt Linux zunächst ein Profil der Hardware-Architektur an. Das ist notwendig, um die jeweiligen Taskmigrationskosten und den mit einer Taskmigration verbundenen Nutzen zu bestimmen.

Insgesamt unterscheidet Linux hierzu drei unterschiedliche Mehrprozessorarchitekturen:

1. Simultaneous Multithreading (SMT)

   Diese von Intel eingeführte Architektur hat in den meisten Fällen zwei Registersätze und Pipelines, sodass sehr leicht zwischen zwei Threads umgeschaltet werden kann. Allerdings gibt es nur eine Verarbeitungseinheit. Der Performancegewinn wird allgemein mit bis zu 10 Prozent angegeben. Im strengen Sinn stellt dies allerdings kein wirkliches Multiprozessorsystem dar.

2. Symmetric Multi-Processing (SMP)

   Mehrere Prozessorkerne nutzen gemeinsam einen physikalischen Speicher. Da auf einen Speicher nicht parallel zugegriffen werden kann, müssen gleichzeitige Zugriffe der Prozessorkerne sequentialisiert werden. Das führt zu Performanceeinbußen.

3. Non-uniform Memory Architecture (NUMA)

   Einzelne Prozessoren verfügen über einen eigenen, lokalen Speicher, auf den sie exklusiv zugreifen können. Zur Kommunikation der Prozessoren untereinander gibt es zudem meistens einen globalen, gemeinsam genutzten Speicher. Je nach Speicher sind die Zugriffe unterschiedlich schnell.

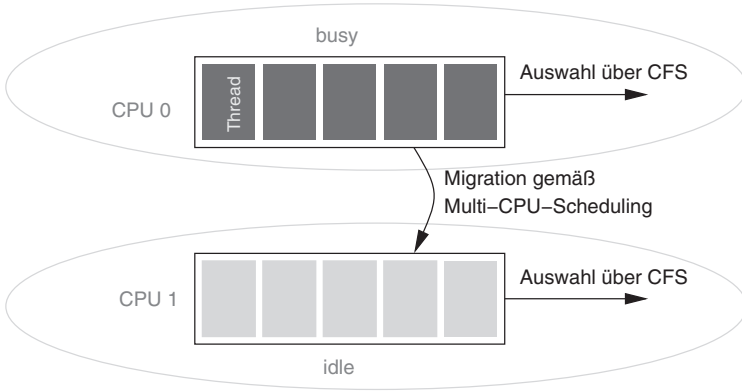


Abbildung 3-15
Der übergeordnete Mehrprozessor-Scheduler verteilt die Tasks auf die einzelnen Prozessorkerne.

Ziel des Mehrprozessor-Schedulings ist es, die Last auf allen Prozessoren im System gleichmäßig zu verteilen. Grundsätzlich ist das Scheduling zweistufig aufgebaut: Auf jedem Prozessor arbeitet der Singlecore-Scheduler. Der Mehrprozessor-Scheduler verteilt die Tasks auf die einzelnen Prozessoren, sodass der Einprozessorscheduler sich aus der Liste der ihm zugeteilten Tasks die als Nächstes von ihm zu bearbeitende heraussucht.

Der Mehrprozessor-Scheduler ist häufig als eigene Task realisiert, die im Linux-Kernel beispielsweise »Migration« heißt.

Das Verschieben eines Rechenprozesses von einer auf die andere CPU wird als Taskmigration bezeichnet.

An folgenden Stellen respektive zu den folgenden Zeitpunkten wird der Mehrprozessor-Scheduler aktiv:

1. Bei Aufruf der Systemcalls `fork()`, `clone()`, `exec()` und `exit()`. .
2. Wenn eine CPU idle wird, wenn also die Liste der zu bearbeitenden Rechenprozesse leer geworden ist.
3. Zeitgesteuert (periodisch).

Auch bei einer Ungleichverteilung der Last ist eine Taskmigration nicht unbedingt wünschenswert, da diese zunächst mit Verlusten bezahlt werden muss. Bei einem SMP-System geht beispielsweise bei einer Taskmigration der Inhalt sämtlicher Caches verloren.

Am preiswertesten ist die Taskmigration auf einem SMT-System; hier ist allerdings der Gewinn auch am geringsten. Am teuersten ist die Verschiebung auf einem NUMA-System; hier sind aber auch die Gewinne möglicherweise am größten.

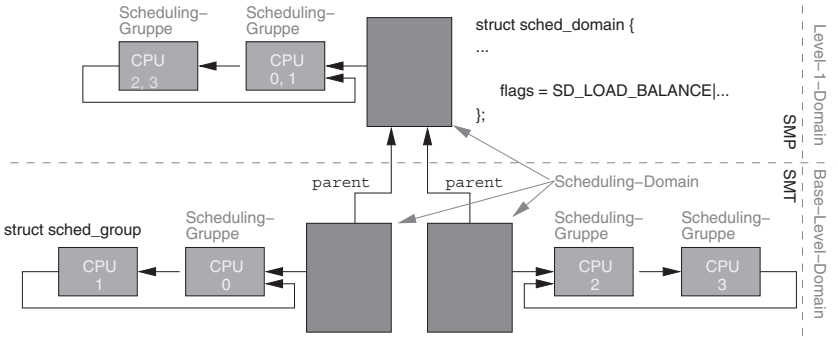


Abbildung 3-16
Die Modellierung eines SMP-Systems im Linux-Kernel

Da in der Praxis meist gemischte Hardware-Architekturen vorkommen, führte der Linux-Kernel sogenannte Scheduling-Domains und Scheduling-Gruppen ein. Eine Scheduling-Domain ist der Container für (untergeordnete) Scheduling-Gruppen. Eine Scheduling-Gruppe wiederum steht für eine CPU oder für eine andere (untergeordnete) Scheduling-Domain. Ein Zweiprozessorsystem beispielsweise modelliert Linux in einer Scheduling-Domain mit zwei Gruppen; für jede CPU eine. Ein Rechner mit Hyperthreading-Prozessor wird ebenfalls auf eine Domain und zwei Gruppen abgebildet. Ein heterogenes System dagegen besteht aus mehreren Domänen, wobei die Scheduling-Gruppen der übergeordneten Container (Domains) nicht Prozessoren, sondern eben weitere Domains repräsentieren. Ein einfaches Beispiel für eine derartig baumartige Topologie finden Sie in Abbildung 3-16. Hier ist eine Architektur mit zwei Hyperthreading-Prozessoren mithilfe dreier Scheduling-Domains abgebildet: zwei Basis-Domains und die übergeordnete Level-1-Domain. Die Level-1-Domain umspannt alle vier Prozessoren, die Basis-Domains jeweils einen Hyperthreading Prozessor mit seinen zwei logischen Kernen.

Der Mehrprozessor-Scheduler sorgt für die Lastverteilung innerhalb einer Scheduling-Domain. Die Entscheidung über eine Migration wird dabei abhängig von der Last innerhalb der Scheduling-Gruppen, den Kosten für die Migration und dem erwarteten Gewinn gefällt.

In Unix-Systemen kann über die Funktionen `sched_get_affinity()` und `sched_set_affinity()` die Affinität, also die Zugehörigkeit eines Threads zu einer CPU ausgelesen und festgelegt werden. Damit ist also die Zuordnung eines Threads auf eine spezifische CPU möglich. Durch diesen Mechanismus können Realzeitprozesse von Prozessen ohne strenge zeitliche Anforderungen separiert und damit deterministischer abgearbeitet werden (siehe Abschnitt 4.2.3).

3.2.3 Memory Management

Aufgaben der Memory Management Unit (MMU) sind

- der Speicherschutz,
- die Adressumsetzung,
- virtuellen Speicher und
- erweiterten Speicher zur Verfügung stellen.

Speicherschutz. Applikationen (Prozesse, aber nicht Threads) werden voreinander geschützt, indem jeder Prozess seinen eigenen Adressraum bekommt. Ein Zugriff ist damit nur möglich auf eigene Daten-, Stack- und Codesegmente. Daten bzw. Teile der Applikation werden vor Fehlzugriffen geschützt. Greift eine Applikation auf Speicherbereiche zu, die nicht zur Applikation gehören, oder versucht die Applikation, aus einem Codesegment Daten zu lesen, führt dies – dank MMU – zu einer Ausnahmebehandlung, die als Konsequenz ein Abbruch des Zugriffes zur Folge hat.

Adressumsetzung. Programme sollen einen einheitlichen Adressraum bekommen, um das Laden von Programmen zu beschleunigen und Shared-Libraries zu ermöglichen. War in früheren Zeiten der Loader des Betriebssystems, der Applikationen in den Speicher geladen und danach gestartet hat, dafür verantwortlich, dem Programm die richtigen (freien) Adressen zuzuweisen, kann mit MMU der Linker bereits die Adressen vergeben. Aus Sicht jeder Applikation beginnt der eigene Adressraum ab der Adresse 0. Mehrere Tasks können sich ein (Code-)Segment teilen. Durch die Trennung von Code- und Datensegmenten und mithilfe der MMU können mehrere Prozesse, die auf dem gleichen Code beruhen, ein oder mehrere Codesegmente teilen. Dadurch wird der Hauptspeicherbedarf reduziert.

Virtuellen Speicher zur Verfügung stellen. Durch die MMU kann virtueller Speicher zur Verfügung gestellt werden. Damit können Applikationen auf mehr Speicher zugreifen, als physikalisch vorhanden ist. Als Speicherersatz wird ein Hintergrundspeicher (Festplatte) verwendet, der sogenannte Swap-Space. Der vorhandene Hauptspeicher wird durch die Speicherverwaltung in Seiten (oder sogenannte Pages) eingeteilt. Wenn keine freien Pages mehr zur Verfügung stehen, werden die Inhalte belegter Seiten auf den Hintergrundspeicher ausgelagert und die freigeräumte Page kann genutzt werden. Dieser Vorgang wird Paging oder Swapping genannt. Swapping bezeichnet ursprünglich die Auslagerung kompletter Tasks, wird heutzutage jedoch häufig als Synonym für Paging verwendet. In Realzeitsystemen wird Swapping nur bedingt eingesetzt, da es zu nicht-deterministischen Systemen führt.

Zugriff auf erweiterte Speicherbereiche. Für einige Applikationen reicht der Adressraum von 16- und 32-Bit-Prozessoren mit 64 kByte

und 4 GByte nicht aus. Die Prozessorhersteller haben Techniken eingebaut, um physikalisch mehr Speicher anzuschließen (bei x86-Prozessoren heißt die Technik *Process Adress Extension*, PAE). Die Speicherverwaltung muss Techniken zur Verfügung stellen, um diesen erweiterten Speicherbereich – oft auch als Highmem bezeichnet – ansprechen zu können.

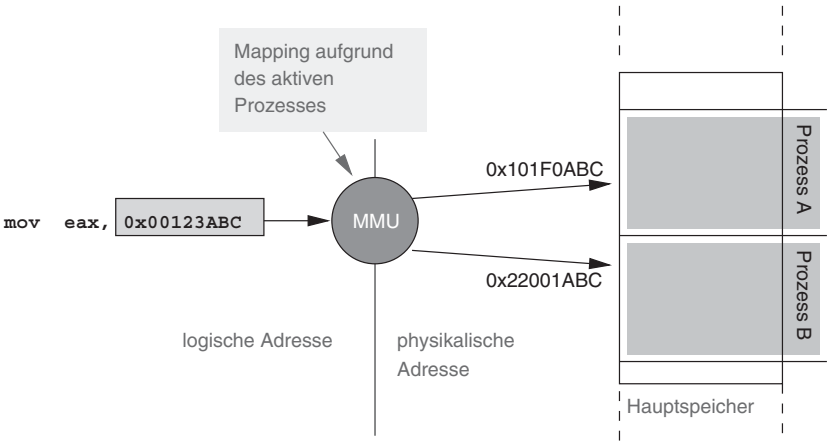


Abbildung 3-17
Adressumsetzung

MMUs bestehen aus Hardware, die durch entsprechende Software initialisiert werden muss. Heutige Mikroprozessoren haben im Regelfall eine MMU integriert. Prinzipiell funktioniert der Vorgang so, dass der Prozessorkern eine logische Adresse erzeugt. Diese logische Adresse wird durch die MMU in eine physikalische Adresse umgesetzt. Die Umsetzungsregeln selbst werden dazu in die MMU geladen. Eine MMU muss also durch das Betriebssystem initialisiert werden.

Prinzipiell kann eine MMU den physikalischen Speicher auf zwei Arten aufteilen: in Segmente und in Seiten. Während Speicherseiten feste (physikalische) Größen haben und einander auch nicht überlappen, ist die Größe der Segmente ebenso dynamisch wie ihre wirkliche Lage im Hauptspeicher.

Moderne Betriebssysteme nutzen die Segmentierung, um damit unter anderem Thread Local Storage (TLS) zu realisieren. Für Speicherschutz oder auch Swapping wird dagegen vorwiegend Paging, also die Speicherverwaltung auf Basis von Seiten, eingesetzt.

Hierbei wird der Hauptspeicher in gleich große (beispielsweise 4 KByte) Seiten bzw. Pages eingeteilt. Die vom Prozessor erzeugte logische Adresse selbst besteht aus zwei semantischen Einheiten: einem Teil, der innerhalb der MMU den sogenannten Seitendeskriptor adressiert, und einem Teil, der eine Speicherzelle innerhalb der Page auswählt.

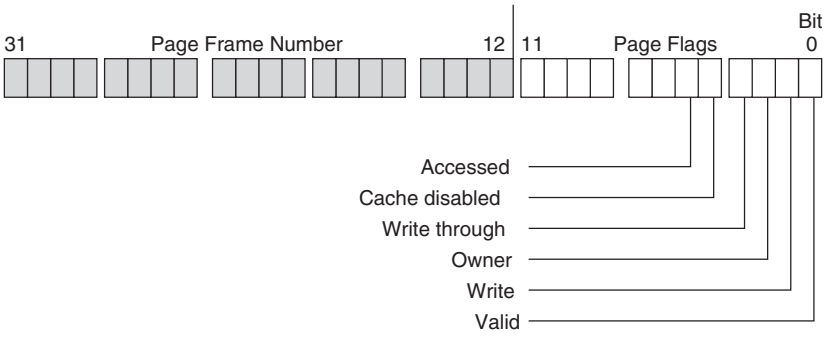


Abbildung 3-18
Seitendeskriptor beim x86

Der Seitendeskriptor wiederum enthält zum einen die physikalische Adresse der Seite (Page Frame Number) und zum anderen eine Reihe von zur Seite gehörenden Zugriffsflags. Die physikalische Adresse der Seite ergibt zusammen mit dem Offset innerhalb der Seite die vollständige physikalische Adresse.

Im Seitendeskriptor des x86-Prozessors existieren unter anderem die folgenden Zugriffsflags:

Valid (V)
Die zur Page gehörenden Daten befinden sich im (physischen) Hauptspeicher.

Write (W/R)
Dieses Flag gibt an, ob die Daten nur lesbar oder auch beschreibbar sind.

Owner (K/U)
Dieses Flag gibt an, ob die Seite von einer Applikation (Userland) oder vom Kernel verwendet wird.

Write through (T)
Dieses Bit legt das Caching-Verfahren fest.

CacheDisable (N)
Wenn dieses Bit gesetzt ist, kann die zugehörige physische Seite nicht im Cache zwischengespeichert werden.

Accessed (A)
Dieses Bit wird gesetzt, wenn die Seite oder die referenzierende Seitentabelle gelesen oder geschrieben wurde.

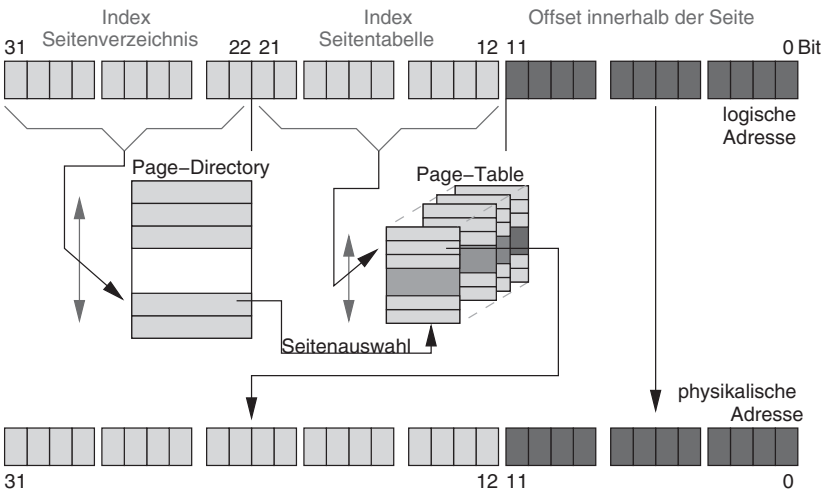


Abbildung 3-19
Zweistufiges Paging beim x86

Die auf Speicherseiten basierende MMU ist meistens im Prozessor integriert und über eigene Befehle ansprechbar. Das Paging auf einer x86-CPU ist für 32-Bit-Systeme zweistufig, für 64-Bit-Systeme dreistufig aufgebaut (Abbildung 3-19). Jede Task hat ein eigenes Page-Directory. Die obersten 10 Bit der logischen Adresse wählen einen Eintrag in diesem Page-Directory aus. Dieser Eintrag adressiert eine Page-Table. Die Bits 12 bis 21 der logischen Adresse wählen dann einen Eintrag in der Page-Table, der schließlich den Seitendeskriptor enthält. Da Page-Directory und Page-Tables im Hauptspeicher abgelegt sind, muss im Worst Case für einen Daten- oder Codezugriff dreimal auf den Hauptspeicher zugegriffen werden. In diesem Fall sind die Zugriffe sehr zeitaufwendig. Die Prozessorhersteller haben daher für die Adressumsetzung einen Cache, den sogenannten *Translation Lookaside Buffer* (TLB), eingebaut. Da die dort zwischengespeicherten Umsetzungsinformationen nur jeweils für die aktive Task gelten, ist der TLB mit jedem Taskwechsel (Context-Switch) zu leeren.

Realzeitaspekte

Die Speicherverwaltung ist für Realzeitsysteme in mehrfacher Hinsicht relevant. Um virtuellen Speicher zur Verfügung zu stellen, lagern viele Betriebssysteme Teile des Hauptspeichers auf den Hintergrundspeicher aus (Paging). Das führt für die Applikationen, die ausgelagert sind, zu hohen Latenzzeiten, denn sobald der Scheduler diese auswählt, müssen erst die Speicherbereiche vom Hintergrundspeicher in den Hauptspeicher transportiert werden. Folglich ist das Paging am besten komplett auszustellen. Alternativ muss es zumindest für die Realzeitapplikationen deaktiviert werden (Systemcall `mlock()` beziehungsweise `mlockall()`).

Erlaubt ein Betriebssystem, auch im Kernel dynamisch Speicher zu reservieren, kann es im Laufe des Betriebs zu einer Fragmentierung des Speichers kommen. In diesem Fall kann der Kernel Anfragen nach größeren, zusammenhängenden Speicherbereichen nicht mehr befriedigen. Erst nachdem andere Komponenten wieder Speicher freigegeben haben, ist das möglich. Das wiederum führt zu Verzögerungen und im ungünstigsten Fall sogar zum Absturz des Systems. Auf dynamische Speicherverwaltung ist daher möglichst generell zu verzichten beziehungsweise sämtliche Speicheranforderungen sind direkt zum Start der Realzeitapplikation durchzuführen (siehe Abschnitt 4.10).

Des Weiteren ist darauf zu achten, dass Betriebssysteme intern keine Algorithmen verwenden, die nicht deterministisch sind. In manchen Fällen hängt die Laufzeit von der Betriebsdauer, der Anzahl der Tasks oder Ähnlichem ab.

3.2.4 I/O-Management

Das Ein-/Ausgabe-Management erfüllt in einem Kernel die folgenden Aufgaben:

- Einheitliches Application Programming Interface (API) zur Verfügung stellen.
- Systemkonforme Integration von Hardware ermöglichen (Treiberinterface).
- Strukturierte Ablage von Informationen in Dateien und Verzeichnissen (Filesysteme) ermöglichen.

3.2.4.1 Programmierinterface (API)

Das Betriebssystem stellt auf Applikationsebene ein Systemcall-Interface zur Verfügung, um auf Geräte zuzugreifen. Dazu gehören im Wesentlichen Funktionen, um Geräte zu initialisieren, Daten zu den Geräten zu transferieren und Daten von den Geräten zu lesen. Die Applikationsschnittstelle zum I/O-Subsystem ist so gestaltet, dass sie für fast jede Art von Geräten nutzbar ist.

Schnittstellenfunktionen

Ähnlich wie in der Hardware werden für die Applikation Zugriffe auf Peripherie auf ein einfaches Lesen und Schreiben (read und write) abgebildet. Bevor jedoch auf ein Gerät geschrieben bzw. von einem Gerät gelesen werden kann, muss dazu die Ressource beim Betriebssystem angefordert werden. Bei der Anforderung entscheidet das System, ob die Task den Zugriff bekommt oder ob der Zugriff abgelehnt wird. Gründe für eine Ablehnung können sein:

- fehlende Zugriffsrechte und
- die Ressource ist bereits belegt.

Beispiel 3-6
Geräteschnittstelle

```
#define ZUMACHEN  0x00
#define AUFMACHEN 0x01

...
char Tuere;

...
fd=open( ``Aufzugtuer'', O_RDWR );
if (fd<0) {
    ...    /* Fehlerbehandlung */
}
Tuere = AUFMACHEN;
count=write( fd, &Tuere, sizeof(Tuere) ); // Tuere auf
if (count<sizeof(Tuere)) {
    ...    /* Fehlerbehandlung */
}
...
close( fd );
```

Die *klassischen* Zugriffsfunktionen (ANSI-C) für den Zugriff auf Geräte sind die gleichen wie für den Zugriff auf Dateien. Geräte werden also an dieser Schnittstelle wie Dateien gehandhabt!

`open`

Mit `open` wird der spätere Zugriffswunsch auf ein Gerät beim Betriebssystem angemeldet. Das Gerät selbst wird in symbolischer Form spezifiziert. Außerdem muss angegeben werden, in welcher Form (lesend/schreibend) auf das Gerät zugegriffen werden soll und in welcher Art (z.B. blockierend oder nicht blockierend). `open` liefert als Ergebnis einen sogenannten *Descriptor* zurück, wenn das Betriebssystem die Ressource der Task zuteilt. Dieser Descriptor wird von den folgenden Diensten als Kennung verwendet, sodass die produktiven Dienste (read/write) effizient bearbeitet werden können (Beispiel 3-6).

`close`

Dieser Systemcall gibt die angeforderte Ressource wieder frei.

`read`

Zum lesenden Zugriff wird der Systemcall `read` verwendet.

`write`

Mit dieser Funktion können Daten an die Peripherie übermittelt werden.

`ioctl`

Lässt sich eine bestimmte Funktionalität, die ein Peripheriegerät besitzt, nicht auf Schreib-/Leseaufrufe abbilden, steht der IO-Control-Aufruf zur Verfügung. Dieser Aufruf ist gerätespezifisch. Über IO-Controls werden beispielsweise die verschiedenen Betriebsparameter und Betriebsarten einer seriellen Schnittstelle konfiguriert oder auch ein atomares Lesen und Schreiben ermöglicht.

Zugriffsarten

Da man bei Realzeitbetriebssystemen darauf achten muss, dass das Warten auf Ereignisse das System selbst nicht blockiert, lassen sich Aufrufe, die zum Warten führen können, unterschiedlich parametrieren. Insgesamt lassen sich drei Aufrufarten unterscheiden:

1. Aufruf mit implizitem Warten (blockierender Aufruf)
2. nicht blockierender Aufruf
3. Aufruf mit explizitem Warten

Normalerweise wird die Task, die auf ein Gerät zugreift, so lange schlafen gelegt, bis das Gerät geantwortet hat oder eine Fehlersituation (das Gerät antwortet innerhalb einer definierten Zeitspanne nicht) eingetreten ist. Man spricht von einem *Warteaufruf mit implizitem Warten* oder auch von einem synchronen Zugriff.

Demgegenüber spricht man von einem *Warteaufruf mit explizitem Warten*, wenn der Warteaufruf (Zugriff auf das Gerät) direkt zurückkehrt. Die Task, die den Aufruf durchgeführt hat, kann – unabhängig von der Bearbeitung des Warteaufrufs innerhalb des Betriebssystems – weiterrechnen. Ist der Zugriff auf das Gerät durch das I/O-Subsystem abgeschlossen, wird die Task darüber entweder per Ereignis (auf das an anderer Stelle entweder explizit gewartet, oder welches durch die Task gepollt wird) informiert, oder das Betriebssystem ruft eine vorher übergebene Callback-Funktion auf. Bei Warteaufrufen mit explizitem Warten spricht man auch von einem asynchronen Zugriff.

Aus Sicht des Betriebssystems sind Warteaufrufe mit explizitem Warten nicht trivial zu realisieren. Das liegt an der Schwierigkeit, dass von der Applikation mehrere explizite Warteaufrufe gestartet werden können. Das Betriebssystem muss sich die Informationen innerhalb des Betriebssystems aber merken. Darüber hinaus muss das I/O-Subsystem darauf eingerichtet sein, dass zwischen Start des Zugriffs und Ende des Zugriffs die zugehörige Applikation beendet wird.

Bei der dritten Variante, dem nicht blockierenden Zugriff, kehrt der Funktionsaufruf direkt zurück, auch wenn die angeforderten Daten nicht zur Verfügung stehen. Der zugreifende Thread wird also nicht – wie im Fall des blockierenden Zugriffs – schlafen gelegt, sondern kann direkt weiterarbeiten. Der blockierende oder nicht blockierende Zugriff wird bei Unix-Betriebssystemvarianten entweder beim *Öffnen*/Initialisieren (Geräte werden über die Funktion `open` geöffnet bzw. initialisiert) eingestellt oder nachträglich über die Funktion `fcntl` (siehe Beispiel 4-28).

Das I/O-System vermeidet Schlafen darüber hinaus an der Applikationsschnittstelle mit dem sogenannten `select`- respektive `poll`-Aufruf (Beispiel 4-29). Diese Funktionen überprüfen gleich mehrere Ein- beziehungsweise Ausgabequellen (zum Beispiel Peripheriegeräte oder auch Netzwerkverbindungen) darauf, ob von diesen ohne zu schlafen Daten gelesen oder auf diese Daten geschrieben werden können. Da die Funktionen gleich mehrere Ein-/Ausgabequellen überprüfen, können sie übrigens als Multiplexer eingesetzt werden, die bei entsprechender Konfiguration die zugehörige Task auch schlafen legen können.

3.2.4.2 Treiberinterface

Um an der Applikationsschnittstelle ein einheitliches Interface anbieten zu können, muss ein Peripheriemodul gemäß bestimmter Konventionen (Schnittstellen) in den Betriebssystemkern eingebunden werden. Dazu existiert innerhalb des Betriebssystemkerns die Gerätetreiberschnittstelle. Mithilfe dieser Schnittstelle erstellt der Informatiker oder Ingenieur einen sogenannten Gerätetreiber (Device-Driver), der – oftmals dynamisch während der Laufzeit – in den Betriebssystemkern integriert wird. Verwendet dann eine Applikation eine der Zugriffsfunktionen `open()`, `close()`, `read()`, `write()` oder `ioctl()`, wird im Gerätetreiber eine entsprechende Funktion aufgerufen, die das Peripheriemodul so ansteuert, dass dieses die spezifizierte Funktionalität erbringt.

Ein Gerätetreiber ist also:

- ein Satz von Funktionen, die den Zugriff auf eine spezifische Hardware (Gerät) steuern.
- die gerätespezifische Implementierung der allgemeinen Schnittstellenfunktionen.

Die Aufgabe des Betriebssystems (I/O-Management) im Kontext der Gerätetreiber ist:

- eine eindeutige Schnittstelle zur Treiberintegration zur Verfügung zu stellen,
- die Verwaltung der Ressourcen (Interrupts, Ports etc.),
- Mechanismen für die Realisierung von Zugriffsarten und Zugriffsschutz bereitzustellen,
- die Zuordnung der allgemeinen Schnittstellenfunktionen (Systemcalls) zu den gerätespezifischen Funktionen (zum Beispiel `driver_open()`) durchzuführen.

Auch heute findet man noch sehr häufig Ingenieure, die Peripheriemodule nicht über einen Treiber, sondern direkt aus der Applikation heraus ansteuern. Jedoch ist ein Treiber notwendig:

- Um benötigte Ressourcen vom Betriebssystem zugeteilt zu bekommen. Schließlich ist das Betriebssystem für die Verwaltung der Ressourcen zuständig. Wird ein Gerät nicht systemkonform eingebunden, kann das Betriebssystem die zugesicherten Eigenschaften nicht mehr garantieren. Es hat beispielsweise keinerlei Kontrolle darüber, ob eine Ressource (ein Interrupt, ein I/O-Bereich oder ein sonstiger Speicherbereich) bereits vergeben ist oder nicht.
- Um systemkritische Teile zu kapseln. Zugriffe auf Hardware sind sicherheitskritische Aktionen, die nur innerhalb eines Treibers durchgeführt werden dürfen. Wird aber beispielsweise die Hardware aus der Applikation heraus angesteuert (indem die Register bzw. Speicherbereiche der Hardware in den Adressraum der Applikation eingeblendet werden), muss die Applikation erweiterte Zugriffsrechte erhalten. Damit wiederum gefährdet sie die Sicherheit des gesamten Systems. Programmierfehler können nicht nur zum Absturz der Task, sondern sogar zum Absturz des gesamten Systems führen.
- Um bei Applikationsfehlern das Gerät in den sicheren Zustand überführen zu können. Durch die eindeutige Trennung zwischen Applikation und Gerätetreiber ist es für das Betriebssystem möglich, bei Applikationsfehlern das Gerät in den sicheren Zustand zu überführen. Stürzt die Applikation durch einen Fehler ab, erkennt dies das Betriebssystem. Da es außerdem weiß, welche Applikation auf das Gerät zugegriffen hat, kann es die im Gerätetreiber befindliche Funktionalität aktivieren, um einen sicheren Gerätezustand herzustellen. Insbesondere dieser Grund ist bei sicherheitskritischen Realzeitsystemen entscheidend!

Die meisten Betriebssysteme haben das durch Unix eingeführte Konzept übernommen, Geräte auf Applikationsebene auf Dateien abzubilden und mit den bereits vorgestellten Systemcalls anzusprechen. Verwendet also eine Applikation den `open()`-Aufruf, um damit den Zugriff auf ein Gerät zu erhalten, ruft der Betriebssystemkern eine gerätespezifische `driver_open()`-Funktion im Treiber auf. Jeder Treiber stellt eine solche Funktion zur Verfügung, weshalb es im Kernel so viele `driver_open()`-Funktionen wie Treiber gibt, während an der Applikationsschnittstelle genau eine `open()`-Funktion existiert.

Die Verbindung zwischen dem `open()`-Aufruf in der Applikation und dem gewünschten Treiber wird über eine Gerätedatei und eine Gerätenummer realisiert. Im Unterschied zu einer normalen Datei sind bei einer Gerätedatei keine les- oder schreibbaren Daten hinterlegt, sondern diese ist mit einer eindeutigen Gerätenummer verknüpft. Die Gerätenummer wiederum repräsentiert den gewünschten Treiber. Um beispielsweise einen über USB angeschlossenen Drucker anzusprechen, muss die Datei `/dev/usb/lp0` geöffnet werden. Wie die folgende Ausgabe zeigt, hat diese die Gerätenummer 180; folglich hat sich der zugehörige Treiber unter der Gerätenummer 180 beim Kernel angemeldet:

```
user@host:~< ls -l /dev/usb/lp0
crw-rw---- 1 root lp 180, 0 2012-05-08 06:54 /dev/usb/lp0
```

Abbildung 3-20 zeigt die Verknüpfung zwischen Applikation und Treiber noch einmal anhand eines Linux-Gerätetreibers. Wird der Treiber per `insmod` geladen, so wird beim Laden die Funktion `init_module` aufgerufen. Diese Funktion *registriert* den Treiber, wobei die Gerätenummer (Major-Number) und die Tabelle mit den Treiberfunktionen dem Kernel übergeben werden. In der Abbildung wird die einfacher zu handhabende, aber nicht mehr ganz aktuelle Variante über die Funktion `register_chrdev(MY_MAJOR_NUMBER, ..., &mydevice_table)` gezeigt. Das in der Abbildung dargestellte manuelle Anlegen der Gerätedatei über das Kommando `mknod` kommt heute im Wesentlichen in eingebetteten Systemen vor. Moderne Linux-Systeme nutzen hierfür einen Automatismus.

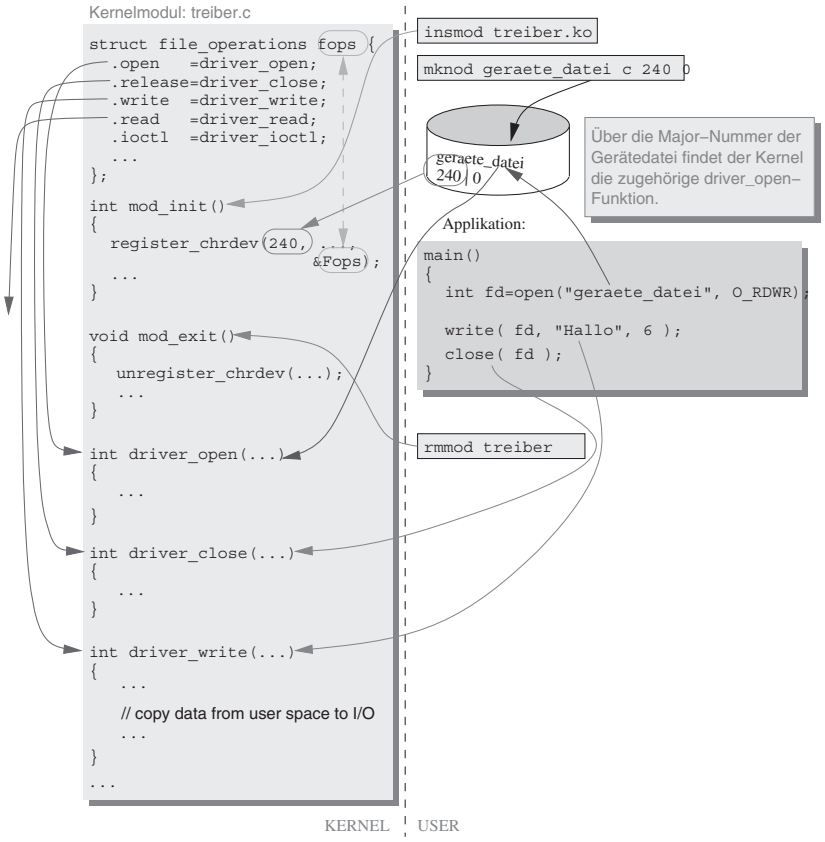


Abbildung 3-20 *Einbindung des Gerätetreibers in das System*

Bei der Treibererstellung sind damit im Wesentlichen diese internen Funktionen zu kodieren:

init_module. Die Funktion, die beim Laden des Treibers aufgerufen wird, meldet den Treiber beim Betriebssystem an.

cleanup_module. Diese Funktion meldet den Treiber wieder beim Betriebssystem ab.

driver_open. Diese Funktion überprüft, ob die anfragende Applikation Zugriff auf das Gerät bekommen soll oder nicht. Eventuell werden Hardware-Ressourcen reserviert.

driver_close. Sollten durch `driver_open()` Ressourcen reserviert worden sein, werden diese hier wieder freigegeben. Wichtig ist aber vor allem, dass das Gerät wieder in einen definierten und sicheren Zustand überführt wird.

driver_read. Aufgabe der Funktion ist es, die Daten von der Hardware auszulesen und in den von der Applikation zur Verfügung gestellten Speicherbereich zu schreiben. Dabei sind die Zugriffsarten (blockierend, nicht blockierend) zu berücksichtigen.

driver_write. Aufgabe der Funktion ist es, Daten vom Userland in den Kernel zu kopieren und von hier aus in die Hardware zu schreiben. Dabei sind die Zugriffsarten (blockierend, nicht blockierend) zu berücksichtigen.

Detaillierte Informationen zur Erstellung von Gerätetreibern unter Linux finden sich in [QuKu2011].

3.2.4.3 Filesysteme

Filesysteme dienen zum Abspeichern bzw. Sichern von:

- Programmen (Betriebssystemkern, Dienstprogrammen und Applikationen)
- Konfigurationsinformationen
- Daten (z.B. HTML-Seiten)

auf sogenanntem Hintergrundspeicher.

Werden in der Desktop- und Serverwelt klassischerweise Festplatten als Hintergrundspeicher verwendet, benutzt man im Bereich eingebetteter Systeme EEPROMs, Flash-Speicher oder Solid State Disks (SSD), also Festplatten, die auf Flash-Speicher aufbauen.

Damit auf einen Hintergrundspeicher mehrere Dateien/Daten abgelegt werden können, ist eine Organisationsstruktur (Filesystem, z.B. FAT, VFAT, NTFS, Linux Ext4 Filesystem) notwendig. Diese Organisationsstruktur sollte:

- einen schnellen Zugriff ermöglichen und
- wenig Overhead (bezüglich Speicherplatz) für die Verwaltungsinformation benötigen.

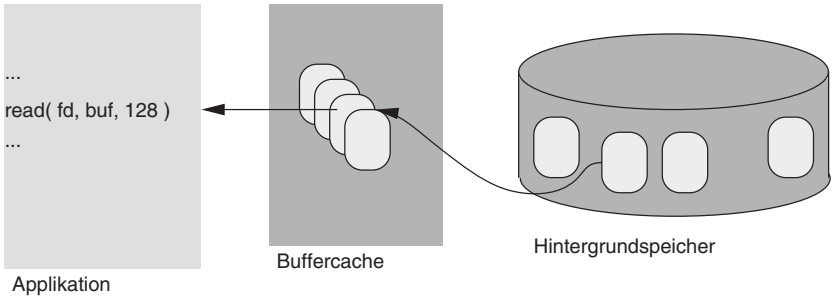


Abbildung 3-21
Lesen über den Buffercache

Um einen schnellen Zugriff zu ermöglichen, arbeiten einige Systeme mit einem dazwischengeschalteten Cache, dem sogenannten Buffer- oder Pagecache. Möchte eine Applikation eine Datei, die auf dem Filesystem abgelegt ist, lesen, schaut der Betriebssystemkern im Buffercache nach, ob die gewünschte Information dort vorhanden ist oder nicht. Ist dies

nicht der Fall, wird die Information in den Cache geladen und dann weiter an die Applikation gereicht. Wird ein Schreibauftrag erteilt, gehen die Daten zunächst an den Buffercache. Erst einige Zeit später werden die Daten auf dem Hintergrundspeicher mit den Daten aus dem Cache abgeglichen.

Dieses Verfahren ist für Realzeitsysteme oftmals nicht tolerabel:

- Das Filesystem kann dann Inkonsistenzen enthalten, wenn das System abstürzt, der Hintergrundspeicher aber nicht mit dem Cache rechtzeitig abgeglichen worden ist. Hiergegen helfen sogenannte Journaling Filesysteme, die Transaktionen mitprotokollieren.
- Zugriffe auf das Filesystem sind nicht deterministisch, da nicht vorhergesagt werden kann, ob angeforderte Daten im Cache gefunden werden oder zu einem Leseauftrag an den Hintergrundspeicher führen.

Um dennoch das Verfahren im Umfeld von Realzeitsystemen einsetzen zu können, wird der `sync`-Modus verwendet. Beim `sync`-Modus werden Dateien direkt (also unter Umgehung des Buffercache) geschrieben, ein Lesezugriff kann aber über den deutlich schnelleren Buffercache stattfinden.

Ebenfalls zur Performance-Steigerung wird der IO-Scheduler im Betriebssystemkern eingesetzt. Anstatt Ein-/Ausgabeaufträge direkt an den Hintergrundspeicher weiterzureichen, speichert der IO-Scheduler die Aufträge eine kurze Zeit lang zwischen, um sie danach sortiert abzuarbeiten. Das dient der Reduktion von Zugriffszeiten, die sich durch den mechanischen Aufbau der Festplatten ergeben. Der IO-Scheduler reduziert dazu die Kopfbewegungen. Ähnlich wie beim Task-Scheduler gibt es auch beim IO-Scheduler unterschiedliche Algorithmen. Der einfachste Algorithmus ist der *Elevator*. Bei diesem werden die Aufträge der Reihe nach sortiert, sodass der Lesekopf immer nur Bewegungen nach vorne machen muss, bevor er dann auf den Plattenanfang zurückgesetzt wird.

Da Solid State Disks (SSD) keine mechanischen Elemente besitzen, sind die zurzeit eingesetzten IO-Scheduler für diese ungeeignet. Daher schaltet der Systemarchitekt das IO-Scheduling für SSDs häufig aus (unter Linux durch Auswahl des Algorithmus noop).

3.2.5 Timekeeping (Zeitverwaltung)

Zeiten spielen in einem Realzeitsystem naturgemäß eine wesentliche Rolle. Ein Zeitgefühl wird für unterschiedliche Aufgaben benötigt:

- Zeitmessung
  Das Messen von Zeiten ist eine oft vorkommende Aufgabe in der Automatisierungstechnik. Geschwindigkeiten lassen sich beispielsweise über eine Differenzzeitmessung berechnen.
- Zeitsteuerung für Dienste
  Spezifische Aufgaben in einem System müssen in regelmäßigen Abständen durchgeführt werden. Zu diesen Aufgaben gehören Backups ebenso wie Aufgaben, die der User dem System überträgt (z.B. das Erfassen von Messwerten zu bestimmten Zeitpunkten).
- Watchdog (Zeitüberwachung)
  Neben der Zeitmessung spielt die Zeitüberwachung eine sicherheitsrelevante Rolle bei Realzeitsystemen. Zum einen werden einfache Dienste (z.B. die Ausgabe von Daten an eine Prozessperipherie) zeitüberwacht, zum anderen aber auch das gesamte System (Watchdog). Dazu muss das System in regelmäßigem Abstand einen Rückwärtszähler (Timer) auf seinen Initialwert setzen, damit der Zähler es nicht schafft, bis null herunterzuzählen. Ist das System in einem undefinierten Zustand, fällt das Setzen auf den Initialwert aus. Der Rückwärtszähler erreicht die Null, was entweder direkt einen Reset am Prozessor auslöst oder einen Interrupt, dessen Interrupt-Service-Routine das System wieder in einen definierten Zustand bringt.

3.2.5.1 Realisierungsformen

Das für Betriebssystem und Anwendung benötigte Zeitgefühl wird über Hardware- oder Software-Zähler realisiert, die periodisch getaktet inkrementieren oder dekrementieren. Die zeitliche Auflösung der Zähler ergibt sich aus der Taktfrequenz, die Genauigkeit aus der Stabilität des Taktgebers. Grundsätzlich gilt: Je höher die Taktfrequenz, desto genauer die Zeitmessung. Die Bitbreite des Zählers entscheidet über den (Zeit-)Messbereich.

Die Software-Zähler werden typischerweise im Rahmen einer Interrupt-Service-Routine inkrementiert beziehungsweise dekrementiert. Klassischerweise wird dafür ein periodischer Interrupt eingesetzt (Timer-Interrupt), der beispielsweise alle 10 ms, also mit einer Rate von 100 Hz auftritt. Innerhalb der durch den Interrupt aktivierten Interrupt-Service-Routine wird nicht nur der Zähler inkrementiert, sondern auch überprüft, ob Tasks in den Zustand lauffähig zu versetzen sind, die auf das Ablaufen einer Zeit geschlafen haben. Daraufhin läuft der Scheduling-Algorithmus und anschließend der Context-Switch.

Gemäß dieser Architektur sind Zeitaufträge mit einer Auflösung von maximal dem Timertick (Zeitabstand zwischen zwei Timer-Interrupts) möglich. Möchte beispielsweise eine Task für wenige Mikrose-

kunden schlafen, wird sie trotzdem erst mit dem nächsten Timertick geweckt. Im ungünstigsten Fall ist das bei 100 Hz nach knapp 10 ms. Um die Genauigkeit von Zeitoperationen zu verbessern, muss die Rate der Timer-Interrupts erhöht werden. Das geht allerdings zulasten der Effizienz, was sich insbesondere bei leistungsschwachen Systemen bemerkbar machen kann.

Moderne Realzeitsysteme sind *tickless*. Tickless bedeutet jedoch mitnichten, dass es gar keinen Timer-Interrupt gibt, sondern nur, dass der Timer-Interrupt nicht periodisch, sondern bedarfsgemäß auftritt. Dazu wird mit jedem Timer-Interrupt der nächste Zeitpunkt berechnet, an dem der Interrupt erneut ausgelöst werden soll. Entsprechend wird der Interrupt-Controller programmiert. Das hat gleich mehrere Vorteile:

- Die Effizienz wird gesteigert.
- Energie wird eingespart, da während der inaktiven Zeiten das System schlafen gelegt werden kann.
- Die Genauigkeit von Zeitaufträgen wird erhöht, da nicht mehr auf den nächsten Interrupt gewartet werden muss. Vielmehr wird der Interrupt exakt zum benötigten Zeitpunkt ausgelöst.

Nachteilig ist, dass mit jedem Interrupt eine Berechnung des nächsten Auslösezeitpunkts und die Umprogrammierung des Timer-Bausteins erfolgen muss. Außerdem müssen die Software-Timer entsprechend der real vergangenen Zeit aktualisiert werden.

3.2.5.2 Zeitgeber

Zwei Ausprägungen von Zeitgebern lassen sich unterscheiden:

Absolutzeitgeber. Zeiten werden grundsätzlich relativ zu einem Ereignis (beispielsweise die Geburt Christi oder die sogenannte Unix-Zeit, der 1.1.1970) angegeben. Falls dieses Ereignis aus Sicht eines Rechnersystems außerhalb und vor allem vor der Aktivierung des Rechnersystems liegt, spricht man von einer absoluten Zeit (Uhren).

Relativzeitgeber. Steht das Ereignis in direkter Beziehung zum Rechnersystem, beispielsweise das Ereignis *Start des Rechners*, spricht man von einer relativen Zeit. Der zugehörige Zeitgeber ist ein Relativzeitgeber. Die Relativzeitgeber gibt es als Vorwärts- und als Rückwärtszähler (Timer).

Bieten Prozessoren einen sogenannten Time-Stamp-Counter (TSC) an, einen Zähler, der mit der Taktfrequenz der CPU getaktet wird, nutzt das Betriebssystem diesen häufig, um damit das Timekeeping auf eine sehr genaue Basis (Nanosekunden) zu stellen. Bei einem Interrupt wird die real vergangene Zeit durch Auslesen des TSC bestimmt und die internen (Software-)Zeitzähler werden entsprechend angepasst. Dabei gibt es allerdings das Problem der variablen Taktfrequenzen. Um Energie zu

sparen, werden moderne Prozessoren mit möglichst niedriger Frequenz getaktet. Erst wenn mehr Leistung benötigt wird, wird auch die Taktfrequenz erhöht. Der Taktfrequenzwechsel muss vom Timekeeping natürlich berücksichtigt werden.

Da Applikationen im Userland ebenfalls Zugang zum Timestamp-Counter haben, müssen sich diese über Taktfrequenzänderungen informieren lassen und einen Wechsel entsprechend berücksichtigen.

Beim Umgang mit Zeiten gibt es im Wesentlichen zwei Problemfelder:

- Zählerüberläufe
  Die verwendeten Hard- und Software-Zähler haben eine endliche Breite und können damit abhängig von der gewählten Auflösung auch nur einen endlichen Zeitraum messen. Danach gibt es einen Zählerüberlauf. Der Zählerüberlauf ist dann kritisch, wenn Komponenten Zeitvergleiche durchführen. Bei einem Zählerüberlauf ist der jüngere Zeitstempel kleiner als der ältere und führt damit zu einem falschen Vergleichsergebnis, infolgedessen es zu einem unvorhersehbaren Verhalten kommen kann. Der Linux-Kernel in der Version 2.4 bspw. hat mit seinem 32-Bit-Zähler und einer Timer-Rate von 100 Hz einen Zählerüberlauf nach knapp 497 Tagen. Die Zähler, die die sogenannte Unix-Zeit repräsentieren, laufen am 19. Januar 2038 um 03:14:08 über.
  Zur Lösung des Problems sind zum einen ausreichend große Zähler zu verwenden. Linux bspw. setzt ab Kernel 2.6 auf 64 Bit breite Zähler. Das reicht aus, um in Milliarden von Jahren keinen Überlauf zu befürchten. Zum anderen sind Zeitvergleiche mit Vorsicht zu programmieren. Falls die zu vergleichenden Zeitstempel nicht weiter auseinanderliegen als die Hälfte des Messbereiches, können die in Abschnitt 4.4.2 vorgestellten Makros eingesetzt werden.
- Zeitsynchronisation
  Rechner werden nur noch selten als vollständig autarke Systeme eingesetzt. Vielmehr sind sie Teil eines Gesamtsystems, bei dem mehrere Komponenten Informationen austauschen. Hierbei spielt sehr häufig die Zeit eine wichtige Rolle, sodass die Systeme zeitlich synchronisiert werden. Ist dabei eine Zeitkorrektur notwendig, darf diese niemals sprunghaft vorgenommen werden! Wird die Zeit sprunghaft vorgestellt (z. B. bei der Umstellung von Winterzeit auf Sommerzeit), kommen manche Zeitpunkte nicht vor (bei der genannten Zeitumstellung bspw. die Zeit 2:30 Uhr). Steht zu einem ausgelassenen Zeitpunkt aber ein Zeitauftrag (Backup) an, wird dieser nicht durchgeführt und geht verloren. Wird die Zeit sprunghaft zurückgestellt, kommen demgegenüber Zeitpunkte gleich zweifach vor. Zeitaufträge werden dabei u. U. doppelt ausgeführt.

3.2.5.3 Zeitsynchronisation

Für eine funktionierende Zeitsynchronisation ist es zunächst notwendig, dass die Systemzeit auf der Zeitzone UTC (Universal Time Zone) basiert. Damit wird das Rechnersystem unabhängig vom Ort und unabhängig von Sommer- und Winterzeiten. Die Anwendungssoftware kann auf Basis der Systemzeit und des aktuellen Rechnerstandortes die lokale Zeit ableiten. Des Weiteren darf niemals eine sprunghafte Zeitkorrektur vorgenommen werden. Die Zeit muss vielmehr der Zielzeit angepasst werden. Dazu wird die Zeitbasis (das Auftreten des periodischen Interrupts) gefühlvoll verlangsamt oder beschleunigt. Gefühlvoll bedeutet in diesem Zusammenhang, dass bei großen zeitlichen Unterschieden zunächst eine stärkere Korrektur erfolgt. Je näher man der Zielzeit ist, desto behutsamer erfolgt die Korrektur. Das Network Time Protocol (NTP) ist übrigens für diese Art der Zeitsynchronisation entwickelt worden. Der Systemcall `adjtime` parametriert die zeitliche Anpassung im Kernel.

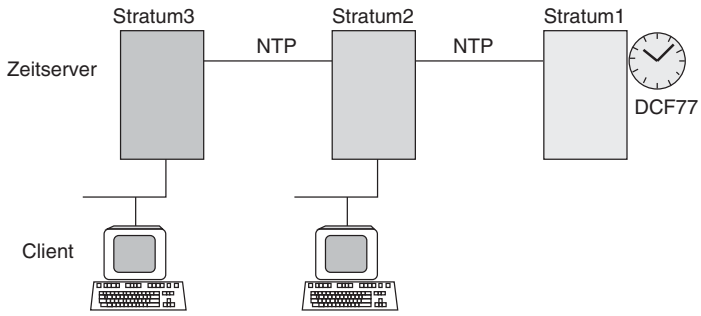


Abbildung 3-22 *Zeitsynchronisation per NTP*

Beim Network Time Protocol stellen Zeitserver die aktuelle Zeit zur Verfügung, wobei die Zeitserver klassifiziert werden. Der Zeitserver, der die Uhrzeit selbst bestimmt (z.B. durch eine DCF77-Funkuhr), ist ein sogenannter Stratum-1-Server. Der Server, der die Uhrzeit von einem Stratum-1-Server bezieht, ist der Stratum-2-Server usw. Bei der Verteilung der Uhrzeit werden Berechnungen über die Laufzeit der Pakete zwischen den Rechnern durchgeführt und die Uhrzeit wird entsprechend korrigiert.

Precision Time Protocol IEEE 1588

Das Precision Time Protocol (PTP) kann zur Synchronisation von Zeitgebern in einem lokalen Netzwerk (LAN) mit einer Genauigkeit im Mikrosekundenbereich und auch darunter (Sub-Mikrosekundenbereich) eingesetzt werden. Es ist als IEEE 1588/2002 und IEEE 1588/2008 normiert. Auch wenn das Protokoll kein Übertragungsmedium zwischen

den einzelnen Stationen festlegt, basiert es zumeist auf Ethernet. Das Protokoll kann in Software, aber auch der höheren Genauigkeit wegen in Hardware abgewickelt werden.

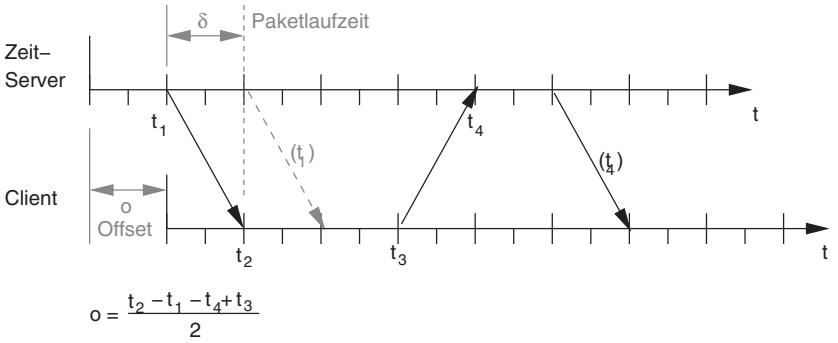


Abbildung 3-23
Zeitsynchronisation per Precision Time Protocol (PTP)

Das Protokoll ist in seiner Basisfunktionalität relativ einfach aufgebaut. Eine Station im LAN ist Zeitserver, die anderen sind die Slaves. Zeitserver sollte immer die Station sein, deren Uhr am genauesten ist.

Abbildung 3-23 zeigt die Abläufe bei der Zeitsynchronisation. Der Master schickt dem Client eine Nachricht, die die genaue Zeit t_1 enthält, zu der die Nachricht selbst verschickt wird (Absendezeitpunkt). Ist es aus technischen Gründen nicht möglich, den Sendezeitpunkt im Paket selbst mitzuschicken - was von der Hardware unterstützt werden muss – schickt der Server ein sogenanntes Follow-up-Paket, welches dann den korrekten Sendezeitpunkt des ersten Paketes enthält. Der Client empfängt die erste Nachricht und hält den Empfangszeitpunkt t_2 fest.

Die Zeitdifferenz zwischen t_2 und t_1 enthält zum einen den Zeitunterschied (Offset) zwischen den Uhren der beiden Rechner und zusätzlich noch die Laufzeit der Datenübertragung. Um Letztere herausrechnen zu können, schickt jetzt seinerseits der Client dem Zeitserver ein Paket, den sogenannten Delay-Request. Den Absendezeitpunkt t_3 speichert der Client ab. Der Zeitserver packt den Empfangszeitpunkt t_4 dieses Paketes in ein Antwortpaket, das er dem Client zurückschickt.

Die Zeitdifferenz zwischen den beiden Zeitstempeln t_4 und t_3 ergibt sich wiederum aus der Paketlaufzeit und dem eigentlich gesuchten Offset zwischen den beiden Uhren. Da allerdings die Übertragungsrichtung des Paketes umgekehrt ist, geht der Offset jetzt mit negativem Vorzeichen ein. Die folgende Rechnung zeigt, dass sich die Paketlaufzeiten gegenseitig aufheben und wie der Client aus den vier Zeitpunkten den Offset zwischen den Uhren berechnet:

$$
\begin{aligned}
t_2 - t_1 &= o + \delta \\
t_4 - t_3 &= -o + \delta \\
\delta &= t_2 - t_1 - o \\
\delta &= t_4 - t_3 + o \\
t_2 - t_1 - o &= t_4 - t_3 + o \\
o &= \tfrac{1}{2}(t_2 - t_1 - t_4 + t_3)
\end{aligned}
$$

Ein Client erfasst die folgenden vier Zeitstempel: t_1 = 1000 us, t_2 = 1060 us, t_3 = 1400 us und t_4 = 1456 us. Damit ergibt sich für den Offset o zu `o=1/2(1060-1000-1456+1400)=4`.

Die Zeitsynchronisation wird bis zu zehn Mal pro Sekunde zwischen den Teilnehmern durchgeführt. Darüber hinaus legt die Norm einen Mechanismus fest, durch den der Zeitserver, also der Rechner mit der genauesten Uhr, automatisiert identifiziert wird.

3.2.6 Sonstige Realzeitaspekte

Innerhalb eines Realzeitbetriebssystems wird die Berechenbarkeit im Wesentlichen durch zwei Aspekte beeinflusst:

- Die Berechenbarkeit der Algorithmenlaufzeit
- Der Einsatz von Caches

Die Laufzeit von Algorithmen hängt sehr häufig von der Anzahl der zu verarbeitenden Daten ab. Die Abhängigkeit selbst wird durch die Komplexität (Stichwort Landau-Symbol) angegeben. Zusätzlich ist die Laufzeit einiger (schlecht entworfener und implementierter) Algorithmen noch von der Historie abhängig: Je länger der Algorithmus aktiv ist, desto länger benötigt er für einen Durchlauf. Derartige Algorithmen eignen sich nicht für den Einsatz im Realzeitumfeld.

Die hohe Performance moderner Systeme wird nicht zuletzt durch Einsatz diverser Caches erreicht, die die Verarbeitungszeiten der eingesetzten Software stark beeinflussen. Mit Caches werden Zwischenspeicher bezeichnet. Wählt beispielsweise eine Codesequenz eine Adresse aus, um von dieser Daten zu lesen, erfolgt der Zugriff über den Zwischenspeicher. Befinden sich die gesuchten Daten im Zwischenspeicher (Cache-Hit), ist der Zugriff ausgesprochen schnell. Wird aber in der Codesequenz eine Adresse verwendet, deren zugehöriger Inhalt sich nicht im Cache befindet – ein Cache-Miss also –, muss der Cache erst mit den Daten gefüllt werden. In dieser Zeit wartet die zugreifende Codesequenz, sodass es zu (ungewollten) Verzögerungen kommt. Ob es Cache-Misses oder Cache-Hits gibt, ist quasi nicht im Vorhinein berechenbar und damit gibt es kein deterministisches Verhalten. Das würde erst durch das Abschalten der Caches erreicht. Das Abschalten andererseits

führt zu signifikanten Leistungseinbußen. Daher werden Caches nur dann deaktiviert, wenn harte Realzeitanforderungen mit engen Zeitschranken und hohem Sicherheitsrisiko (Safety) zu erfüllen sind.

In der Hardware eines Realzeitsystems finden wir folgende Caches:

- Daten- und Instruktionscache
- Translation Lookaside Buffer (TLB)

Insbesondere Daten- und Code-Caches sind häufig hierarchisch organisiert. Dabei gibt es bis zu drei Cache-Ebenen: Den First-, Second- und Third-Level-Cache. Der First-Level-Cache ist typischerweise der kleinste und gleichzeitig der schnellste. Er wird bei einer Multicore-Architektur exklusiv von einem Core genutzt. Bei den übrigen Caches kann durchaus eine Mitnutzung unterschiedlicher Cores vorkommen.

Der TLB ist ein Cache, der Speicherverwaltungsinformationen zwischenspeichert. Er bevorratet direkt die Zuordnungen zwischen einer virtuellen und der zugehörigen physikalischen Adresse. Da diese Zuordnungen threadspezifisch sind, muss der TLB mit jedem Threadwechsel geleert (geflusht) werden. Ansonsten würde der gerade aktivierte Thread ungültige Umsetzungen verwenden. Um zu verhindern, dass bereits beim Übergang in den Kernel – beispielsweise beim Aufruf eines Systemcalls – ein TLB-Flush notwendig ist, reservieren moderne Betriebssysteme wie Linux einen Teil des virtuellen Adressraums für den Kernel. Das führt zu einer erheblichen Effizienzsteigerung.

Einige Prozessorarchitekturen, ARM bspw., verwenden den normalen Datencache, um dort Einträge der Speicherverwaltung zu puffern.

Weiterhin ist zu beachten, dass Peripheriegeräte, wie beispielsweise Festplatten, ebenfalls Caches enthalten.

In der Systemsoftware finden sich unter anderem folgende Caches:

- Page-Cache
- Objekt-Cache (Buddy-System mit typisierten Objekten)

Der Page-Cache puffert Daten, die zwischen Hauptspeicher und Hintergrundspeicher transportiert werden.

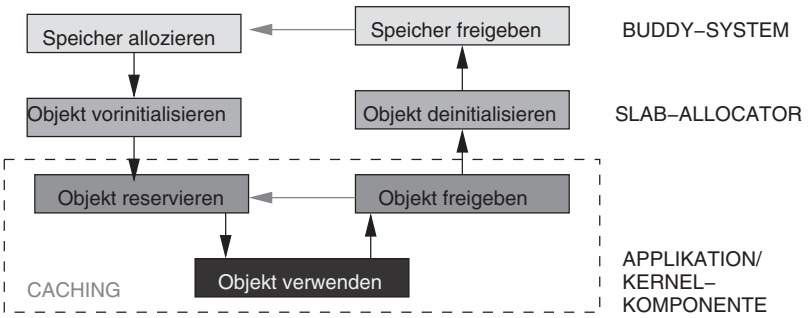


Abbildung 3-24
Zwischenspeicher für Kernelobjekte

Der Objekt-Cache ist Teil der dynamischen Speicherverwaltung im Kernel und bietet den Subsystemen sogenannte typisierte Objekte an. Als Idee steckt dahinter, direkt zum Start des Systems eine Anzahl typisierter Objekte, beispielsweise Spinlocks oder Semaphore, anzulegen und zu initialisieren. Das hat gleich mehrere Vorteile: Erstens geht das Anlegen und Initialisieren durch das Ausnutzen der Prozessor-Caches schneller vonstatten; schließlich wird der Code einmal geladen und dann mehrfach durchlaufen. Zweitens fordern Codesequenzen des Kernels die Objekte an (Objekt reservieren), die dann bereits initialisiert, also ohne weiteren Zeitverlust für den Zugriff vorbereitet sind. Werden diese Objekte wieder freigegeben, kommen sie zurück in den Pool und können bei der nächsten Nachfrage nach einem Objekt diesen Typs wieder verwertet werden.

Typisierte Objekte im Linux-Kernel

Im Linux-Kernel ist der sogenannte Slab-Allocator für die typisierten Objekte und deren Caching verantwortlich. Der Slab-Allocator legt die typisierten Objekte immer in ganzen Speicherseiten beziehungsweise einem Vielfachen davon ab. Für die Verwaltung dieser Speicherseiten, also das Reservieren und Freigeben, ist das Buddy-System zuständig.

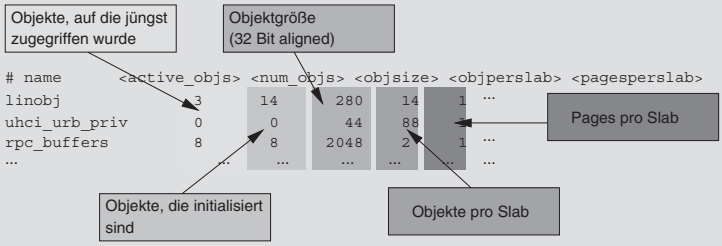


Abbildung 3-25 *Typisierte Objekte im Linux-Kernel*

Aktuelle Informationen bezüglich der im Objekt-Cache befindlichen Objekte, deren Größe, Anzahl und Verwendung lässt sich durch Auslesen der Datei `/proc/slabinfo` generieren (siehe Abbildung 3-25).

3.3 Linux

Linux als Standardbetriebssystem ist in den vergangenen Jahren um Realzeiteigenschaften erweitert worden, sodass es inzwischen ein deterministisches Zeitverhalten mit kurzen Latenzzeiten ermöglicht. Je nach Kernelversion muss der Betriebssystemkern eventuell aber noch mit dem sogenannten PREEMPT-RT-Patch versehen werden. Auf einem sehr leis-

tungsfähigen System sind damit Task-Latenzzeiten von weniger als 10 Mikrosekunden zu realisieren; auf einem ARM-System liegen Task-Latenzzeiten im unteren dreistelligen Mikrosekundenbereich.

Neben Elementen wie das prioritätengesteuerte Scheduling, Memory-Locking oder die Prioritätsinversion, die man in klassischen Realzeitbetriebssystemen findet, bietet Linux darüber hinaus mit Threaded Interrupts oder CPU-Affinität moderne Realzeittechnologien. Dank der hohen Skalierbarkeit und guten Portierbarkeit eignet sich Linux insbesondere für eingebettete Systeme

Scheduling. Linux verwendet ein prioritätengesteuertes (Singlecore-)Scheduling mit insgesamt 140 Prioritätsebenen. Diese Ebenen sind in zwei Gruppen eingeteilt. Die Ebenen von 0 bis 39 repräsentieren den Bereich der dynamischen Prioritäten, von 40 bis 139 handelt es sich um den Bereich der statischen Prioritäten. Während eine statische Priorität vom Scheduler respektiert und nicht verändert wird, erlaubt sich der Scheduler, eine Priorität aus dem Bereich von 0 bis 39 eigenständig zu verändern. Vereinfacht ausgedrückt wird die Priorität einer Task, die sehr viele Ein- und Ausgaben macht und wenig rechnet, erhöht. Eine Task, die sehr viel rechnet, wird je nach Lastsituation in ihrer Priorität erniedrigt. Dieses dynamische Anpassen der Priorität findet allerdings nur innerhalb der Ebenen 0 bis 39 statt.

Gibt es innerhalb einer Prioritätsebene aus dem Bereich der statischen Prioritäten mehrere rechenbereite Tasks, wird die nächste Task abhängig von der für die jeweilige Prioritätsebene gewählten Konfiguration entweder nach FCFS (First Come First Serve) oder nach dem Zeitscheibenverfahren (Round Robin) ausgewählt.

Auf Multicore-Maschinen findet eine Aufteilung wie in Multicore-Scheduling (Seite 59) beschrieben statt.

Speicherverwaltung. Die Speicherverwaltung basiert primär auf Paging. Innerhalb des Kernels kann dynamisch Speicher reserviert werden, wobei hier auf unterer Ebene Speicherseiten über ein Buddy-System reserviert werden, welches eine Fragmentierung des Hauptspeichers bestmöglich vermeidet. Darauf aufbauend werden mit dem Slab-Allocator typisierte Objekte unterstützt, die insbesondere bezüglich des Zeitverhaltens beim Einsatz sehr effizient sind (siehe).

Applikationsseitig bietet Linux die Möglichkeit, um ein deterministisches Zeitverhalten zu ermöglichen, Speicherbereiche im Hauptspeicher zu locken und damit von einer potenziellen, zeitaufwendigen Auslagerung abzusehen. Ohnehin ist es möglich (so wie Android es beispielsweise macht), für den Realzeiteinsatz das Paging komplett abzuschalten.

IO-Management. Linux unterstützt eine Reihe von Dateisystemen, insbesondere auch solche, die sich für eingebettete Systeme eignen. Bei-

spielsweise sind die Dateisysteme JFFS2 oder YAFFS2 für Flash-Speicher prädestiniert.

Gerätetreiber. Linux bringt bereits von Haus aus eine sehr breite Unterstützung für unterschiedliche Hardware mit. Darüber hinaus gibt es ein standardisiertes Interface zur systemkonformen Einbindung von Gerätetreibern. Module ermöglichen die leichte Erweiterbarkeit, insbesondere auch um Treiber.

Unterbrechungsmodell. Nicht nur Applikationen, sondern auch Codesequenzen des Kernels sind unterbrechbar (preemptable), man spricht von sogenannter Kernel-Preemption. Dieses wichtige Merkmal ist der Grund für die kurzen Latenzzeiten, bedingt aber andererseits, dass kritische Abschnitte sorgfältig vor gleichzeitigem Zugriff geschützt werden müssen. Linux hat ein aus vier Ebenen bestehendes Unterbrechungsmodell (siehe Abschnitt 4.3.6). Mithilfe sogenannter Threaded Interrupts reduziert sich das Modell auf drei Ebenen und ermöglicht es dem Entwickler, Threads der eigenen Realzeitapplikation vor aktivierten Interrupt-Service-Routinen abarbeiten zu lassen.

Zeitverwaltung. Bei Linux handelt es sich um ein Tickless-System (Realisierungsformen (Seite 77)), das je nach Situation den nächsten Unterbrechungszeitpunkt ausrechnet. Außerdem werden Watchdogs unterstützt.

Userland. Je nach Anwendungsfall stehen für den Linux-Kernel verschiedene Userlands zur Verfügung. Sehr verbreitet im embedded Bereich ist beispielsweise Multicall-Binary *busybox*, die mehr oder minder Kern eines kompletten Userlands ist ([http://www.busybox.net]). Das Userland im Android-Umfeld wird als *BIONIC* bezeichnet.

Sonstiges. Über die genannten Punkte hinaus bietet Linux eine Reihe von modernen Technologien an, mit denen sich Systeme vor Angriffen von außen schützen lassen (IT-Security).

Selbstbau-Linux für eingebettete Systeme

Erste Erfahrungen mit Linux als eingebettetem System kann man mithilfe der Werkzeuge Buildroot und QEMU machen. QEMU emuliert diverse Hardware-Plattformen, die beispielsweise auf einem x86, einem ARM oder einem PowerPC-Prozessor basieren. Mit Buildroot lässt sich sehr leicht für unterschiedliche Plattformen ein komplettes Linux-System aufbauen, welches aus dem Kernel und einem einfachen Userland besteht [QuKu2011-59].

Als Erstes sollten Sie auf Ihrem Hostsystem `qemu` installieren; auf einem Ubuntu beispielsweise durch `apt-get install qemu qemu-system`. Danach laden Sie Buildroot in der aktuellen Version von der Uclibc-Projektseite ([http://buildroot.uclibc.org/download.html]) herunter und packen es in einem Verzeichnis, beispielsweise `/home/user/system/`, aus. Ein `make help`, im Hauptverzeichnis von Buildroot aufgerufen, lis-

tet die von Haus aus unterstützten Plattformen auf. Wählen Sie beispielsweise `qemu_arm_versatile` für die ersten Versuche. Geben Sie dazu `make qemu_arm_versatile_defconfig` ein. Anschließend können Sie per `make menuconfig` die Parametrierung anpassen. Unter dem Punkt `Kernel`, `Kernel version` lässt sich eine aktuelle Kernelversion, beispielsweise `3.4`, einstellen. Außerdem sollte unter dem Menüpunkt `System Configuration` und dann `Port to run a getty (login prompt) on` anstelle von `ttyAMA0tty1` eingegeben werden. Unter `Build options`, `Number of jobs to run simultaneously` kann man die Anzahl der Prozessoren eintragen, die die Entwicklungsmaschine zur Verfügung stellt. Nach dem Speichern der Konfiguration startet ein `make` den Generierungsvorgang.

`Buildroot` lädt über das Internet die Quellcode-Pakete der benötigten Komponenten herunter. Diese landen im Buildroot-Unterverzeichnis `dl`. Anschließend generiert Buildroot die Toolchain und mithilfe der Toolchain sowohl das Rootfilesystem als auch den Kernel. Das Generieren sämtlicher Komponenten kann abhängig von der Leistungsstärke der eingesetzten Hardware ohne Weiteres eine Stunde in Anspruch nehmen. Alle Ergebnisse des Generierungsprozesses landen im Unterverzeichnis `output`, insbesondere unter `output/images` finden sich das Rootfilesystem (`rootfs.ext2`) und der Kernel (`zImage`).

Wenn alle Komponenten generiert wurden, testen Sie das System mit folgendem Kommando:

```
user@host:~/buildroot-2012.02> \
    qemu-system-arm -M versatilepb \
    -m 128 --kernel output/images/zImage \
    output/images/rootfs.ext2 \
    -append "root=/dev/sda"
```

Die Option `M` spezifiziert die zu emulierende Hardware-Plattform, `-m` die Größe des zur Verfügung stehenden Hauptspeichers (hier 128 Megabyte). Per `-kernel` wird der Kernel angegeben, per `-append` die dem Kernel zu übergebenden Parameter. Hierbei muss dem Kernel zumindest das Gerät mitgeteilt werden, über das auf die Root-Partition zuzugreifen ist. Geht alles gut, erscheint ein neues Fenster mit einem Login. Als User `root` eingeloggt, steht ein rudimentäres System im Konsolenmodus zur Verfügung. Dieses lässt sich per Buildroot-Konfiguration und Neugenerierung in sehr weiten Teilen an die eigenen Bedürfnisse anpassen. Um beispielsweise den Kernel zu konfigurieren, rufen Sie im Hauptverzeichnis von Buildroot `make linux-menuconfig` auf. Viele weitere Möglichkeiten sind in der Buildroot-Dokumentation aufgeführt, die als HTML-Dateien im Unterverzeichnis `docs` von Buildroot liegen.

4 Aspekte der nebenläufigen Realzeitprogrammierung

4.1 Allgemeines

Da Realzeitapplikationen eng mit dem darunter liegenden Betriebssystem verzahnt sind, sind der Entwurf und die Programmierung sehr anspruchsvoll. Realzeitapplikationen greifen sehr häufig auf Peripherie zu und verarbeiten Zeitinformationen. Sie bestehen typischerweise aus mehreren Rechenprozessen, die untereinander per Inter-Process-Communication (IPC) Daten austauschen und die sich synchronisieren müssen. Dadurch entstehen innerhalb der Applikationen sogenannte kritische Abschnitte, die geschützt werden müssen – was wiederum Rückwirkungen auf das Zeitverhalten hat.

Um die Einhaltung der Realzeitbedingungen zu gewährleisten, muss für die Aufteilung der Aufgabe auf Tasks sowie die Konfiguration des Taskgebildes (Prioritätenvergabe) und des Schedulers (Wahl des Scheduling-Verfahrens, Einstellung der Scheduling-Parameter) gesorgt werden. Eventuell ist es sogar notwendig, im Rahmen der Multicore-Programmierung einzelne Rechenprozesse auf Rechnerkerne zu fixieren und das Auslagern von Codebereichen softwaretechnisch zu unterbinden.

Auch wenn Realzeit nicht zwangsläufig »Schnelligkeit« bedeutet, sind Realzeitapplikationen typischerweise besonders effizient realisiert. Realzeitprogrammierer verwenden nur in Ausnahmefällen Floating-Point-Variablen und -Operationen, da diese auch bei Vorhandensein einer Hardware-Unterstützung (Floating-Point-Unit) erheblich Rechenzeit benötigen. Sie versuchen lieber, die Berechnungen mit Integeroperationen durchzuführen. Dazu wird beispielsweise bei einer Division der Divident mit $10^{\text{Anzahl gewünschter Nachkommastellen}}$ multipliziert. Die Rechnung 2/3, die normalerweise im Integerbereich 1 ergibt, ergibt damit bei drei Nachkommastellen `(2x1000/3)=666`.

Die in den folgenden Kapiteln vorgestellten Funktionen sind unter einem Linux nutzbar. Viele der Funktionen sind unter POSIX 1003.1 standardisiert und ermöglichen damit die Erstellung von weitgehend portierbarer Realzeitsoftware.

POSIX

POSIX, Portable Operating System Interface, ist ein Standard, der Programmierinterfaces, systeminterne Ablaufmechanismen der Systemsoftware, Systemkommandos und auch Testmethoden beschreibt. Durch Verwendung des Standards sollen Applikationen leichter auf andere Plattformen portiert werden können, folglich besteht eine Portabilität auf Source-Code-Ebene. POSIX standardisiert insbesondere auch Interfaces zu Realzeitfunktionalitäten.

Die aktuelle Version des Standards ist 1003.1-2008 [POSIX 1003.1]. Sie besteht aus den Kapiteln:

- Basisdefinitionen
- Systeminterfaces und Header
- Kommandos und Werkzeuge

Der Standard ist sehr umfangreich und eine vollständige Implementierung ist für ressourcenbeschränkte, eingebettete Systeme nicht sinnvoll. Daher sind unter der Nummer POSIX 1003.13 POSIX-Profile für den Bereich der Realzeitsysteme definiert worden:

- PSE51 (Minimal)
  Dieses Profil ist für kleine, eingebettete Systeme gedacht, deren Hardware ohne MMU (Memory Management Unit), Festplatte und typischerweise auch ohne Bildschirm auskommt. Softwareseitig handelt es sich um ein Singletasking-System. In dieses Profil fällt beispielsweise die Steuerung eines Toasters.
- PSE52 (Controller)
  Dieses Profil wird für Steuerungen verwendet, die keinen Speicherschutz realisieren und kein Multitasking unterstützen, wohl aber neben den Basisfunktionen auf einen Hintergrundspeicher (Festplatte) zugreifen können. Es unterstützt einfache Dateisysteme. In dieses Profil fallen beispielsweise Geräte zur Langzeitdatenerfassung.
- PSE53 (Dedicated)
  Dieses Profil deckt eingebettete Systeme ab, deren CPU dank eingebauter MMU Speicherschutz realisieren kann und denen ein Hintergrundspeicher (Flash oder Festplatte) zur Verfügung steht, auf dem Daten über ein Dateisystem organisiert abgelegt werden. Es handelt sich um Multitasking-Systeme. Das Profil definiert Interfaces für die Kommunikation und für die asynchrone Ein-/Ausgabe. Beispiel: Basisstationen eines Mobilfunknetzes.
- PSE54 (Multi-Purpose)
  Möglichst generell einsetzbare Systeme mit Realzeitanforderungen fallen in dieses Profil. Systeme mit diesem Profil bieten Speicherschutz, Netzwerk, Dateisysteme und Multiuser-Fähigkeiten. Shells und Systemprogramme sind definiert. Außerdem haben die Systeme Tastatur und Bildschirm, durchaus auch ein grafisches Benutzerinterface (GUI). Beispiel: System zur Flugverkehrsüberwachung.

4.2 Programmtechnischer Umgang mit Tasks

Das Taskmanagement ist für die Erzeugung, das Beenden und die Parametrierung von Tasks zuständig. Zwischen den erzeugenden und den erzeugten Tasks besteht ein Eltern-Kind-Verhältnis. Die Kindtasks erben dabei sämtliche Eigenschaften – man sagt das komplette Environment – des Elternjobs. Dazu gehören beispielsweise die Besitzverhältnisse, die Priorität oder auch Referenzen auf offene Datei- oder Kommunikationsverbindungen.

4.2.1 Tasks erzeugen

Zur Erzeugung von Prozessen steht der Systemcall `pid_t fork(void)` zur Verfügung. Nach dem Aufruf gibt es eine Kopie des Prozesses, der `fork()` aufgerufen hat. Der neue Prozess unterscheidet sich nur in der neuen Prozessidentifikationsnummer (PID) und im Rückgabewert der Funktion. Der Prozess, der `fork()` aufgerufen hat, bekommt die PID der neu angelegten Prozesskopie zurück, der neue Prozess hingegen bekommt `0` zurück. Anhand des Rückgabewertes kann der Programmierer also erkennen, ob es sich um den Eltern- oder um den Kindprozess handelt.

```
if (fork()) {
    /* Parentprocess */
    ...
} else {
    /* Childprocess */
    ...
}
```

Sehr häufig starten Elternprozesse Kindprozesse, damit diese selbstständig eine Verarbeitung durchführen und ein Ergebnis liefern. `Make` beispielsweise startet per `fork()` den Compiler und wartet darauf, dass der Compiler mitteilt, ob er den Kompilierungsvorgang erfolgreich beenden konnte oder beispielsweise wegen eines Syntaxfehlers abbrechen musste. Ist Letzteres der Fall, wird `Make` sinnvollerweise stoppen. Erfolg oder Misserfolg teilt ein Programm über seinen Exitcode mit. Der Exitcode entspricht entweder dem Rückgabewert der Funktion `main()` oder dem Parameter der Funktion `exit()` selbst. Im Elternprozess wird er über die Funktion `pid_t wait(int *status)` oder `pid_t waitpid(pid_t pid, int *status, int options)` abgeholt. Erstere legt an der übergebenen Adresse `status` den Exitcode eines Kindprozesses ab. Welcher Kindprozess das ist – falls mehrere gestartet wurden –, ist am Rückgabewert erkennbar. Um den Rückgabewert eines bestimmten Kindprozesses abzuholen, dient

`waitpid()`. Zur Auswertung des von `wait()` zurückgelieferten Werts dient ein Satz von Makros (siehe Manpage und Beispiel 4-1).

Beispiel 4-1 *fork()*

```
childpid = fork();
if (childpid==0) { /* child */
    ....
    return result;
}
waitpid( childpid, &status, 0 );
printf("child returned 0x%x\n", status);
if (WIFEXITED(status)) {
    printf("child status = %d\n", WEXITSTATUS(status));
}
```

Durch Aufruf der Funktion `int execve(const char *filename, char *const argv[], char *const envp[])` überschreibt ein Prozess sein Codesegment. Über diesen Mechanismus wird ein komplett anderer Code abgearbeitet. Beispiel 4-2 zeigt, wie parallel zum gerade laufenden Prozess das Programm `cat /etc/issue` aufgerufen wird. Soll im Übrigen ein normales Kommando aufgerufen werden, kann auch alternativ die Funktion `int system(const char *command)` verwendet werden.

Beispiel 4-2 *Execve*

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main( int argc, char **argv, char **envp )
{
   int childpid, status;
   char *childargv[] = { "/bin/cat", "/etc/issue", NULL, NULL };
   char *childenvp[] = { NULL };

   childpid = fork();
   if (childpid==(-1) ) {
      perror( "fork" );
      return -1;
   }
   if (childpid==0) { /* child */
      execve("/bin/cat", childargv, childenvp);
   }
   /* parent only */
   printf("waiting for my child %d to die...\n", childpid);
   wait(&status);
   printf("child died...\n");
   return 0;
}
```

Threads erzeugen

Threads werden über die Funktion `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)` erzeugt. Intern verwendet `pthread_create()` den Systemcall `clone()`. Um die Funktion nutzen zu können, muss die Bibliothek `libpthread` zum Programm gebunden werden (Option `-lpthread`). Anders als bei `fork()` startet der neue Thread die Verarbeitung in einer separaten Funktion, deren Adresse über den Parameter `start_routine` übergeben wird. Dieser Funktion wird beim Aufruf das Argument `arg` übergeben. Der Parameter `attr` steuert die Erzeugung, `thread` enthält nach dem Aufruf die Kennung des neuen Threads. `pthread_create()` gibt im Erfolgsfall `0` zurück, ansonsten einen Fehlercode.

Beispiel 4-3
Programmbeispiel Thread-Erzeugung

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *child( void *args )
{
    printf("The child thread has ID %d\n", getpid() );
    // You can either call pthread_exit or "return".
    // pthread_exit( NULL );
    return NULL;
}

int main( int argc, char **argv, char **envp )
{
    pthread_t child_thread;

    if( pthread_create( &child_thread, NULL, child, NULL )!= 0 ) {
        fprintf(stderr,"creation of thread failed\n");
        return -1;
    }
    /* Now the thread is running concurrent to the main task        */
    /* But it won't return to this point (aulthough it might        */
    /* execute a  return statement at the end of the Child routine).*/

    /* waiting for the termination of child_thread */
    pthread_join( child_thread, NULL );
    printf("end of main thread\n");
    return 0;
}
```

4.2.2 Tasks beenden

Ein Rechenprozess beendet sich, wenn er die Hauptfunktion (`main`) durchlaufen hat. Alternativ ruft der Job den Systemcall `void exit(int status)` auf. Typischerweise wird das erfolgreiche Ende durch den Wert null angezeigt, wohingegen ein negativer Code Fehlerzustände indiziert.

Ein Thread beendet sich selbst, indem er `void pthread_exit(void *retval)` aufruft. Der Eltern-Thread verwendet `int pthread_join(pthread_t thread, void **retval)`, um auf den Exitcode des Threads `thread` zu warten (siehe Beispiel 4-3).

Per `int kill(pid_t pid, int sig)` schickt ein Prozess einem anderen Job, der die Prozessidentifikationsnummer `pid` besitzt, das Signal `sig`. Typischerweise führt dieses dazu, dass sich der Job daraufhin beendet. Allerdings können die Signale durch die Programme – mit Ausnahme von `SIGKILL` (Signalnummer 9) – ignoriert oder abgefangen und damit das Beenden verhindert oder auch nur verzögert werden.

```
if (kill(child_pid,SIGINT)==-1) {
    perror("kill failed");
}
```

Die eigene Prozess-Identifikationsnummer (PID) wird durch `pid_t getpid(void)` abgefragt, die Thread-Identifikationsnummer (TID) per `pid_t gettid(void)`. Allerdings gibt es für diesen Linux-spezifischen Systemcall keinen direkten Funktionsaufruf in der Standard-C-Bibliothek.

Eltern-Threads können Kind-Threads durch Aufruf der Funktion `int pthread_kill(pthread_t thread, int sig);` dazu bewegen, sich zu beenden (siehe Beispiel 4-4).

Beispiel 4-4 *Programmbeispiel Threads killen*

```
pthread_t child_thread;

if( pthread_create( &child_thread, NULL, child, NULL )!= 0 ) {
    fprintf(stderr,"creation of thread failed\n");
    return -1;
}
...
pthread_kill( child_thread, SIGINT );
pthread_join( child_thread, NULL );
...
```

4.2.3 Tasks parametrieren

Neu erzeugte Tasks erben die Realzeiteigenschaften der Elterntasks. Um diese anzupassen, steht eine Reihe von Funktionen zur Verfügung.

Per `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)` wird für den Job mit der Priorität `pid` das Schedu-

ling-Verfahren `policy` für die in der Datenstruktur `param` definierte Prioritätsebene eingestellt. Ist der Parameter `pid0`, gelten die Einstellungen für den aufrufenden Prozess. Folgende Scheduling-Verfahren stehen zur Verfügung:

- SCHED_RR: Round Robin
- SCHED_FIFO: First Come First Serve
- SCHED_OTHER: Default-Verfahren für Jobs ohne (Realzeit-)Priorität

Die gerade aktiven Scheduling-Parameter werden durch Aufruf von `int sched_getparam(pid_t pid, struct sched_param *param)` ausgelesen. `pid` referenziert wieder den Job. An der mit `param` übergebenen Speicheradresse legt die Funktion später die Parameter (Priorität, Prioritätsbereich etc.) ab.

Beispiel 4-5
Tasks parametrieren

```
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <time.h>

#define PRIORITY_OF_THIS_TASK     15

char *Policies[] = {
   "SCHED_OTHER",
   "SCHED_FIFO",
   "SCHED_RR"
};

static void print_scheduling_parameter()
{
   struct timespec rr_time;

   printf("Priority-Range SCHED_FF: %d - %d\n",
      sched_get_priority_min(SCHED_FIFO),
      sched_get_priority_max( SCHED_FIFO ) );
   printf("Priority-Range SCHED_RR: %d - %d\n",
      sched_get_priority_min(SCHED_RR),
      sched_get_priority_max( SCHED_RR));
   printf("Current Scheduling Policy: %s\n",
      Policies[sched_getscheduler(0)] );
   sched_rr_get_interval( 0, &rr_time );
   printf("Intervall for Policy RR: %ld [s] %ld [nanosec]\n",
      rr_time.tv_sec, rr_time.tv_nsec );
}

int main( int argc, char **argv )
{
   struct sched_param scheduling_parameter;
```

```
   struct timespec t;
   int i,j,k,p;

   print_scheduling_parameter();
   sched_getparam( 0, &scheduling_parameter );
   printf("Priority: %d\n", scheduling_parameter.sched_priority );

   // only superuser:
   // change scheduling policy and priority to realtime priority
   scheduling_parameter.sched_priority = PRIORITY_OF_THIS_TASK;
   if( sched_setscheduler( 0, SCHED_RR, &scheduling_parameter )!= 0 ) {
      perror( "Set Scheduling Priority" );
      exit( -1 );
   }
   sched_getparam( 0, &scheduling_parameter );
   printf("Priority: %d\n", scheduling_parameter.sched_priority );

   t.tv_sec = 10;          // sleep
   t.tv_nsec = 1000000;
   nanosleep( &t, NULL );

   print_scheduling_parameter();
   return 0;
}
```

Thread-Attribute festlegen

Threads können bei der Erzeugung beispielsweise bezüglich der Stackgröße parametriert werden. Dazu wird – anders als in den bisherigen Beispielen – ein POSIX-Attribut-Objekt (Typ `pthread_attr_t`) erzeugt und der Funktion `pthread_create()` als Parameter übergeben. Das Attribut-Objekt kann mehrfach verwendet und nach der Erzeugung der Threads auch wieder freigegeben werden. Der Zugriff auf die Attribute des Attribut-Objekts darf nur über Zugriffsfunktionen erfolgen, die in Tabelle 4-1 aufgelistet sind.

Tabelle 4-1
Methoden für die Zugriffe auf Thread-Attribute (Auswahl)

Funktion	Beschreibung
`pthread_attr_init`, `pthread_attr_destroy`	Objekt initialisieren bzw. deinitialisieren
`pthread_attr_setstacksize`, `pthread_attr_getstacksize`	Diese Methoden geben Zugriff auf die Stackgröße.
`pthread_attr_setstackaddr`, `pthread_attr_getstackaddr`	Diese Methoden geben Zugriff auf die Stackadresse.

Funktion	Beschreibung
`pthread_attr_setdetachstate`, `pthread_attr_getdetachstate`	Threads können »detached« (PTHREAD_CREATE_DETACHED) oder »joinable« (PTHREAD_CREATE_JOINABLE) sein. Im Zustand »detached« dürfen die zugehörigen Thread-Ressourcen, beispielsweise die Thread-Id oder der TCB, direkt neu genutzt werden. Im Zustand »joinable« ist das erst möglich, nachdem `pthread_join()` aufgerufen worden ist.
`pthread_attr_setscope`, `pthread_attr_getscope`	Der sogenannte Contention Scope definiert, in welchem Kontext das Scheduling eines Threads stattfindet. Threads mit dem Scope PTHREAD_SCOPE_SYSTEM werden vom normalen Scheduler verwaltet, Threads mit dem Scope PTHREAD_SCOPE_PROCESS werden von einem POSIX-internen Scheduler verarbeitet. Dieser kommt dann zum Einsatz, wenn der System-Scheduler die Threadgruppe ausgewählt hat.
`pthread_attr_setinheritsched`, `pthread_attr_getinheritsched`	Dieses Attribut legt fest, ob die Attribute eines erzeugten Threads identisch mit denen des erzeugenden Threads (PTHREAD_INHERIT_SCHED) sind oder ob diese geändert werden können (PTHREAD_EXPLICIT_SCHED).
`pthread_attr_setschedpolicy`, `pthread_attr_getschedpolicy`	Diese Methoden geben Zugriff auf die Scheduling-Policy.
`pthread_attr_setschedparam`, `pthread_attr_getschedparam`	Diese Methoden geben Zugriff auf die Scheduling-Parameter.

Task-Affinität

Multicore-Architekturen bieten durch die CPU-Isolation und -Affinität interessante Möglichkeiten, Anforderungen an das Zeitverhalten zu erfüllen. Dazu werden einzelne Prozessor-Kerne für Realzeitaufgaben reserviert und kritische Jobs auf diese Kerne fixiert. Zentral hierfür ist die Datenstruktur `cpu_set_t`, die die Prozessorkerne repräsentiert. Jeder Kern `cpu` ist dabei durch ein einzelnes Bit dargestellt. Auf diese Datenstruktur ist eine Reihe von Funktionen (Methoden) anwendbar, die das Bitfeld `set` löschen (`void CPU_ZERO(cpu_set_t *set)`) und einzelne Bits setzen (`void CPU_SET(int cpu, cpu_set_t *set)`) beziehungsweise rücksetzen (`void CPU_CLR(int cpu, cpu_set_t *set)`). Per `int CPU_ISSET(int cpu, cpu_set_t *set)` wird überprüft, ob die CPU `cpu` im Set `set` gesetzt ist oder nicht.

Um einen Job auf spezifische CPU-Kerne zu fixieren, verwenden Sie die Funktion `int sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask)`. Multicore-Scheduling (Seite 59) zeigt die programm-

technische Anwendung. Die Vorbereitung des Systems – das Abschotten einzelner Prozessorgruppen (Isolation) – ist im Multicore-Scheduling (Seite 59) beschrieben.

Beispiel 4-6 *Erzwungene Taskmigration*

```
#include <stdio.h>
#include <stdlib.h>
#define __USE_GNU
#include <sched.h>
#include <errno.h>

int main( int argc, char **argv )
{
   cpu_set_t mask;
   unsigned int cpuid, pid;

   if( argc != 3 ) {
      fprintf(stderr,"usage: %s pid cpuid\n", argv[0]);
      return -1;
   }
   pid = strtoul( argv[1], NULL, 0 );
   cpuid = strtoul( argv[2], NULL, 0 );
   CPU_CLR( cpuid, &mask ); // Aktuelle CPU verbieten.
   if( sched_setaffinity( pid, sizeof(mask), &mask ) ) {
      perror("sched_setaffinity"); // UP-System???
      return -2;
   }
   return 0;
}
```

4.3 Schutz kritischer Abschnitte

Zur Lösung einer Realzeitaufgabe werden im Normalfall mehrere Threads oder Prozesse verwendet, die (quasi-)parallel arbeiten. Man unterscheidet dabei

- disjunkte Prozesse, bei denen der Ablauf unabhängig von den anderen Prozessen ist,
- nicht disjunkte Prozesse, die gemeinsame Daten oder Ressourcen nutzen. Diese lassen sich wiederum unterscheiden in:
  - konkurrierende Prozesse, die um den Zugriff auf Daten konkurrieren, und
  - kooperierende Prozesse (meist verkettet), bei denen der eine Prozess Daten für den anderen Prozess liefert (Hersteller-/Verbraucher-modell).

Die Wirkung der gegenseitigen Beeinflussung nicht disjunkter paralleler Prozesse ist ohne Synchronisation nicht vorhersagbar und im Regelfall nicht reproduzierbar.

Der Programmteil, in dem auf gemeinsame Betriebsmittel (z.B. Daten) zugegriffen wird, heißt *kritischer Abschnitt (Critical Section)*. Greifen zwei oder mehr Codesequenzen (Threads, Prozesse oder auch Interrupt-Service-Routinen) auf dieselben Daten zu, kann es zu einer sogenannten *Race Condition* kommen. Bei einer Race Condition hängt das Ergebnis des Zugriffs vom Prozessfortschritt ab.

4.3.1 Semaphor und Mutex

Race Conditions lassen sich vermeiden, indem nie mehr als ein Prozess in einen kritischen Abschnitt eintritt (Mutual Exclusion, gegenseitiger Ausschluss). Dies lässt sich mithilfe von Semaphoren sicherstellen.

Ein Semaphor wird zur Synchronisation, insbesondere bei gegenseitigem Ausschluss, eingesetzt. Es handelt sich dabei um eine Integer-Variable, die wie folgt verwendet wird:

- Der Wert des Semaphors wird auf einen Maximalwert N initialisiert.
- Bei einem Zugriff auf das Semaphor (P-Operation, das P stammt aus dem Niederländischen und steht für *passeeren*) wird
  - dessen Wert um 1 erniedrigt und
  - der Prozess *schlafend* gelegt, wenn der neue Semaphor-Wert negativ ist.
- Bei der Freigabe eines Semaphors (V-Operation, V leitet sich vom Niederländischen *vrijgeven* ab) wird
  - dessen Wert um 1 erhöht und,
  - falls der neue Semaphor-Wert negativ oder gleich null ist, ein auf das Semaphor wartender (schlafender) Prozess aufgeweckt.

P-Operation

```
s = s - 1;
if (s < 0) {
    sleep_until_semaphore_is_free();
}
```

V-Operation

```
s = s + 1;
if (s <= 0) {
    wake_up_sleeping_process_with_highest_priority();
}
```

P- und V-Operationen sind selbst kritische Abschnitte, deren Ausführung nicht unterbrochen werden darf. Daher sind Semaphor-Operationen im Betriebssystem als Systemcalls verankert und zur eigenen Absicherung werden Unterbrechungssperren eingesetzt.

Bei einfachen Mikroprozessoren bzw. Laufzeitsystemen werden Semaphor-Operationen ansonsten durch Test-and-Set-Befehle des Mikroprozessors realisiert.

Abbildung 4-1
Anwendung von Semaphor-Operationen

UNGESCHÜTZTE CODEPASSAGE

```
new_element = create_new_element();
for (ptr=root; ptr->next; ptr=ptr->next)
     ;
ptr->next = new_element;
```

GESCHÜTZTE CODEPASSAGE

```
new_element = create_new_element();
P(S);
for (ptr=root; ptr->next; ptr=ptr->next)
     ;
ptr->next = new_element;
V(S);
```

Ein Mutex ist ein Semaphor mit dem Maximalwert N = 1 (binäres Semaphor).

POSIX-Funktionen für Mutexe sind:

```
int pthread_mutex_init
( pthread_mutex_t * mutex , const pthread_mutexattr_t * mutexattr );
```

Diese Funktion initialisiert ein Mutex `mutex` gemäß der Angaben im Parameter `mutexattr`. Ist `mutexattr` NULL, werden Default-Werte zur Initialisierung verwendet.

```
int pthread_mutex_destroy
( pthread_mutex_t * mutex );
```

Über diese Funktion wird ein Mutex (Semaphor, das genau einem Thread den Zugriff auf den kritischen Abschnitt ermöglicht) gelöscht und die damit zusammenhängenden Ressourcen werden freigegeben. Ein Mutex wird nur entfernt, wenn es »frei« (unlocked) ist.

```
int pthread_mutex_lock
( pthread_mutex_t * mutex );
```

Über diese Funktion wird ein kritischer Abschnitt betreten. Sollte der kritische Abschnitt frei sein, wird das Mutex gesperrt (locked) und dem aufrufenden Thread zugehörig erklärt. In diesem Fall returniert die Funktion direkt. Sollte der kritische Abschnitt gesperrt sein, wird der aufrufende Thread so lange in den Zustand »schlafend« versetzt, bis das Mutex wieder freigegeben wird.

Ist das Mutex durch denselben Thread gesperrt, der den Aufruf durchführt, kann es zu einem Deadlock kommen. Um dies zu verhindern, lässt sich ein Mutex so konfigurieren, dass es entweder den Fehlercode EDEADLK returniert (ein »Error Checking Mutex«) oder den Mehrfachzugriff erlaubt (ein »Recursive Mutex«).

```
int pthread_mutex_trylock
( pthread_mutex_t * mutex );
```

Diese Funktion verhält sich im Prinzip wie die Funktion , nur blockiert sie nicht, falls der kritische Abschnitt nicht betreten werden kann. In diesem Fall returniert die Funktion EBUSY.

```
int pthread_mutex_unlock
( pthread_mutex_t * mutex );
```

Über diese Funktion wird ein kritischer Abschnitt wieder verlassen.

Beispiel 4-7
Programmbeispiel Mutex

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

static pthread_mutex_t mutex;

int main( int argc, char **argv, char **envp )
{
   pthread_mutex_init( &mutex, NULL );

   pthread_mutex_lock( &mutex ); // enter critical section
   printf("%d in critical section...\n", getpid());
   pthread_mutex_unlock( &mutex ); // leave critical section

   pthread_mutex_destroy( &mutex );
   return 0;
}
```

Ein pthread-Mutex darf bei normaler Konfiguration nicht rekursiv reserviert werden. Rekursiv bedeutet, dass der Thread, der ein Mutex bereits reserviert hat, dieses erneut reserviert. In diesem Fall ist das Verhalten nicht definiert. Ähnlich ist es, wenn Thread A ein Mutex reserviert und Thread B selbiges wieder freigibt. Ebenfalls nicht definiert ist das Verhalten, bei dem ein Thread ein Mutex freigibt (unlockt), welches nicht reserviert ist.

Ein Semaphor kann zur *kooperativen* Synchronisation zwischen Rechenprozessen verwendet werden, indem die zusammengehörigen P- und V-Operationen aufgesplittet werden (siehe Abschnitt 4.6). pthread-Mutexe unterstützen diese Möglichkeit jedoch nicht. Nur der Thread, der die Funktion `pthread_mutex_lock()` aufgerufen hat, darf das zugehörige `pthread_mutex_unlock()` aufrufen.

4.3.2 Programmtechnische Behandlung der Prioritätsinversion

Der Umgang mit Prioritätsinversion wird bei der Initialisierung eines Mutex über die Funktion `pthread_mutexattr_setprotocol()` konfiguriert. Dabei hat der Programmierer mit dem Define `PTHREAD_PRIO_NONE` die Möglichkeit, ganz auf die Behandlung einer Prioritätsinversion zu verzichten, per `PTHREAD_PRIO_INHERIT` die klassische Prioritätsvererbung (Priority Inheritance) oder per `PTHREAD_PRIO_PROTECT` das Protokoll Priority Ceiling zu aktivieren.

Wird Priority Ceiling aktiviert, kann über `pthread_mutexattr_getprioceiling()` und `pthread_mutexattr_setprioceiling()` die obere Schranke (Ceiling) gelesen beziehungsweise auch gesetzt werden.

Beispiel 4-8 zeigt, wie bei der Initialisierung des Mutex Prioritätsvererbung eingeschaltet wird. Allerdings ist diese nur dann wirksam, wenn der Rechenprozess eine Realzeitpriorität besitzt, also der Scheduling-Klasse `rt_sched_class` zugeordnet ist (siehe Abschnitt 4.2.3).

Beispiel 4-8 *Prioritätsvererbung aktivieren*

```
#include <stdio.h>
#include <unistd.h>
#define __USE_UNIX98 /* Needed for PTHREAD_PRIO_INHERIT */
#include <pthread.h>

static pthread_mutex_t mutex;
static pthread_mutexattr_t attr;

int main( int argc, char **argv, char **envp )
{
   // don't forget: rt-priority is needed for priority inheritance
   pthread_mutexattr_init( &attr );
   pthread_mutexattr_setprotocol( &attr, PTHREAD_PRIO_INHERIT );
   pthread_mutex_init( &mutex, &attr );
```

```
    pthread_mutex_lock( &mutex ); // enter critical section
    printf("%d in critical section...\n", getpid());
    pthread_mutex_unlock( &mutex ); // leave critical section

    pthread_mutex_destroy( &mutex );
    return 0;
}
```

4.3.3 Deadlock

Durch Critical Sections (gegenseitigen Ausschluss) kann es leicht zu Verklemmungen, den sogenannten *Deadlocks*, kommen. Muss eine Task beispielsweise zwei Datenstrukturen manipulieren, die durch zwei unabhängige Semaphore geschützt sind, und eine zweite Task muss das Gleiche tun, nur dass sie die Semaphore in umgekehrter Reihenfolge alloziert, ist eine Verklemmung die Folge (Deadlock, Abbildung 4-2).

In der Praxis kommen derartige Konstellationen häufig vor. Dabei sind sie nur sehr schwer zu entdecken, da das Nehmen und Freigeben des Semaphors nicht selten in Funktionen gekapselt ist. Die Erkennung, Vermeidung und Behandlung von Deadlocks gehört zu den schwierigsten Aufgaben bei der Programmierung moderner Realzeitanwendungen. Beim Software-Entwurf können Deadlocks unter Umständen durch die Modellierung und Analyse des zugehörigen Petrinetzes entdeckt werden (siehe Abschnitt 7.3).

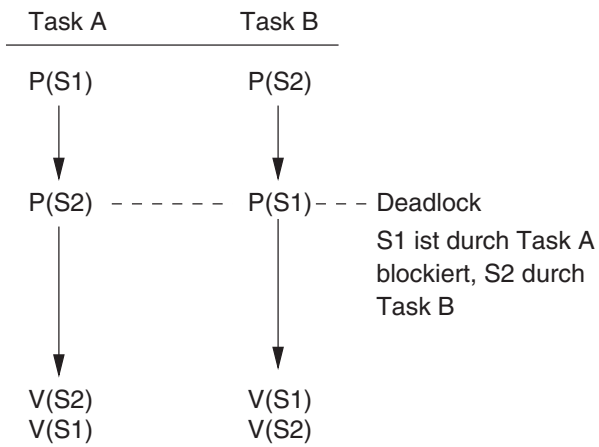


Abbildung 4-2
Deadlock

Deadlocks vermeiden

Damit in einem System Deadlocks auftreten, müssen folgende vier Bedingungen gleichzeitig erfüllt sein:

1. Ressourcen stehen nur exklusiv zur Verfügung, mehrere Tasks können somit nicht gleichzeitig auf die Ressourcen zugreifen.
2. Eine Task arbeitet mit mehreren Ressourcen gleichzeitig, hält also eine Ressource, während sie auf eine andere wartet.
3. Die Ressource kann einer Task nicht einfach entzogen werden.
4. Bei der Benutzung der Ressourcen der Tasks untereinander entsteht eine zyklische Kette. Wenn also z.B. Task A die Ressource R1 hat und auf R2 wartet, die Task B auf die Ressource R1 wartet, während sie R2 hat, entsteht eine zyklische Kette.

Der Programmierer muss nun einfach eine der vier Bedingungen im System vermeiden, dann können auch nie Deadlocks auftreten. Klingt einfacher als gesagt.

An der 1. Bedingung lässt sich nicht viel ändern, denn viele Ressourcen können nur exklusiv genutzt werden, damit die Datenkonsistenz gewahrt bleibt.

Würden in einem System alle Tasks nur immer eine Ressource gleichzeitig benutzen (vgl. 2. Bedingung), könnten – abgesehen vom Selflock – keine Deadlocks mehr auftreten. Ein Selflock tritt auf, wenn eine Task erneut eine Ressource locken will, die sie bereits benutzt und somit gelockt hat. Durch den erneuten Lockaufruf würde die Task für immer blockiert werden.

Alternativ kann eine Tasks auch auf die Freigabe aller benötigten Ressourcen warten, bevor sie eine nach dem anderen versucht zu locken. Die Wartezeit für eine Task kann je nach Menge der Ressourcen jedoch sehr hoch werden.

Die 3. Bedingung hängt von Art der Ressource ab, denn nicht jede Ressource kann jederzeit einer Task ohne Datenverlust entzogen werden.

Unter der Voraussetzung der ersten drei Bedingungen kommt es schließlich mit der 4. Bedingung zum klassischen Deadlock. Dieser kann verhindert werden, indem eine lineare Ordnung für den Zugriff auf Ressourcen vorgeschrieben wird. Greifen alle Tasks auf die Ressourcen in der gleichen Reihenfolge zu, also zuerst R1 und dann R2, kann es zu keinem Deadlock kommen.

Ansetzen kann man als Programmierer somit an der 2. und 4. Bedingung. Jedoch sind die Lösungen dafür nicht effizient, denn die Tasks warten unter Umständen sehr viel länger, bis sie freie Ressourcen tatsächlich nutzen.

4.3.4 Schreib-/Lese-Locks

Ein Job wird bei korrektem Einsatz eines Mutex blockiert, sobald er auf einen kritischen Abschnitt zugreifen möchte, der bereits durch einen anderen Job belegt ist.

Falls der kritische Abschnitt aus dem Zugriff auf globale Daten besteht, käme es aber nur dann zu einer Race-Condition, falls die zugrei-

fenden Rechenprozesse die Daten modifizieren. Das parallele Lesen ist jedoch unkritisch. Aus dieser Erkenntnis heraus bieten viele Systeme sogenannte Schreib-/Lese-Locks an. Dabei handelt es sich um Semaphore, die parallele Lesezugriffe erlauben, aber einem schreibenden Job einen exklusiven Zugriff ermöglichen.

Bei der Anforderung des Mutex muss der Rechenprozess mitteilen, ob er den kritischen Abschnitt nur zum Lesen oder auch zum Schreiben betreten möchte. Folgende Fälle werden unterschieden:

1. Der kritische Abschnitt ist frei:
   - ❑ Der Zugriff wird gewährt.
2. Der kritische Abschnitt ist von mindestens einem Rechenprozess belegt, der lesend zugreift, und es gibt zurzeit keinen Rechenprozess, der schreibend zugreifen möchte:
   - ❑ Eine Task, die lesen möchte, erhält Zugriff.
   - ❑ Eine Task, die schreiben möchte, wird blockiert.
3. Der kritische Abschnitt ist von mindestens einer Task belegt, die lesend zugreift und es gibt zurzeit mindestens eine Task, die schreibend zugreifen möchte:
   - ❑ Die anfragende Task wird blockiert (unabhängig davon, ob sie lesen oder schreiben möchte). Würde eine Task, die lesen möchte, nicht blockiert, könnte die Task, die schreiben möchte, »sterben«.
4. Der kritische Abschnitt ist von einer Task belegt, die schreibend zugreift:
   - ❑ Die anfragende Task wird blockiert (unabhängig davon, ob sie lesen oder schreiben möchte).

4.3.5 Weitere Schutzmaßnahmen für kritische Abschnitte

Ein Semaphor lässt sich dann zum Schutz kritischer Abschnitte einsetzen, wenn der Abschnitt durch zwei oder mehr Applikationen (User-Prozesse) betreten werden soll. Kritische Abschnitte aber, die beispielsweise innerhalb des Betriebssystemkerns, eines Treibers oder in Interrupt-Service-Routinen vorkommen, können damit nicht gesichert werden. Hier werden andere Techniken benötigt: Unterbrechungssperren und Spinlocks.

Unterbrechungssperre. Hierbei werden die Interrupts oder auch nur die Taskwechsel auf einem System für die Zeit des Zugriffs auf den kritischen Abschnitt gesperrt. Um die Latenzzeiten kurz zu halten, darf der Zugriff selbst nur kurz dauern. Bei modernen Betriebssystemen ist diese

Methode nur für Zugriffe innerhalb des Betriebssystemkerns, also beispielsweise bei der Realisierung von Treibern, interessant. Außerdem verhindert eine lokale Interruptsperre nicht den real parallelen Zugriff (siehe Unterbrechungssperre (Seite 28)).

Spinlocks. Da bei Multiprozessorsystemen (Symmetric Multiprocessing, SMP) zwei oder mehr Interrupt-Service-Routinen real-parallel bearbeitet werden können, hilft eine lokale Unterbrechungssperre nicht weiter. Eine globale, also für alle Prozessorkerne geltende Unterbrechungssperre führt zu unerwünschten Latenzzeiten. Daher arbeitet man im Regelfall mit sogenannten Spinlocks.

Bei Spinlocks entscheidet – wie schon beim Semaphor – eine Variable darüber, ob ein kritischer Abschnitt betreten werden darf oder nicht. Ist der kritische Abschnitt bereits besetzt, wartet die zugreifende Einheit aktiv so lange, bis der kritische Abschnitt wieder freigegeben worden ist (siehe Abbildung 4-3).

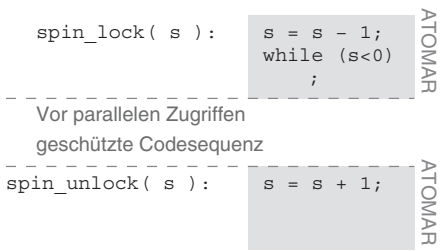


Abbildung 4-3
Aktives Warten beim Einsatz von Spinlocks

Damit ergeben sich für die vorgestellten drei Methoden zum gegenseitigen Ausschluss bei einem kritischen Abschnitt unterschiedliche Einsatzfelder. Die Unterbrechungssperre steht normalen Applikationen nicht zur Verfügung. Bei Einprozessorsystemen (Uni-Processor, UP) wird sie im Betriebssystemkern eingesetzt. Bei SMP lassen sich zwar die Interrupts auf allen Prozessoren sperren, bevor aber auf den kritischen Abschnitt zugegriffen werden kann, ist sicherzustellen, dass nicht zufällig zwei Prozessoren die ISR bearbeiten.

Spinlocks werden ebenfalls im Kernel eingesetzt. Sie realisieren ein aktives Warten und lassen sich aus diesem Grund im Kernel nur in einer SMP-Umgebung verwenden. Auf Applikationsebene funktionieren sie auch in Einprozessorumgebungen. Spinlocks werden einem Mutex oder Semaphor dann vorgezogen, wenn die Bearbeitungszeit des kritischen Abschnitts deutlich kürzer ist als die Zeit, die für einen Kontextwechsel benötigt wird.

Das Semaphor schließlich ist für den gegenseitigen Ausschluss auf Applikationsebene gedacht. Es ist sowohl für UP- als auch für SMP-Umgebungen verwendbar. Der Einsatz im Betriebssystemkern ist möglich,

aber nur dann, wenn der kritische Abschnitt nicht von einer ISR betreten werden soll. In diesem Fall müsste man nämlich die ISR schlafen legen, was nicht möglich ist.

4.3.6 Unterbrechungsmodell

Das Unterbrechungsmodell ist verantwortlich für den Schutz kritischer Abschnitte und damit auch für Latenzzeiten im System. Es teilt das Betriebssystem in mehrere Ebenen ein.

Der Systemarchitekt muss es kennen, da die Ebenen unterschiedlich priorisiert sind und er durch Zuordnung von Funktionen zu Ebenen das Realzeitverhalten generell beeinflusst. Der Programmierer muss es kennen, da er nur so kritische Abschnitte identifizieren und effizient schützen kann.

Grundsätzlich muss man hierfür die quasi-parallelen von der real-parallelen Verarbeitung unterscheiden. Erstere entsteht durch die Unterbrechbarkeit: Auf einer Einkernmaschine wird ein einzelner Job A unterbrochen (preempted), ein anderer Job B wird lauffähig und abgearbeitet und nach einiger Zeit wird auch dieser wieder unterbrochen und Job A fortgesetzt. Damit laufen, wenn auch ineinander geschachtelt, Job A und Job B (quasi-)parallel.

Auf einer Mehrkernmaschine dagegen kann Job A auf einem Prozessorkern und Job B auf einem anderen Prozessorkern zeitgleich, real-parallel ablaufen. Diese Parallelverarbeitung ist im Unterbrechungsmodell nicht direkt sichtbar.

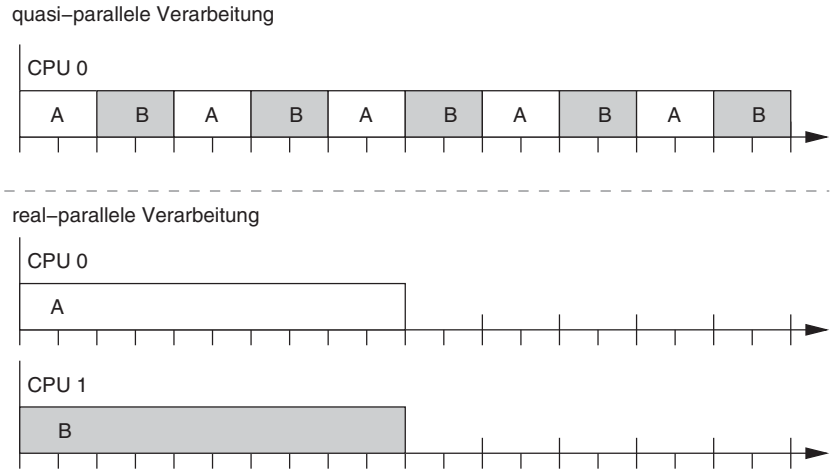


Abbildung 4-4 *Quasi- und real-parallele Verarbeitung*

Im Linux-Kernel gibt es das in Abbildung 4-5 dargestellte Unterbrechungsmodell, das aus vier Ebenen besteht.

1. Die Applikationsebene (Userland).
2. Die Kernel-Ebene.
3. Die Soft-IRQ-Ebene.
4. Die Interrupt-Ebene.

Auf der Applikationsebene, dem Userland, laufen die Userprogramme, die beispielsweise über Prioritäten gescheduled werden. Kritische Abschnitte, wie die Zugriffe auf globale Variablen, werden über Semaphore oder Spinlocks geschützt.

Rufen Applikationen Systemcalls auf (beispielsweise `open()`, `read()` oder `clock_nanosleep()`), werden diese auf der Kernel-Ebene im User-Kontext abgearbeitet. Der Systemcall hat dabei die Priorität des aufrufenden Jobs. Auf gleicher Ebene laufen Kernel-Threads ab, die sich dadurch von User-Threads unterscheiden, indem sie keine Ressourcen (z.B. Speicher) im Userland belegen und damit im Kernel-Kontext arbeiten. Kernel-Threads haben eine Priorität und konkurrieren mit User-Threads. Anders als User-Threads werden die Kernel-Threads jedoch nicht unterbrochen.

Höchste Priorität in der Abarbeitungsreihenfolge haben Interrupt-Service-Routinen, wobei Linux standardmäßig alle Interrupts gleich priorisiert. Man spricht hier auch von Hardware-Priorität. Sobald ein Interrupt auftritt und Interrupts freigegeben sind, werden sie auf dem lokalen Kern gesperrt und die zugehörige ISR wird abgearbeitet. Eine quasi-parallele Verarbeitung findet damit nicht statt. Auf einer Mehrkernmaschine können real jedoch mehrere ISRs parallel verarbeitet werden, ein und dieselbe typischerweise jedoch nicht. Eine Datenstruktur, die von nur einer ISR verwendet wird, muss daher nicht gesondert geschützt werden.

Sind anstehende ISRs abgearbeitet, gibt der Kernel Interrupts frei und startet Soft-IRQs, die ebenfalls im Interruptkontext abgearbeitet werden. Soft-IRQs können nicht unterbrochen werden, können aber auf einer Mehrkernmaschine real-parallel abgearbeitet werden. Doch auch hier gilt die Einschränkung, dass ein und derselbe Soft-IRQ nicht mehrfach gleichzeitig ablaufen kann.

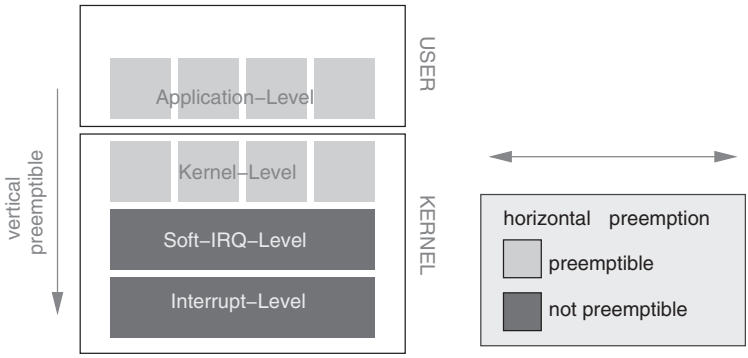


Abbildung 4-5
Unterbrechungsmodell

Da Funktionen, die im Interrupt-Kontext ablaufen, nicht schlafen gelegt werden können, kann ein kritischer Abschnitt auf dieser Ebene niemals per Semaphor geschützt werden. Hier wird auf dem lokalen Kern eine Interruptsperre eingesetzt und für die reale Parallelität ein Spinlock verwendet.

Aktuelle Linux-Kernel bieten die Möglichkeit von sogenannten Threaded Interrupts. Hierbei laufen Interrupts als Kernel-Threads, also im Kernel-Kontext ab. Vorteil: Threads können durch den Systemarchitekten untereinander und sogar bezüglich sonstiger User-Tasks priorisiert werden (siehe Abschnitt 5.3).

4.4 Umgang mit Zeiten

Betriebssysteme stellen Realzeitapplikationen Zeitgeber mit unterschiedlichen Eigenschaften zur Verfügung, über die das Zeitverhalten kontrolliert wird. Diese sind gekennzeichnet durch ihre

- Genauigkeit,
- die Zuverlässigkeit,
- den Bezugspunkt,
- die Darstellung und
- den maximalen messbaren Zeitbereich.

Das Betriebssystem repräsentiert Zeiten unter anderem mit den folgenden Datentypen (Darstellung):

- `time_t`: Zeit in Sekundenauflösung.
- `clock_t`: Timerticks.
- `struct timeval`: Zeit in Mikrosekunden-Auflösung.
- `struct timespec`: Zeit in Nanosekunden-Auflösung.
- `struct tms`: Accounting, Rechen- und Reaktionszeiten.
- `struct tm`: absolute Zeitangabe.

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};

struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};

struct tms {
    clock_t tms_utime;  /* user time */
    clock_t tms_stime;  /* system time */
    clock_t tms_cutime; /* user time of children */
    clock_t tms_cstime; /* system time of children */
};

struct tm {
    int tm_sec;         /* seconds */
    int tm_min;         /* minutes */
    int tm_hour;        /* hours */
    int tm_mday;        /* day of the month */
    int tm_mon;         /* month */
    int tm_year;        /* year */
    int tm_wday;        /* day of the week */
    int tm_yday;        /* day in the year */
    int tm_isdst;       /* daylight saving time */
};
```

Der Datentyp `time_t` repräsentiert typischerweise einen vorzeichenbehafteten `long`-Datentyp. Der negative Zeitbereich wird genutzt, um Zeiten vor dem Bezugspunkt (z. B. vor dem 1.1.1970) auszudrücken.

Die Strukturen `struct timeval` und `struct timespec` bestehen aus jeweils zwei Variablen, die einmal den Sekundenanteil und einmal den Mikro- beziehungsweise den Nanosekundenanteil repräsentieren. Die Darstellung erfolgt jeweils normiert. Das bedeutet, dass der Mikro- oder Nanosekundenanteil immer kleiner als eine volle Sekunde bleibt. Ein Zeitstempel von beispielsweise *1 Sekunde, 123456 Mikrosekunden* ist gültig, *1 Sekunde, 1234567 Mikrosekunden* ist ungültig. In normierter Darstellung ergäben sich *2 Sekunden, 234567 Mikrosekunden*.

Die Darstellungs- beziehungsweise Repräsentationsform reflektiert auch den darstellbaren Wertebereich. Da bei den dargestellten Datenstrukturen für den Typ `time_t` ein `long` eingesetzt wird, lassen sich auf einer 32-Bit-Maschine rund 4 Milliarden Sekunden zählen, auf einer 64-Bit-Maschine 2^{64} (mehr als 500 Milliarden Jahre).

Folgende Bezugspunkte für Zeiten haben sich eingebürgert:

- Start des Systems,

- Start eines Jobs und
- Start einer Epoche, beispielsweise »Christi Geburt« oder der 1.1.1970 (Unix-Epoche). Dieser Bezugspunkt weist zudem noch eine örtliche Komponente auf: Der Zeitpunkt 19:00 Uhr in Europa entspricht einem anderen Zeitpunkt in den USA (minus sechs Stunden zur Ostküste).

Die Genauigkeit wird beeinflusst durch die Taktung des Zeitgebers, deren Schwankungen und durch Zeitsprünge.

Das Attribut *Zuverlässigkeit* eines Zeitgebers beschreibt dessen Verhalten bei (bewussten) Schwankungen der Taktung und bei Zeitsprüngen: Ein Zeitgeber kann beispielsweise der Systemuhr folgen (CLOCK_REALTIME) oder unabhängig von jeglicher Modifikation an der Systemzeit einfach weiter zählen (CLOCK_MONOTONIC). Die POSIX-Realzeiterweiterung beziehungsweise Linux-Systeme definieren hierzu folgende Clocks [Manpage: clock_gettime]:

CLOCK_REALTIME
: Dieser Zeitgeber repräsentiert die systemweite, aktuelle Zeit. Er reagiert auf Zeitsprünge (sowohl vorwärts als auch rückwärts), die beispielsweise beim Aufwachen (Resume) nach einem Suspend (Schlafzustand des gesamten Systems) ausgelöst werden. Er reagiert ebenfalls auf unterschiedliche Taktungen, die beispielsweise durch NTP erfolgen. Dieser Zeitgeber liefert die Sekunden und Nanosekunden seit dem 1.1. 1970 UTC (Unix-Zeit) zurück.

CLOCK_MONOTONIC
: Dieser Zeitgeber läuft entsprechend seiner Auflösung stets vorwärts, ohne dabei Zeitsprünge zu vollziehen. Er ist also unabhängig von der mit Superuser-Privilegien zu verändernden Systemuhr. Allerdings reagiert dieser Zeitgeber auf Modifikationen der Taktung, die beispielsweise durch NTP erfolgen.

CLOCK_MONOTONIC_RAW
: Dieser Zeitgeber ist Linux-spezifisch. Er reagiert weder auf Zeitsprünge noch auf im Betrieb geänderte Taktungen (NTP).

CLOCK_PROCESS_CPUTIME_ID
: Dieser Zeitgeber erfasst die Verarbeitungszeit (Execution Time) des zugehörigen Prozesses. Das funktioniert aber nur zuverlässig auf Singlecore-Systemen beziehungsweise wenn sichergestellt werden kann, dass keine Prozessmigration stattfindet.

CLOCK_THREAD_CPUTIME_ID
: Dieser Zeitgeber erfasst die Verarbeitungszeit (Execution Time) des zugehörigen Threads. Das funktioniert aber nur zuverlässig

auf Singlecore-Systemen beziehungsweise wenn sichergestellt werden kann, dass keine Prozessmigration stattfindet.

Der maximal messbare Zeitbereich schließlich ergibt sich durch die Auflösung des Zeitgebers und die Bitbreite der Variablen:

```
zeitbereich = auflösung * 2
```

Im Folgenden werden zunächst Funktionen vorgestellt, mit denen die aktuelle Zeit bestimmt werden kann. Danach zeigen wir, wie zwei Zeitpunkte miteinander verglichen werden können. Das Rechnen mit Zeiten wird in Abschnitt 4.4.3 diskutiert. Schließlich folgt ein Abschnitt über das gewollte Schlafenlegen beziehungsweise die Implementierung von Timerfunktionen.

4.4.1 Aktuelle Zeit bestimmen

Es gibt unterschiedliche Systemfunktionen, mit denen die aktuelle Zeit gelesen werden kann. Favorisiert ist die Funktion `int clock_gettime(clockid_t clk_id, struct timespec *tp)`, die die Zeit seit dem 1.1.1970 (Unix-Zeit) als *Universal Time* (Zeitzone UTC) zurückliefert (`struct timespec`). Konnte die aktuelle Zeit gelesen werden, gibt die Funktion null, ansonsten einen Fehlercode zurück. Allerdings ist das Auslesen auf 32-Bit-Systemen problematisch, da der 32-Bit-Zähler am 19. Januar 2038 überläuft. Vor allem eingebettete Systeme, bei denen von einer jahrzehntelangen Standzeit ausgegangen wird, könnten von diesem Zählerüberlauf betroffen sein. Wichtig ist, dass sie für diesen Fall korrekt programmiert sind (siehe Abschnitt 4.4.2).

Tabelle 4-2
Funktionen zum Lesen von Zeiten

Funktion	Beschreibung
`time`	Gibt die Anzahl Sekunden zurück, die seit dem 1.1.1970 (UTC) vergangen sind.
`gettimeofday`	Gibt die Anzahl Sekunden und Mikrosekunden zurück, die seit dem 1.1.1970 (UTC) vergangen sind.
`clock_gettime`	Sekunden und Nanosekunden, die seit dem 1.1.1970 (UTC) vergangen sind. Die Genauigkeit kann per `clock_getres()` ausgelesen werden.
`times`	Liefert die aktuelle Zeit als Timerticks (Bezugszeitpunkt ist nicht genau definiert); zusätzlich auch die Verarbeitungszeit, die im Kernel und die im Userland angefallen ist.
TSC	Der TSC ist ein mit der Taktfrequenz der CPU getakteter, sehr genauer Zähler, auf den per Maschinenbefehle zugegriffen werden kann. Vorsicht: Die Taktfrequenz der CPU kann schwanken.

```
struct timepspec timestamp;
...
if (clock_gettime(CLOCK_MONOTONIC,&timestamp)) {
```

```
    perror("timestamp");
    return -1;
}
printf("seconds: %ld, nanoseconds: %ld\n",
    timestamp.tv_sec, timestamp.tv_nsec);
```

Durch die Wahl der Clock `CLOCK_PROCESS_CPUTIME_ID` beziehungsweise `CLOCK_THREAD_CPUTIME_ID` kann auch die Verarbeitungszeit ausgemessen werden (Profiling).

Die Genauigkeit der zurückgelieferten Zeit kann mithilfe der Funktion `clock_getres(clockid_t clk_id, struct timespec *res)` ausgelesen werden.

Beispiel 4-9
Auslesen der Zeitgebergenauigkeit

```
#include <stdio.h>
#include <time.h>

int main( int argc, char **argv, char **envp )
{
   struct timespec ts;

   clock_getres( CLOCK_MONOTONIC, &ts);
   printf("resolution CLOCK_MONOTONIC: %ld sec, %ld nanoseconds\n",
      ts.tv_sec, ts.tv_nsec );
   clock_getres( CLOCK_REALTIME, &ts);
   printf("resolution CLOCK_REALTIME: %ld sec, %ld nanoseconds\n",
      ts.tv_sec, ts.tv_nsec );
   return 0;
}
```

Die Funktion `clock_gettime()` ist nicht in der Standard-C-Bibliothek zu finden, sondern in der Realzeit-Bibliothek `librt`. Daher ist bei der Programmgenerierung diese Bibliothek hinzuzulinken (Parameter `-lrt`). Steht nur die Standard-C-Bibliothek zur Verfügung, kann `time_t time(time_t *t)` oder auch `int gettimeofday(struct timeval *tv, struct timezone *tz)` eingesetzt werden.

`time()` gibt die Sekunden zurück, die seit dem 1.1.1970 (UTC) vergangen sind.

```
time_t now;
...
now = time(NULL);
```

`gettimeofday()` schreibt an die per `tv` übergebene Speicheradresse die Sekunden und Mikrosekunden seit dem 1.1.1970. Das Argument `tz` wird typischerweise mit `NULL` angegeben.

Liegen die Sekunden seit dem 1.1.1970 vor (`timestamp.tv_sec`), können diese mithilfe der Funktionen `struct tm *localtime_r(const time_t`

*timep, struct tm *result); oder struct tm *gmtime_r(const time_t *timep, struct tm *result); in die Struktur struct tm konvertiert werden.

```
struct tm absolute_time;

if (localtime_r( timestamp.tv_sec, &absolute_time )==NULL) {
   perror( "localtime_r" );
   return -1;
}
printf("year: %d\n", absolute_time.tm_year);
```

Die Funktion time_t mktime(struct tm *tm) konvertiert eine über die Struktur struct tm gegebene Zeit in Sekunden seit dem 1.1.1970 (time_t).

Mithilfe der Funktion clock_t times(struct tms *buf) lässt sich sowohl die aktuelle Zeit zu einem letztlich nicht genau definierten Bezugspunkt als auch die Verarbeitungszeit (Execution Time) des aufrufenden Prozesses bestimmen. Die Zeiten werden als Timerticks (clock_t) zurückgeliefert. Die zurückgelieferte Verarbeitungszeit ist aufgeschlüsselt in die Anteile, die im Userland, und die Anteile, die im Kernel verbraucht wurden. Außerdem werden diese Anteile auch für Kindprozesse gelistet.

Mithilfe des Systemcalls sysconf(_SC_CLK_TCK) kann die Anzahl der Timerticks pro Sekunde ausgelesen werden. Der Kehrwert gibt dann die Zeitdauer eines Timerticks an (siehe Beispiel 4-10).

Beispiel 4-10 *Lesen von Verarbeitungszeiten*

```
#include <stdio.h>
#include <sys/times.h>
#include <unistd.h>
#include <time.h>

int main( int argc, char **argv, char **envp )
{
   struct tms exec_time;
   clock_t act_time;
   long ticks_per_second;
   long tickduration_in_ms;

   ticks_per_second = sysconf(_SC_CLK_TCK);
   tickduration_in_ms = 1000/ticks_per_second;

   act_time = times( &exec_time );
   printf("actual time (in ms): %ld\n", act_time*tickduration_in_ms);
   printf("execution time (in ms): %ld\n",
      (exec_time.tms_utime+exec_time.tms_stime)*tickduration_in_ms);
```

```
    return 0;
}
```

Sehr genaue Zeiten lassen sich erfassen, falls der eingesetzte Prozessor einen Zähler besitzt, der mit der Taktfrequenz des Systems getaktet wird. Bei einer x86-Architektur (PC) heißt dieser Zähler *Time-Stamp-Counter* (TSC). Der TSC kann auch von einer normalen Applikation ausgelesen werden, allerdings muss sichergestellt sein, dass sich die Taktfrequenz zwischen zwei Messungen nicht ändert. Alternativ kann man sich vom Betriebssystem über die Taktänderung informieren lassen.

4.4.2 Der Zeitvergleich

Sollen zwei Zeitstempel verglichen werden, die über eine `struct timeval` repräsentiert sind, kann das Makro `int timercmp(struct timeval *a, struct timeval *b, CMP)` eingesetzt werden. Für `CMP` ist der Vergleichsoperator »<«, »>«, »==«, »<=« oder »>=« einzusetzen. Die Linux-Manpage weist darauf hin, dass auf anderen Plattformen die Vergleiche »<=«, »==« und »>=« nicht immer korrekt implementiert sind, und empfiehlt daher, für eine portierbare Realzeitapplikation die negierten Varianten zu verwenden:

```
if (!timercmp( a, b, < )) {
    // a ist identisch b oder später
    ...
}
if (!timercmp( a, b, > )) {
    // a ist früher als b oder gleich
    ...
}
if (!timercmp( a, b, != )) {
    // a ist identisch mit b
    ...
}
```

Liegen die Zeitstempel in Form einer `struct timespec`, also mit Nanosekundenanteil vor, gibt es keine vorgefertigten Makros. Hier könnte alternativ die in Beispiel 4-11 dargestellte Funktion verwendet werden, die »-1« zurückgibt, falls der erste Zeitstempel (`first`) zeitlich vor dem zweiten Zeitstempel (`second`) liegt, »0«, falls beide Zeitstempel identisch sind, und »1«, falls der erste Zeitstempel zeitlich nach dem ersten Zeitstempel liegt.

Beispiel 4-11
Vergleich zweier Zeitstempel vom Typ struct timespec

```
/*********************************************************/
/* return  1: falls "first" zeitlich nach "second" liegt */
/* return  0: falls "first" und "second" identisch sind  */
/* return -1: falls "first" zeitlich vor "second" liegt  */
/*********************************************************/
int timespec_cmp( struct timespec *first, struct timespec *second )
{
    if (first->tv_sec > second->tv_sec)
        return 1; // first later than second (first > second)
    if (first->tv_sec < second->tv_sec)
        return -1; // first earlier than second (first < second)

    if (first->tv_nsec > second->tv_nsec)
        return 1; // first later than second (first > second)

    if (first->tv_nsec < second->tv_nsec)
        return -1; // first earlier than second (first < second)

    return 0; // first == second
}
```

Zwei Absolutzeiten (`struct tm`) werden am einfachsten über deren Repräsentation in Sekunden verglichen. Die Umwandlung erfolgt über die Funktion (`time_t mktime(struct tm *tm)`). Allerdings ist dabei zu beachten, dass es auf einem 32-Bit-System am 19. Januar 2038 zu einem Überlauf kommt. Wird einer der beiden Zeitstempel vor dem 19. Januar 2038 genommen, der andere danach, kommt es zu einem falschen Ergebnis, wenn nur die beiden Werte per »<« beziehungsweise »>« verglichen werden.

Das ist ein generelles Problem und kann dann gelöst werden, wenn sichergestellt ist, dass die zu vergleichenden Zeiten nicht weiter als die Hälfte des gesamten Zeitbereiches auseinanderliegen. In diesem Fall lassen sich die Makros einsetzen, die im Linux-Kernel für den Vergleich zweier Zeiten eingesetzt werden. Das Makro `time_after(a,b)` liefert *true* zurück, falls es sich bei `a` um eine spätere Zeit als `b` handelt. Das Makro `time_after_eq(a,b)` liefert *true* zurück, falls es sich bei `a` um eine spätere Zeit oder um die gleiche Zeit wie `b` handelt. Die Zeitstempel `a` und `b` müssen beide vom Typ `unsigned long` sein. Das wird durch den Compiler auch mithilfe des Schlüsselwortes »typecheck« sichergestellt. Natürlich können die Makros auch auf andere Datentypen angepasst werden (Kernel-Headerdatei [linux/jiffies.h]).

```
#define time_after(a,b)         \
        (typecheck(unsigned long, a) && \
         typecheck(unsigned long, b) && \
         ((long)(b) - (long)(a) < 0))
#define time_before(a,b)        time_after(b,a)

#define time_after_eq(a,b)      \
        (typecheck(unsigned long, a) && \
         typecheck(unsigned long, b) && \
         ((long)(a) - (long)(b) >= 0))
#define time_before_eq(a,b)     time_after_eq(b,a)
```

Beispiel 4-12
Programmtechnische Realisierung von Zeitvergleichen

```
#define time_after(a,b)         \
         ((long)(b) - (long)(a) < 0)
#define time_after_eq(a,b)      \
         ((long)(a) - (long)(b) >= 0)

...
struct timespec start, end;

clock_gettime( CLOCK_REALTIME, &start );
clock_gettime( CLOCK_REALTIME, &end );

if (time_after(start.tv_nsec, end.tv_nsec) ) {
   printf("%ld (start) is later than %ld (end)\n",
      start.tv_nsec, end.tv_nsec );
} else if (time_after_eq(start.tv_nsec, end.tv_nsec) ) {
   printf("%ld (start) is equal to %ld (end)\n",
      start.tv_nsec, end.tv_nsec );
} else {
   printf("%ld (start) is earlier than %ld (end)\n",
      start.tv_nsec, end.tv_nsec );
}
```

4.4.3 Differenzzeitmessung

In Realzeitapplikationen ist es häufig notwendig, eine Zeitdauer zu messen, Zeitpunkte zu erfassen oder eine definierte Zeit verstreichen zu lassen. Dabei sind folgende Aspekte zu beachten:

- Die Genauigkeit der eingesetzten Zeitgeber,
- die maximalen Zeitdifferenzen,
- Schwankungen der Zeitbasis, beispielsweise durch Schlafzustände,
- Modifikationen an der Zeitbasis des eingesetzten Rechnersystems (Zeitsprünge) und
- die Ortsabhängigkeit absoluter Zeitpunkte.

Zur Bestimmung einer Zeitdauer verwendet man häufig eine Differenzzeitmessung. Dabei wird vor und nach der auszumessenden Aktion jeweils ein Zeitstempel genommen. Die Zeitdauer ergibt sich aus der Differenz dieser beiden Zeitstempel.

Dies ist allerdings nicht immer ganz einfach. Liegt der Zeitstempel beispielsweise in Form der Datenstruktur `struct timespec` (als Ergebnis der Funktion `clock_gettime()`) vor, werden die Sekunden zunächst getrennt von den Nanosekunden subtrahiert. Ist der Nanosekundenanteil negativ, muss der Sekundenanteil um eins erniedrigt und der Nanosekundenanteil korrigiert werden. Dazu wird zu der Anzahl der Nanosekunden pro Sekunde (also eine Milliarde) der negative Nanosekundenanteil addiert (siehe Beispiel 4-13).

Beispiel 4-13 *Quellcodebeispiel zur Differenzzeitmessung*

```
#define NANOSECONDS_PER_SECOND 1000000000

struct timespec * diff_time( struct timespec before, struct timespec after,
    struct timespec *result )
{
    if (result==NULL)
        return NULL;

    if ((after.tv_sec<before.tv_sec) ||
        ((after.tv_sec==before.tv_sec) &&
        (after.tv_nsec<=before.tv_nsec))) { /* after before before */
        result.tv_sec = result.tv_nsec = 0;
    }
    result->tv_sec = after.tv_sec - before.tv_sec;
    result->tv_nsec= after.tv_nsec- before.tv_nsec;

    if (result->tv_nsec<0) {
        result->tv_sec--;
        /* result->tv_nsec is negative, therefore we use "+" */
        result->tv_nsec = NANOSECONDS_PER_SECOND+result->tv_nsec;
    }
    return result;
}
```

Für Zeitstempel vom Typ `struct timeval` kann der Code leicht umgeschrieben werden. Anstelle der `NANOSECONDS_PER_SECOND` sind `MICROSECONDS_PER_SECOND` einzusetzen. Sind die Zeitstempel vorzeichenlos, sieht die Rechnung für den Sekundenanteil etwas komplizierter aus, soll hier aber nicht weiter erläutert werden.

Für Zeitstempel vom Typ `struct timeval` steht zudem noch die Funktion `void timersub(struct timeval *a, struct timeval *b, struct timeval *res)` zur Verfügung. Diese legt die Differenz in normierter Form in `res` ab.

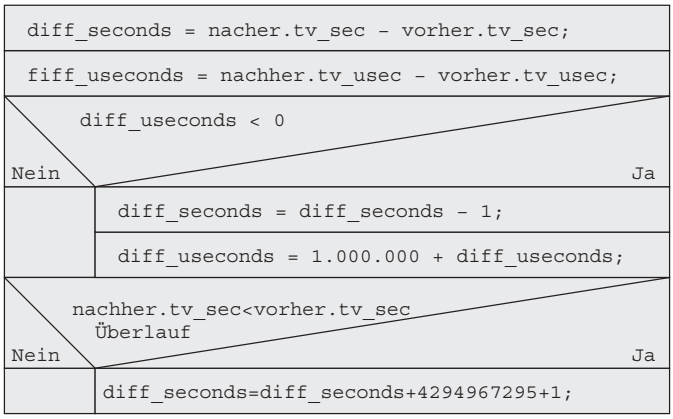


Abbildung 4-6 *Struktogramm Differenzzeitmessung (32-Bit-System)*

Einen Überlauf fängt Beispiel 4-13 übrigens durch eine Abfrage, ob der spätere Zeitstempel kleiner ist als der frühere Zeitstempel, direkt zu Beginn des Beispiels ab. Die Funktion returniert in diesem Fall »0«. Alternativ kann aber auch der Überlauf berücksichtigt werden. Dann ergibt sich der in Abbildung 4-6 in Form eines Struktogramms (siehe Abschnitt 7.2) dargestellte Ablauf, der für ein 32-Bit-System gültig ist.

Etwas einfacher ist die Differenzbildung, wenn aus der Datenstruktur eine einzelne Variable mit der gewünschten Auflösung, beispielsweise Mikrosekunden, generiert wird. Im Fall von `struct timeval` wird dazu der Sekundenanteil mit einer Million multipliziert und der Mikrosekundenanteil aufaddiert. Bei der Multiplikation können natürlich Informationen verloren gehen, allerdings geht der gleiche Informationsgehalt auch beim zweiten Zeitstempel verloren. Für die Differenzbildung ist dieser Umstand nicht relevant, solange der zu messende zeitliche Abstand kleiner als 1000 Sekunden ist und es während der Messung keinen Überlauf beim Sekundenanteil gibt.

```
/*Zeitstempel liegen als struct timeval vor */
time_in_usec=((nachher.tv_sec*1000000)+nachher.tv_usec)-
    ((vorher.tv_sec*1000000)+vorher.tv_usec);

/* Zeitstempel liegen als struct timespec vor */
time_in_usec=((nachher.tv_sec*1000000)+(nachher.tv_nsec/1000))-
    ((vorher.tv_sec*1000000)+(vorher.tv_usec/1000));
```

```
static int difference_micro(struct timeval *before,
    struct timeval *after)
{
    return (signed long long) after->time.tv_sec * 1000000ll +
        (signed long long) after->time.tv_usec -
        (signed long long) before->time.tv_sec * 1000000ll -
   (signed long long) before->time.tv_usec;
}
```

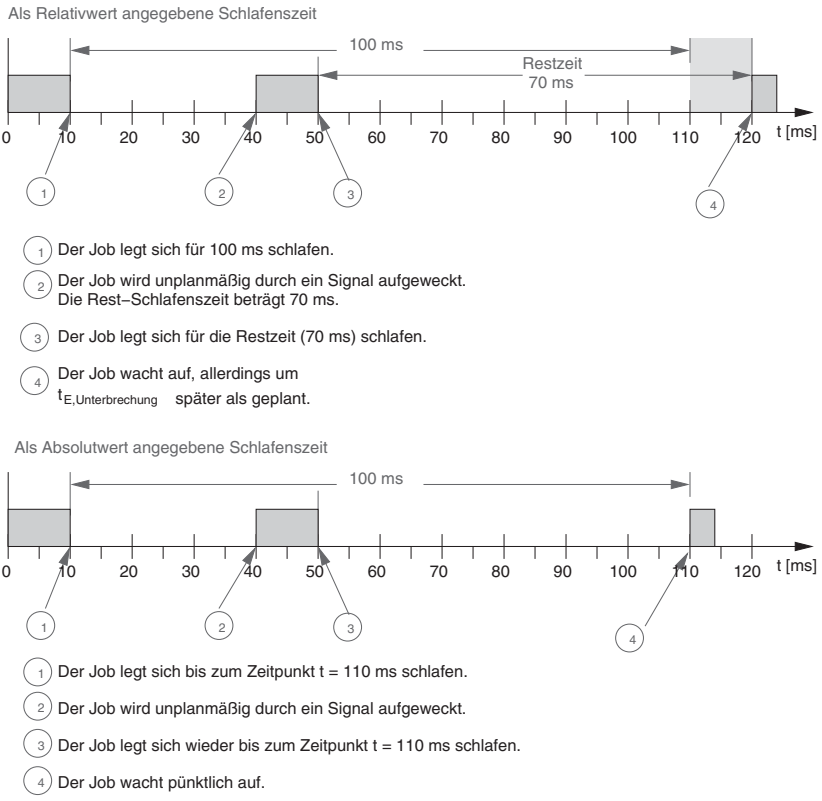
4.4.4 Schlafen

Threads legen sich für eine Zeitspanne oder aber bis zu einem Punkt, an dem sie aufgeweckt werden, schlafen (absolutes oder relatives Schlafen). Innerhalb von Realzeitapplikationen kann zusätzlich die zu verwendende Zeitquelle (`CLOCK_MONOTONIC` oder `CLOCK_REALTIME`) definiert werden. Beachten Sie, dass negative Zeitangaben beim Schlafenlegen typischerweise nicht definiert sind.

Tabelle 4-3 *Funktionen zum Schlafenlegen von Jobs*

Funktion	Beschreibung
`clock_nanosleep`	Auflösung Nanosekunden, Auswahl des Zeitgebers, Auswahl absolut/relativ.
`nanosleep`	Nanosekunden
`sleep`	Auflösung in Sekunden
`usleep`	Mikrosekunden

Abbildung 4-7 *Vorteile beim Schlafen mit absoluter Zeitangabe*



Ob relativ oder absolut geschlafen wird, hat durchaus Relevanz: Wird das Schlafen unterbrochen und danach mit der Restzeit neu aufgesetzt, kommt es durch den zusätzlichen Aufruf zu einer Verzögerung. Bei der

Verwendung einer absoluten Weckzeit fallen diese zusätzlichen Aufrufe zeitlich nicht ins Gewicht (Abbildung 4-7).

Ein Thread kann sich durch Aufruf der Funktion `int nanosleep(const struct timespec *req, struct timespec *rem);` für die über `req` definierte Zeitspanne schlafen legen. Hierbei wird zwar offiziell die Zeitquelle `CLOCK_REALTIME` verwendet, eine Änderung der Schlafenszeit wird aber nachgeführt. De facto basiert `nanosleep()` daher auf `CLOCK_MONOTONIC`. Der schlafende Thread wird aufgeweckt, wenn entweder die angegebene Relativzeit abgelaufen ist oder der Thread ein Signal gesendet bekommen hat. Ist Letzteres der Fall gewesen und hat der Aufrufer den Parameter `rem` mit einer gültigen Hauptspeicheradresse versehen, legt das Betriebssystem in diesem Speicher die noch übrig gebliebene Schlafenszeit ab.

Beispiel 4-14
Schlafenlegen per nanosleep()

```
...
struct timespec req, rem;
int error;
...
req.tv_sec  = 60;
req.tv_nsec = 1000;
while ((error=nanosleep(&req,&rem))==-1) {
    if (errno==EINTR) {
        req = rem;
    } else {
        perror("nanosleep");
        break;
    }
}
```

Genauer kann das Verhalten beim Schlafenlegen über die Funktion `int clock_nanosleep(clockid_t clock_id, int flags, const struct timespec *request, struct timespec *remain);` eingestellt werden. Über den Parameter `clock_id` wird die Zeitquelle (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`) eingestellt. Der Parameter `flag` erlaubt die Angabe, ob die Zeitangabe `request` relativ (`flag==0`) oder absolut (`flag==TIMER_ABSTIME`) zu interpretieren ist. Um die Restzeit bei einem durch ein Signal provozierten vorzeitigen Abbruch aufzunehmen, kann per `remain` eine Speicheradresse dafür übergeben werden. `request` schließlich enthält die Zeitangabe, bis zu der (`TIMER_ABSTIME`) oder die der Job schlafen soll. Bei Angabe der absoluten Zeitangabe ist auf die Normierung zu achten: Wird der Nanosekundenanteil größer oder gleich eine Milliarde, repräsentiert dieser Anteil also eine Sekunde oder mehr, ist mithilfe der Division und der Modulo-Operation eine Anpassung notwendig:

```
if (sleeptime.tv_nsec>999999999) {
    sleeptime.tv_sec += sleeptime.tv_nsec/1000000000;
    sleeptime.tv_nsec = sleeptime.tv_nsec%1000000000;
}
```

Die professionelle Programmierung mit POSIX-Funktionen zum Schlafenlegen eines Job – Beispiel 4-15, Beispiel 4-16 – zeigt im Übrigen die legere Variante des Schlafens mit relativer Zeitangabe.

Beispiel 4-15
Schlafenlegen einer Task mit absoluter Zeitangabe

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

void sigint_handler(int signum)
{
   printf("SIGINT (%d)\n", signum);
}

int main( int argc, char **argv, char **envp )
{
   struct timespec sleeptime;
   struct sigaction sa;

   sigemptyset(&sa.sa_mask);
   sa.sa_flags = 0;
   sa.sa_handler = sigint_handler;
   if (sigaction(SIGINT, &sa, NULL) == -1) {
      perror( "sigaction" );
      return -1;
   }
   printf("%d sleeps for 10 seconds and 1 millisecond...\n", getpid());
   clock_gettime( CLOCK_MONOTONIC, &sleeptime );
   sleeptime.tv_sec  += 10;
   sleeptime.tv_nsec += 1000000;
   if (sleeptime.tv_nsec>999999999) {
      sleeptime.tv_sec += sleeptime.tv_nsec/1000000000;
      sleeptime.tv_nsec = sleeptime.tv_nsec%1000000000;
   }
   while (clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME,
      &sleeptime,NULL)==EINTR) {
      printf("interrupted...\n");
   }
   printf("woke up...\n");
   return 0;
}
```

```
struct timespec sleeptime;
...
sleeptime.tv_sec  = 0;
sleeptime.tv_nsec = 250000000; /* 250 Millisekunden */
if ((error=clock_nanosleep(CLOCK_MONOTONIC,0,&sleeptime,NULL))!=0 ) {
    printf("clock_nanosleep reporting error %d\n", error);
}
```

Beispiel 4-16
Schlafenlegen einer Task mit relativer Zeitangabe

Die Funktion `clock_nanosleep()` steht nur über die Realzeitbibliothek `librt` zur Verfügung. Beim Linken ist daher die Option `-lrt` mit anzugeben.

Neben `nanosleep()` finden sich in der Standard-C-Bibliothek noch weitere Funktionen, mit denen Threads schlafen gelegt werden können: `int usleep(useconds_t usec)`, `unsigned int sleep(unsigned int seconds)` und `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)`.

Die Funktion `sleep()` bekommt die Schlafenszeit als eine Anzahl von Sekunden übergeben, bei `usleep()` sind es Mikrosekunden. `select()` ist eigentlich nicht primär dazu gedacht, Jobs schlafen zu legen. Diese Eigenschaft wurde insbesondere früher genutzt, um eine leicht portierbare Funktion zum Schlafenlegen bei Auflösung im Mikrosekundenbereich zu haben.

4.4.5 Weckrufe per Timer

Das Betriebssystem kann periodisch Funktionen, sogenannte Timer, aufrufen. Diese Funktionen werden typischerweise als *Signal-Handler* oder im Rahmen eines Threads aktiviert.

Dazu müssen zwei Datenstrukturen vorbereitet werden. `struct sigevent` speichert die Daten, die mit dem Timer selbst und mit der aufzurufenden Funktion zusammenhängen:

```
struct sigevent {
    int  sigev_notify;
    int  sigev_signo;
    union sigval sigev_value;
    void (*sigev_notify_function) (union sigval);
    void *sigev_notify_attributes;
    pid_t sigev_notify_thread_id;
};
```

Das Element `sigev_notify` legt fest, ob die Timerfunktion als Signal-Handler oder im Rahmen eines Threads aktiviert wird. Interessant sind die Werte `SIGEV_SIGNAL` und `SIGEV_THREAD`. Im Fall von `SIGEV_SIGNAL` legt `sigev_signo` die Nummer des Signals fest, welches nach Ablauf des Auslöseintervalls dem Job gesendet wird. Das Feld `sigev_value` nimmt einen

Parameter auf, der entweder dem Signal-Handler oder der im Thread abgearbeiteten Funktion übergeben wird. Der Signal-Handler selbst wird mit der Funktion `sigaction()` (siehe Abschnitt 4.7) etabliert. Unter Linux kann auch noch der Thread spezifiziert werden, dem das Signal zuzustellen ist. Dazu ist der Parameter `sigev_notify_thread_id` mit der Thread-ID zu besetzen und anstelle von `SIGEV_SIGNAL` ist `SIGEV_THREAD_ID` für `sigev_notify` auszuwählen.

Ist für `sigev_notifySIGEV_THREAD` ausgewählt, nimmt `sigev_notify_function` die Adresse der Funktion auf, die im Kontext eines Threads abgearbeitet wird. Per `sigev_notify_attributes` lässt sich die Thread-Erzeugung konfigurieren. Meistens reicht es aus, hier `NULL` zu übergeben.

Die Datenstruktur `struct sigevent` wird per `int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid)` dem Kernel übergeben. `clockid` spezifiziert den Zeitgeber (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`), `evp` die initialisierte `struct sigevent` und in `timerid` findet sich nach dem Aufruf die Kennung des erzeugten, aber deaktivierten Timers.

Der beziehungsweise die Zeitpunkte, zu denen die Timerfunktion aufgerufen werden soll, wird über `struct itimerspec` konfiguriert:

```
struct itimerspec {
    struct timespec it_interval;  /* Timer interval */
    struct timespec it_value;     /* Initial expiration */
};
```

`it_value` gibt in Sekunden und Nanosekunden die Zeit an, zu der erstmalig die Funktion oder der Signal-Handler aufgerufen werden soll, `it_interval` spezifiziert in Sekunden und Nanosekunden die Periode. Ob die Zeiten absolut oder relativ gesehen werden, wird beim Aufruf der Funktion `int timer_settime(timer_t timerid, int flags, const struct itimerspec *new_value, struct itimerspec * old_value)` über den Parameter `flags` (0 oder `TIMER_ABSTIME` festgelegt. `new_value` übernimmt die zeitliche Parametrierung. Falls `old_value` ungleich `NULL` ist, findet sich in dieser Datenstruktur nach dem Aufruf das vorhergehende Intervall.

Die Funktionen sind nur nutzbar, wenn die Realzeitbibliothek `librt` zur Applikation gebunden wird. Dazu ist beim Aufruf des Compilers die Option `-lrt` mit anzugeben.

Beispiel 4-17 *Periodische Taskaktivierung*

```
#include <stdio.h>
#include <time.h>
#include <signal.h>
#include <unistd.h>

void timer_function( union sigval parameter )
{
   printf("timer with id %d active\n", getpid());
   return;
```

```
}

int main( int argc, char **argv, char **envp )
{
   timer_t itimer;
   struct sigevent sev;
   struct itimerspec interval;

   printf("start process.\n");
   sev.sigev_notify            = SIGEV_THREAD;
   sev.sigev_notify_function   = timer_function;
   sev.sigev_value.sival_int   = 99;
   sev.sigev_notify_attributes = NULL;
   if (timer_create(CLOCK_MONOTONIC, &sev, &itimer ) == -1) {
      perror("timer_create");
      return -1;
   }
   interval.it_interval.tv_sec = 1;
   interval.it_interval.tv_nsec= 0;
   interval.it_value.tv_sec    = 1;
   interval.it_value.tv_nsec   = 0;
   timer_settime( itimer, 0, &interval, NULL ); /* activate timer */
   sleep( 5 );
   printf("end process.\n");
   return 0;
}
```

4.5 Inter-Prozess-Kommunikation

Die klassische Inter-Prozess-Kommunikation (Datenaustausch) bietet unter anderem die folgenden Methoden an:

- Pipes/Mailboxes/Messages
- Shared-Memory
- Sockets

Diese werden im Folgenden vorgestellt.

4.5.1 Pipes, Mailbox und Messages

Zum Datenaustausch bieten Betriebssysteme einen Mailbox-Mechanismus (`send`/`receive`-Interface) an. Dabei werden – im Regelfall unidirektional – Daten von Task 1 zu Task 2 transportiert und gequeued (im Gegensatz zu gepuffert; Daten gehen also nicht verloren, da sie nicht im Puffer überschrieben werden).

In vielen Realzeitbetriebssystemen und in Unix-Systemen ist ein Mailbox-Mechanismus über Pipes (FIFO) und über die sogenannten Messages implementiert.

Pipes stellen dabei eine der einfachsten Möglichkeiten zur IPC dar. Über den Systemcall `int pipe(int piped[2])` werden zwei Deskriptoren reserviert. Über `piped[1]` können per `write()` Daten geschrieben, über `piped[0]` per `read()` gelesen werden. Typischerweise reserviert eine Task zunächst per `pipe()` die Pipe-Deskriptoren, um danach per `fork()` oder `pthread_create()` eine neue Task zu starten, die die beiden Deskriptoren erbt. Da die Kommunikation über Pipes unidirektional ist, gibt jede der Tasks den Deskriptor wieder frei, der nicht benötigt wird.

Beispiel 4-18 *Inter-Process-Communication über Pipes*

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv, char **envp)
{
   int pd[2]; /* pipe descriptor */
   pid_t cpid;
   char c;

   if (pipe(pd) == -1) {
      perror("pipe");
      return -1;
   }
   if ((cpid=fork()) == -1) {
      perror("fork");
      return -1;
   }

   if (cpid == 0) {    /* child reads from pipe */
      close(pd[1]);  /* close unused write end */

      while (read(pd[0], &c, 1) > 0)
         write(STDOUT_FILENO, &c, 1);

      write(STDOUT_FILENO, "\n", 1);
      close(pd[0]);
      return 0;

   } else {            /* Parent writes argv[1] to pipe */
      close(pd[0]);          /* Close unused read end */
      write(pd[1], "hello world", strlen("hello world"));
      close(pd[1]);          /* Reader will see EOF */
      wait(NULL);                /* Wait for child */
      return 0;
```

```
   }
}
```

Messages gehören zur Gruppe der klassischen System-V-IPC. Die klassische System-V-IPC ist allerdings nicht für den Einsatz in Realzeitapplikationen zu empfehlen: Zu kompliziert und vor allem fehleranfällig sind das Anlegen, die Verteilung von Zugriffsberechtigungen und das spätere Freigeben der angelegten Ressource. Diese Methode wird daher auch nicht weiter beschrieben. Alternativ stellt POSIX aber ein Message-Interface zur Verfügung.

Beispiel 4-19
Auf POSIX Messages basierter Server (mq_server.c)

```
#include <stdio.h>
#include <string.h>
#include <mqueue.h>

int main(int argc, char **argv, char **envp)
{
   mqd_t mqdes;
   char buf[256];
   unsigned int msg_prio;
   ssize_t nr;
   struct mq_attr attr;

   attr.mq_flags=0;
   attr.mq_maxmsg = 10;
   attr.mq_msgsize = sizeof(buf);
   attr.mq_curmsgs = 0;
   mqdes = mq_open("/hello", O_RDONLY|O_CREAT, 0644, &attr);
   if (mqdes == (mqd_t) -1) {
      perror("mq_open");
      return -1;
   }
   while ( 1 ) {
      nr = mq_receive(mqdes, buf, sizeof(buf), &msg_prio);
      if (nr == -1) {
         perror("mq_receive");
         mq_unlink( "/hello" );
         return -1;
      }
      buf[nr] = '\0';
      printf("read %ld bytes, prio %d: \"%s\"\n",
         (long) nr, msg_prio, buf );
      if (strncmp(buf,"end",3)==0 )
         break;
   }
   mq_unlink( "/hello" );
   return 0;
}
```

Das POSIX-Message-Interface ist dem Interface für den Peripheriezugriff angelehnt, das sowohl den synchronen als auch den asynchronen Nachrichtenaustausch ermöglicht. Die zugehörigen Funktionen sind in Tabelle 4-4 zu finden.

Tabelle 4-4 *POSIX-Funktionen für Messages*

Funktion	Beschreibung
`mq_open()`	Mit dieser Funktion kann eine Message-Queue angelegt oder der Zugriff auf eine bereits angelegte Message-Queue erlangt werden. Die Parameter entsprechen weitestgehend denen von `open()`. Der Name der Message-Queue muss mit einem Slash beginnen und darf keine weiteren Slashes enthalten. Über eine Struktur `struct mq_attr` werden die Queue-Eigenschaften (beispielsweise die maximale Länge der Message) festgelegt.
`mq_close()`	Diese Funktion gibt den Message-Queue-Deskriptor wieder frei.
`mq_send()`, `mq_timedsend()`	Ähnlich der Funktion `write()` werden Daten über die Message-Queue versendet. Dabei kann sowohl eine Priorität als auch ein Timeout angegeben werden.
`mq_receive()`, `mq_timedreceive()`	Funktion zum Empfangen von Daten aus einer Message-Queue. Achtung: Die Größe des Empfangspuffers muss mindestens der maximalen Message-Länge entsprechen!
`mq_notify()`	Diese Funktion ermöglicht das asynchrone Empfangen von Daten. Ein Codebeispiel hierzu findet sich in den Linux-Manpages, wenngleich dort ein in Realzeitapplikationen zu vermeidendes `malloc()` verwendet wird.
`mq_getattr()`, `mq_setattr()`	Diese Funktionen ermöglichen das Auslesen und Verändern der Queue-Attribute, beispielsweise der Zugriffsart (blockierend, nicht blockierend).
`mq_unlink()`	Hiermit wird die Message-Queue aus dem System entfernt.

Beispiel 4-19 und Beispiel 4-20 zeigen die Verwendung von POSIX Messages. Starten Sie zuerst den Server, danach den Client. Der Client erwartet Eingaben auf der Tastatur, die er als Message dem Server zustellt. Dieser gibt die Nachrichten auf dem Bildschirm aus. Die Nachricht »end« beendet den Server. Der Client kann per STRG C abgebrochen werden.

Beispiel 4-20 *Auf POSIX Messages basierter Client (mq_client.c)*

```
#include <stdio.h>
#include <string.h>
#include <mqueue.h>

int main(int argc, char **argv, char **envp)
{
   mqd_t mqdes;
   char buf[256];
   unsigned int msg_prio=2;
   ssize_t nr;
```

```
   mqdes = mq_open("/hello", O_WRONLY);
   if (mqdes == (mqd_t) -1) {
      perror("mq_open");
      return -1;
   }
   while ( 1 ) {
      fgets( buf, sizeof(buf), stdin );
      nr = mq_send(mqdes, buf, strlen(buf), msg_prio);
      if (nr == -1) {
         perror("mq_send");
         return -1;
      }
   }
   return 0;
}
```

Bei der Programmierung von POSIX Messages ist zu beachten, dass der Name der Message-Queue mit einem Slash beginnen muss, dann aber keine weiteren Slashes enthalten darf. Der Empfangspuffer muss mindestens die Länge haben, die beim Erzeugen der Queue in den Attributen mit der maximalen Nachrichtenlänge spezifiziert wurde. Die Prototypen der Funktionen und Datenstrukturen sind in der Headerdatei `mqueue.h` zu finden, die Funktionen selbst in der Realzeitbibliothek `librt`. Beim Linken ist der Parameter »-lrt« mit anzugeben:

```
quade@felicia:/tmp>LDLIBS=-lrt make mq_server
cc     mq_server.c  -lrt -o mq_server
```

Linux stellt über das virtuelle Dateisystem `mqueue` Informationen zur gerade verwendeten Message-Queue zur Verfügung. Dazu ist ein Verzeichnis anzulegen und das Filesystem einzubinden:

```
root@felicia:/# mkdir /dev/mqueue
root@felicia:/# mount -t mqueue none /dev/mqueue/
```

4.5.2 Shared-Memory

Bei einem Shared-Memory handelt es sich um einen gemeinsamen Speicherbereich einer oder mehrerer Tasks innerhalb eines Rechners. Als Inter-Prozess-Mechanismus hat ein Shared-Memory im Vergleich zu Mailboxes typischerweise Geschwindigkeitsvorteile. Die Realisierung eines gemeinsamen Speicherbereichs im Fall von Threads auf Basis globaler Variablen ist trivial. Wollen mehrere Rechenprozesse jedoch einen gemeinsamen Speicher nutzen, müssen sie diesen vom Betriebssystem anfordern. Ähnlich wie beim Dualport-RAM, bei dem allerdings unterschiedliche Prozessoren zugreifen, kann die Speicheradresse des gemeinsamen Speicherbereichs von Task zu Task unterschiedlich sein. Deshalb

ist es auch hier wichtig, Pointer innerhalb von Datenstrukturen relativ zum Anfang (oder Ende) des Speicherbereichs anzugeben.

Aus den gleichen Gründen, die für die System-V-Messages gelten, ist auch das System-V-Shared-Memory zwischen Rechenprozessen zu vermeiden. Die Verwaltung mit Anlegen der Ressource, die Identifikation der Ressource, die Verwaltung von Zugriffsberechtigungen und schließlich die Freigabe sind im Detail ausgesprochen komplex. Außerdem ist unter normalen Umständen nur eine Kommunikation lokal (innerhalb eines Rechnersystems) möglich.

POSIX bietet ebenfalls Funktionen zum Anlegen von gemeinsamen Speicherbereichen an, die in ihrer Handhabung durch die symbolische Namensgebung deutlich einfacher sind. Bezüglich des Namens gilt die gleiche Einschränkung wie bei den POSIX-Message-Queues. Der Name beginnt mit einem Slash, darf maximal 255 Zeichen lang sein und keinen weiteren Slash beinhalten. Die reservierten Shared-Memory-Bereiche sind kernelpersistent, bleiben also bis zum Aufruf der Funktion `shm_unlink()` im System.

Beispiel 4-21 *Anlegen eines POSIX-Shared-Memory (shm.c)*

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main( int argc, char **argv, char **envp )
{
   int sd;
   char *shm_adr;

   sd = shm_open( "/foo", O_RDWR, 0755 );
   if (sd < 0) {
      perror("shm_open");
      return -1;
   }
   ftruncate( sd, 1024 );

   shm_adr=(char *)mmap(NULL,1024,PROT_READ|PROT_WRITE,MAP_SHARED,sd,0);
   if (shm_adr==NULL) {
      perror( "mmap" );
      shm_unlink( "/foo" );
      return -1;
   }
   printf("Shared Mem on adr %p\n", shm_adr );
   printf("Found: \"%s\"\n", shm_adr );

   return 0;
}
```

Beispiel 4-22 *Zugriff auf ein POSIX-Shared-Memory (shm_reader.c)*

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main( int argc, char **argv, char **envp )
{
   int sd;
   char *shm_adr;

   sd = shm_open( "/foo", O_RDWR, 0755 );
   if (sd < 0) {
      perror("shm_open");
      return -1;
   }
   ftruncate( sd, 1024 );

   shm_adr=(char *)mmap(NULL,1024,PROT_READ|PROT_WRITE,MAP_SHARED,sd,0);
   if (shm_adr==NULL) {
      perror( "mmap" );
      shm_unlink( "/foo" );
      return -1;
   }
   printf("Shared Mem on adr %p\n", shm_adr );
   printf("Found: \"%s\"\n", shm_adr );

   return 0;
}
```

Beispiel 4-21, und Beispiel 4-22 zeigen den Einsatz der POSIX-Shared-Memory. Beim Linken wird wieder die Option `-lrt` benötigt. Als Erstes ist das Programm `shm`, danach `shm_reader` zu starten.

```
quade@felicia:~>LDLIBS=-lrt make shm shm_reader
cc -g -Wall    shm.c  -lrt -lpthread -o shm
cc -g -Wall    shm_reader.c  -lrt -lpthread -o shm_reader
quade@felicia:~>./shm
Shared Mem on adr 0x7f55bc3e2000
quade@felicia:~>./shm_reader
Shared Mem on adr 0x7f86ca198000
Found: "Hello World"
quade@felicia:~>
```

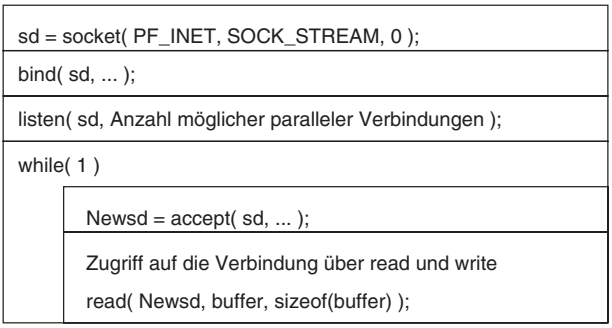
Unabhängig von der Realisierung als POSIX-Shared-Memory oder auf Basis von Threads: Jede globale Variable stellt einen potenziellen kritischen Abschnitt dar und ist gegebenenfalls zu schützen!

4.5.3 Sockets

Die wichtigste Schnittstelle für Inter-Prozess-Kommunikation stellt zurzeit die Socket-Schnittstelle dar. Mittels Sockets können Daten zwischen Prozessen ausgetauscht werden, die auf unterschiedlichen Rechnern lokalisiert sind (verteiltes System).

Die Socket-Schnittstelle bietet Zugriff zur TCP/IP- und zur UDP-Kommunikation. Ist dabei einmal eine Verbindung zwischen zwei Prozessen hergestellt worden, können die Daten mit den bekannten Systemcalls (`read()`, `write()` ...) ausgetauscht werden.

Abbildung 4-8
Basisstruktur einer Socket-Serverapplikation



Auf TCP/IP-Ebene handelt es sich um eine Server-Client-Kommunikation. Ein Server alloziert per `socket`-Funktion einen beliebigen Socket (Ressource) im System, wobei der Socket über eine Nummer identifiziert wird. Damit ein Client auf einen Socket zugreifen kann, muss er die Socketnummer (Socketadresse) kennen. Für Standarddienste, wie beispielsweise http, sind diese Nummern reserviert, man spricht von sogenannten *well known sockets*.

Hat der Server einen Socket alloziert, muss er ihm eine bekannte Socketnummer zuweisen (z.B. 80 bei http). Dies geschieht mit der Funktion `bind()` (siehe Abbildung 4-8). Über einen Socket kann der Server gleichzeitig mehrere Clients bedienen. Über die Funktion `listen()` parametriert er dabei, wie viele derartige Clients er parallel bedienen möchte (Beispiel 4-23). Ein einzelner Port auf einem Rechner kann aus dem Grund mehrere Verbindungen (Clients) bedienen (und unterscheiden), weil eine Verbindung durch

- Ziel-IP-Adresse,
- Ziel-Port,
- Remote-IP-Adresse und
- Remote-Port

charakterisiert ist.

Ist der Socket geöffnet und ist ihm eine bekannte Socketadresse zugewiesen worden, kann der Server auf Verbindungswünsche warten. Dieses Warten wird über den `accept()`-Aufruf realisiert. Accept liefert, wenn auf dem bekannten Socket ein Verbindungswunsch kommt, einen *neuen* Socket zurück. Der ursprüngliche, gebundene Socket (bind socket) ist damit wieder frei, um auf weitere Verbindungswünsche zu warten. Klassische Serverprogramme, wie beispielsweise der http-Server, erzeugen für jeden Verbindungswunsch (also nach dem `accept()`) mittels `fork()` einen eigenen Prozess. Der zurückgegebene Socket (im Bild `ConnectionSocket`) wird vom Server wie ein Filedeskriptor verwendet. Er kann also auf den Filedeskriptor/Socketdeskriptor schreiben und von ihm lesen (bidirektionale Verbindung).

Beispiel 4-23
Socket-Serverprogramm

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

static int bind_socket, connection_socket;

static void emergency_close( int Signal )
{
   close( bind_socket );
   close( connection_socket );
   exit( -1 );
}

int main( int argc, char **argv, char **envp )
{
   unsigned int count;
   struct sockaddr_in my_port, remote_socket;
   char buffer[512];

   signal( SIGINT, emergency_close );
   bind_socket = socket( AF_INET, SOCK_STREAM, 0 );
   if (bind_socket <= 0) {
      perror( "socket" );
      return -1;
   }
   bzero( &my_port, sizeof(my_port) );
   my_port.sin_port = htons( 12345 );
   if (bind(bind_socket,(struct sockaddr *)&my_port,sizeof(my_port))<0) {
      perror( "bind" );
      return -1;
   }
```

```
   listen( bind_socket, 3 );
   while (1) {
      count = sizeof( remote_socket );
      connection_socket = accept( bind_socket,
            (struct sockaddr *)&remote_socket, &count );
      if (connection_socket >= 0) {
         printf("connection established ...\n");
         while ((count=read(connection_socket,buffer,
            sizeof(buffer)))) {
            if (strncmp(buffer,"end",strlen("end")) == 0) {
               break;
            } else {
               write( connection_socket,buffer,count );
            }
         }
         close( connection_socket );
         printf( "connection released ...\n");
      }
   }
   close( bind_socket );
   return 0;
}
```

Abbildung 4-9
Basisstruktur einer Socket-Clientapplikation

sd = socket(PF_INET, SOCK_STREAM, 0);
inet_aton("192.168.12.12", &destination.sin_addr); //Zieladr. spez.
destination.sin_port = htons(Portnummer); //Zielport spezifizieren
destination.sin_family = PF_INET;
connect(sd, &destination, sizeof(destination));
Zugriff auf die Verbindung über read und write write(sd, Nachricht, strlen(Nachricht)+1);
…

Der Verbindungsaufbau über Sockets auf der Clientseite ist nicht so kompliziert (siehe Abbildung 4-9). Der Client alloziert sich – wie der Server auch – per `socket()`-Funktion die Socket-Ressource. Da der Client keine Verbindung entgegennehmen möchte, muss dieser Socket auch nicht an eine bestimmte Nummer gebunden werden. Der Client tätigt nur noch einen `connect()`-Aufruf, um sich mit dem Remote-Rechner zu verbinden. Er verwendet den vom Socket-Aufruf returnierten Socketdeskriptor/Filedeskriptor direkt (Beispiel 4-24).

Beispiel 4-24
Socket-Clientprogramm

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define TEXT "Hello, here is a client ;-)"

static int sd;

static void emergency_close( int signal )
{
   close( sd );
}

int main( int argc, char **argv, char **envp )
{
   int count;
   struct sockaddr_in destination;
   char buffer[512];

   signal( SIGINT, emergency_close );
   sd = socket( PF_INET, SOCK_STREAM, 0 );
   if (sd <= 0) {
      perror( "socket" );
      return -1;
   }
   bzero( &destination, sizeof(destination) );
   //inet_aton( "194.94.121.156", &destination.sin_addr );
   //inet_aton( "127.0.0.1", &destination.sin_addr );
   inet_aton( "192.168.69.39", &destination.sin_addr );
   destination.sin_port = htons( 12345 );
   destination.sin_family = PF_INET;
   if (connect(sd,(struct sockaddr *)&destination,sizeof(destination))<0) {
      perror( "connect" );
      return -1;
   }
   write( sd, TEXT, strlen(TEXT)+1 );
   if ((count=read( sd, buffer, sizeof(buffer))) <=0) {
      perror( "read" );
      close( sd );
      return -1;
   }
   printf( "%d: %s\n", count, buffer );
   write( sd, "end", 4 );
   close( sd );
```

```
    return 0;
}
```

Die Kommunikation über Sockets ist – wie in den Abbildungen ersichtlich – relativ unproblematisch. Reale Serverprogramme werden aber aufgrund notwendiger Flexibilität deutlich komplexer. So programmiert man im Regelfall weder die IP-Adresse noch die Portadresse fest in die Applikation ein. Stattdessen werden symbolische Namen spezifiziert, die in entsprechenden Konfigurationsdateien abgelegt sind (z.B. `/etc/services`) oder über andere Serverdienste (z.B. DNS für die Auflösung von Hostnamen zu IP-Adressen) geholt werden. Für diese Aktionen existiert eine ganze Reihe weiterer Bibliotheksfunktionen.

Die Schnittstelle unterstützt nicht nur die Kommunikation über TCP/IP, sondern auch über UDP. Bei UDP handelt es sich um einen verbindungslosen Dienst. Der Server muss also kein `accept()` aufrufen, sondern kann direkt vom Socket (Bind-Socket) lesen. Um hier Informationen über den Absender zu erhalten, gibt es eigene Systemaufrufe, bei denen IP- und Portadresse des Absenders übernommen werden.

Auch die über IP angebotenen Multicast- und Broadcast-Dienste werden über die Socket-Schnittstelle bedient und sind letztlich Attribute (Parameter) des Sockets.

Die im IP-Protokoll festgelegten Datenstrukturen gehen von einem »Big Endian«-Ablageformat der Variablen (z.B. Integer) aus. Das bedeutet, dass eine Applikation, die auf einem Rechner abläuft, der ein »Little Endian«-Datenablageformat verwendet, die Inhalte der Datenstrukturen erst konvertieren muss. Dieser Vorgang wird durch die Funktionen `ntohX()` (net to host) und `htonX()` (host to net) unterstützt (X steht hier für »s« oder »l«, also für short oder long). Auf einem Rechner mit »Big Endian«-Ablageformat sind die Funktionen (Makros) leer, auf einem Rechner mit »Little Endian«-Ablageformat wird dagegen eine Konvertierung durchgeführt.

Schreibt man eine verteilte Applikation, die unabhängig vom Datenablageformat ist, müssen alle Datenstrukturen in ein einheitliches Format gewandelt werden. Die Daten müssen zum Verschicken konvertiert und beim Empfang wieder rückkonvertiert werden. Folgende Technologien unterstützen den Programmierer bei dieser Arbeit: Remote Procedure Call (RPC), Remote Message Invocation (RMI, für Java), OLE (Microsoft) und Corba (Unix). RPC, RMI, OLE und Corba kümmern sich dabei nicht nur um die Konvertierung, sondern auch um den eigentlichen Datentransport.

4.6 Condition-Variable (Events)

Während über ein Semaphor der Zugriff auf ein gemeinsam benutztes Betriebsmittel synchronisiert wurde (Synchronisation konkurrierender Tasks), werden zwei oder mehrere kooperierende Tasks über sogenannte Condition-Variablen oder Events bezüglich ihres Programmablaufs synchronisiert. Dabei schläft eine Task so lange, bis eine Condition-Variable (Ereignis), welche durch eine andere Task gesetzt wird, ihren Zustand ändert.

Eine Condition-Variable ist ein Synchronisationselement, welches Rechenprozessen erlaubt, so lange den Prozessor freizugeben, bis eine bestimmte Bedingung erfüllt ist. Die Basisoperationen auf Condition-Variablen sind:

- Schlafen, bis die Condition-Variable den Zustand ändert, und
- Setzen der Condition-Variablen.

Das Setzen der Condition-Variablen, wenn kein anderer Job darauf schläft, bleibt wirkungslos; das Ereignis wird nicht zwischengespeichert. Um die daraus resultierende Deadlock-Situation zu vermeiden, kombiniert man die Condition-Variable mit einem Mutex.

Um innerhalb von Realzeitapplikationen, die auf ein POSIX-Interface aufsetzen, eine Condition-Variable nutzen zu können, gibt es einen Satz von Funktionen.

Die Condition-Variable selbst wird innerhalb der Applikation definiert. Die Initialisierung kann statisch (Compiler) durch Zuweisung mit dem Makro `PTHREAD_COND_INITIALIZER` erfolgen oder dynamisch durch Aufruf der Funktion `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);` angestoßen werden. Wird für `attr` null übergeben, werden die Default-Attribute eingesetzt.

Um auf die Signalisierung der Condition-Variablen zu warten, dienen die Funktionen `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)` und `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)`. Vor Aufruf einer der beiden Funktionen muss das zugehörige Mutex `mutex` zwingend reserviert (gelockt) sein. Die Funktionen geben atomar (ohne dass sie dabei unterbrochen werden) das Mutex frei und legen den aufrufenden Thread schlafen, bis die Signalisierung erfolgt und gleichzeitig das Mutex wieder gelockt werden kann. Nach dem Aufruf einer der beiden Funktionen ist das Mutex `mutex` also wieder gesperrt.

Zur Signalisierung dienen die Funktionen `int pthread_cond_signal(pthread_cond_t *cond)` und `int pthread_cond_broadcast(pthread_cond_t *cond)`. Erstere weckt genau einen Job auf, letztere alle auf der Condition-Variablen `cond` schlafenden Jobs.

Um die Condition-Variable `cond` wieder zu deinitialisieren, dient die Funktion `int pthread_cond_destroy(pthread_cond_t *cond)`. Allerdings ist diese Funktion nicht zwingend erforderlich, da die Variable ohnehin durch die Applikation selbst definiert wird.

Condition-Variablen lassen sich gut zur Lösung sogenannter Producer-/Consumer-Probleme einsetzen. Bei derartigen Problemen erzeugt die Task Producer eine Information und die Task Consumer verwertet diese. Beispiel 4-25 zeigt den Einsatz zweier Condition-Variablen zur Lösung eines solchen Problems.

Beispiel 4-25 *Condition-Variable zur Lösung des Producer-/Consumer-Problems*

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

static pthread_mutex_t mutex;
static pthread_cond_t cond_consumed = PTHREAD_COND_INITIALIZER;
static pthread_cond_t cond_produced = PTHREAD_COND_INITIALIZER;

static void *producer( void *arg )
{
   int count=0;

   pthread_mutex_lock( &mutex );
   while( 1 ) {
      printf("producer: PRODUCE %d...\n", count++);
      pthread_cond_signal( &cond_produced );
      pthread_cond_wait( &cond_consumed, &mutex );
   }
   return NULL;
}

static void *consumer( void *arg )
{
   int count=0;

   sleep( 1 );
   pthread_mutex_lock( &mutex );
   while( 1 ) {
      printf("consumer: CONSUME %d...\n", count++);
      pthread_cond_signal( &cond_consumed );
      pthread_cond_wait( &cond_produced, &mutex );
   }
   return NULL;
}

int main( int argc, char **argv, char **envp )
{
   pthread_t p1, p2;
```

```
   if (pthread_mutex_init( &mutex, NULL )) {
      perror("pthread_mutex_init");
      return -1;
   }

   pthread_create( &p2, NULL, consumer, NULL );
   pthread_create( &p1, NULL, producer, NULL );

   pthread_join( p1, NULL );
   pthread_join( p2, NULL );

   pthread_mutex_destroy( &mutex );
   return 0;
}
```

4.7 Signale

Bei Signalen (englisch Signals) handelt es sich um Software-Interrupts auf Applikationsebene, das heißt: Ein Signal führt zu einer Unterbrechung des Programmablaufs innerhalb der Applikation. Das Programm wird entweder abgebrochen oder reagiert mit einem vom Programm zur Verfügung gestellten Signal-Handler (ähnlich einer Interrupt-Service-Routine).

Signale können zum einen durch eine Applikation ausgelöst werden (Systemcall `kill()`), zum anderen aber auch durch Ereignisse innerhalb des Betriebssystem selbst. So führt zum Beispiel der Zugriff auf einen nicht vorhandenen Speicherbereich (z.B. der Zugriff auf Adresse Null) innerhalb des Betriebssystems dazu, dem Prozess ein Segmentation-Fault-Signal zu schicken, welches der Prozess abfangen könnte. In der zugehörigen Segmentation-Fault-ISR würden alle notwendigen Daten noch gespeichert. Erst danach würde die Applikation beendet werden.

Die Unterschiede zwischen einer Condition-Variablen (Event) und einem Signal sind in Tabelle 4-5 dargestellt.

Signal	Condition-Variable
Die Signalisierung kommt asynchron zum Programmablauf und wird asynchron verarbeitet.	Die Signalisierung kommt synchron zum Programmablauf und wird synchron verarbeitet.
Charakter einer Interrupt-Service-Routine (Software-Interrupt).	Rendezvous-Charakter

Tabelle 4-5 *Unterschiede zwischen einem Signal und einer Condition-Variablen*

> Ein Signal führt – wenn nicht anders konfiguriert – zum sofortigen Abbruch eines gerade aktiven Systemcalls. Werden im Rahmen einer Applikation Signale verwendet bzw. abgefangen, muss jeder Systemcall daraufhin überprüft werden, ob selbiger durch ein Signal unterbrochen wurde, und, falls dieses zutrifft, muss der Systemcall *neu* aufgesetzt werden!

Unix-Betriebssysteme unterstützen Signale über verschiedene Interfaces. Die größte Kontrolle gibt dabei die Funktion `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`. Mit dieser Funktion kann über den Parameter `oldact` die aktuelle Konfiguration ausgelesen und alternativ oder gleichzeitig mit `act` eine neue Konfiguration für das Signal `signum` gesetzt werden.

Die Konfiguration selbst findet sich in der Datenstruktur `struct sigaction`. Der Programmierer hat die Möglichkeit, die Adresse einer Signal-Handler-Funktion (Typ `sa_sighandler`) anzugeben oder alternativ die Adresse eines Signal-Handlers (`sa_sigaction`) zu verwenden, der drei Parameter übergeben bekommt und dem damit vielfältige Information auch über den Prozess zur Verfügung stehen, der das Signal geschickt hat. In das Bitfeld `sa_mask` können die Signalnummern aktiviert werden, die während der Abarbeitung des Signal-Handlers geblockt werden sollen. Über das Datenstrukturelement `sa_flags` wird unter anderem ausgewählt, ob der einfache oder der alternative Signal-Handler verwendet werden soll.

Beispiel 4-26 *Programmierbeispiel Signal*

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

#define message "SIGINT caught\n"

void signal_handler(int value)
{
   write( 1, message, strlen(message) ); // printf is not signalsafe
}

int main(int argc, char **argv, char **envp )
{
   struct sigaction new_action;
   int time_to_sleep;

   new_action.sa_handler = signal_handler;
   sigemptyset( &new_action.sa_mask );
   new_action.sa_flags = 0;

   sigaction( SIGINT, &new_action, NULL );
```

```
   printf("pid: %d\n", getpid() );
   time_to_sleep = 20;
   while( time_to_sleep )
      time_to_sleep = sleep( time_to_sleep );

   return 0;
}
```

Applikationen schicken Signale durch Aufruf der Funktion `int kill(pid_t pid, int sig)`:. `pid` spezifiziert dabei den Job, dem das Signal `num` zugestellt werden soll.

4.8 Peripheriezugriff

Der applikationsseitige Zugriff auf Peripherie erfolgt in mehreren Schritten. Im ersten Schritt teilt die Applikation dem Betriebssystem mit, auf welche Peripherie sie in welcher Art (zum Beispiel lesend oder schreibend) zugreifen möchte. Das Betriebssystem prüft, ob der Zugriff möglich und erlaubt ist. Ist dies der Fall, bekommt die Applikation die Zugriffsberechtigung in Form eines Handles beziehungsweise Deskriptors mitgeteilt. Im zweiten Schritt greift die Applikation mithilfe des Deskriptors auf die Peripherie so oft und so lange wie notwendig zu. Erst wenn keine Zugriffe mehr notwendig sind, wird das Handle beziehungsweise der Deskriptor wieder freigegeben (Schritt 3).

Für den ersten Schritt steht in Unix-basierten Systemen die von normalen Dateizugriffen her bekannte Funktion `int open(const char *pathname, int flags)` zur Verfügung. Da das Konzept *alles ist eine Datei* praktiziert wird, wird die Peripherie über einen Dateinamen (`pathname`), den Gerätedateinamen identifiziert. Die Art des Zugriffs (lesend, schreibend, nicht blockierend) wird über die in der Headerdatei `<fcntl.h>` definierten `flags` spezifiziert:

- `O_RDONLY`: Lesender Zugriff
- `O_WRONLY`: Schreibender Zugriff
- `O_RDWR`: Lesender und schreibender Zugriff
- `O_NONBLOCK`: Nichtblockierender Zugriff

Ein negativer Rückgabewert der Funktion `open` bedeutet, dass der Zugriff nicht möglich ist. Anhand der (thread-)globalen Variablen `errno` kann die Applikation die Ursache abfragen. Ein positiver Rückgabewert repräsentiert das Handle beziehungsweise den Deskriptor.

ssize_t und size_t

Viele Standardfunktionen verwenden den Datentyp `size_t` (size type, vorzeichenlos) oder `ssize_t` (signed size type, vorzeichenbehaftet). Typischerweise repräsentiert er den originären Typ `unsigned long` beziehungsweise `long`.

Applikationen greifen auf die Peripherie dann über die Funktionen `ssize_t read(int fd, void *buf, size_t count)` und `ssize_t write(int fd, const void *buf, size_t count)` zu. Der Parameter `fd` ist der Filedeskriptor (Handle), der von `open` zurückgegeben wird. Die Adresse `buf` enthält den Speicherbereich, in den `read` die Daten ablegt, und `count` gibt an, welche Größe dieser Speicherbereich hat.

`read` kopiert von der Peripherie mindestens ein Byte und maximal `count` Bytes an die Speicheradresse `buf` und gibt – solange kein Fehler aufgetreten ist – die Anzahl der kopierten Bytes zurück. Gibt es zum Zeitpunkt des Aufrufes der Funktion `read` keine Daten, die gelesen werden können, legt die Funktion im sogenannten blockierenden Modus den aufrufenden Thread schlafen und weckt ihn wieder auf, wenn mindestens ein Byte in den Speicherbereich `buf` abgelegt wurde. Werden jedoch wie bei einem Dateizugriff am Ende der Datei keine Daten mehr erwartet, gibt die Funktion jedoch direkt 0 zurück. Im nicht blockierenden Modus (`O_NONBLOCK`) quittiert die Funktion den Umstand, dass keine Daten zur Verfügung stehen, mit einem negativen Rückgabewert. Die (thread-)globale Variable `errno` hat dann den in der Headerdatei <errno.h> definierten Wert `EAGAIN`.

Auch im Fehlerfall, wenn beispielsweise der Aufruf durch ein Signal unterbrochen wurde, gibt `read` einen negativen Wert zurück und `errno` enthält den zugehörigen Fehlercode.

Die Funktion `write` arbeitet identisch. Die Funktion schreibt mindestens ein Byte, maximal aber `count` Bytes auf das über den Filedeskriptor `fd` spezifizierte Gerät. Falls nicht geschrieben werden kann, weil beispielsweise das Sendefifo einer seriellen Schnittstelle bereits voll ist, wird der zugreifende Thread in den Zustand *schlafend* versetzt, bis der Zugriff möglich ist. Beim nicht blockierenden Zugriff gibt `write` in diesem Fall – schreiben ist zur Zeit nicht möglich – einen negativen Wert zurück und `errno` enthält wieder den Wert `EAGAIN`.

Direct-IO versus Buffered-IO

Wenn Ein- und Ausgabebefehle wie die Systemcalls `read()` und `write()` direkt, ohne Verzögerung umgesetzt werden, spricht man von *Direct-IO*.

Funktionen wie beispielsweise `fprintf()`, `fread()`, `fwrite()` oder `fscanf()` gehören zur sogenannten *Buffered-IO*. Bei dieser puffert das Betriebssystem (genauer die Bibliothek) die Daten aus Gründen der Effizienz zwischen. Erst wenn es sinnvoll erscheint, werden die zwischengespeicherten Daten per Systemcall `read()` oder `write()` transferiert. Dies ist beispielsweise der Fall, wenn ein »\n« ausgegeben wird oder wenn 512 Byte Daten gepuffert sind.

Da nur die Direct-IO-Funktionen die volle Kontrolle über die Ein- und Ausgabe geben, werden in Realzeitapplikationen nur diese für den Datentransfer mit der Peripherie eingesetzt.

Beispiel 4-27
Der Gerätezugriff über read und write

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

static char *hello = "Hello World";

int main( int argc, char **argv, char **envp )
{
   int dd; /* device descriptor */
   ssize_t bytes_written, bytes_to_write;
   char *ptr;

   dd = open( "/dev/ttyS0", O_RDWR ); /* blocking mode */
   if (dd<0) {
      perror( "/dev/ttyS0" );
      return -1;
   }

   bytes_to_write = strlen( hello );
   ptr = hello;
   while (bytes_to_write) {
      bytes_written = write( dd, hello, bytes_to_write );
      if (bytes_written<0) {
         perror( "write" );
         close( dd );
         return -1;
      }
      bytes_to_write -= bytes_written;
      ptr += bytes_written;
   }
   close( dd );
   return 0;
```

```
}
```

Der Zugriffsmodus (blockierend oder nicht blockierend) kann im Übrigen per `int fcntl(int fd, int cmd, ... /* arg */)` auch nach dem Öffnen noch umgeschaltet werden (Beispiel 4-28). Dazu werden zunächst mit dem Kommando (`cmd`) `F_GETFL` die aktuellen Flags gelesen und dann wird entweder per Oder-Verknüpfung der nicht blockierende Modus gesetzt beziehungsweise per XOR-Verknüpfung der blockierende Modus wieder aktiviert.

Beispiel 4-28 *Programmbeispiel Zugriffsmodus umschalten*

```
int fd, fd_flags, ret;
...
fd_flags = fcntl( fd, F_GETFL );
if (fd_flags<0) {
    return -1; /* Fehler */
}
fd_flags|= O_NONBLOCK; /* nicht blockierenden Modus einschalten */
if (fcntl( fd, F_SETFL, (long)fd_flags )<0 ) {
    return -1; /* Fehler */
}
...
fd_flags = fcntl( fd, F_GETFL );
if (fd_flags<0) {
    return -1; /* Fehler */
}
fd_flags~= O_NONBLOCK; /* nicht blockierenden Modus einschalten */
if (fcntl( fd, F_SETFL, (long)fd_flags )<0 ) {
    return -1; /* Fehler */
}
...
```

Werden regelmäßig (im nicht blockierenden Modus) Peripheriegeräte daraufhin abgeprüft, ob Daten zum Lesen vorliegen oder ob Daten geschrieben werden können, spricht man von Polling. Polling ist insofern ungünstig, da ein Rechner auch dann aktiv wird, wenn eigentlich nichts zu tun ist. Besser ist der ereignisgesteuerte Zugriff (blockierender Modus), bei dem die Applikation nur dann aktiv wird, wenn Daten gelesen oder geschrieben werden können. Allerdings hat der blockierende Modus in Unix-artigen Systemen den Nachteil, dass per `read` oder `write` immer nur eine Quelle abgefragt werden kann. Oftmals sind aber mehrere Datenquellen oder Datensenken abzufragen. Dieses Problem kann auf zwei Arten gelöst werden.

Die klassische Methode basiert auf der Funktion `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)`, die auch eine portable Möglichkeit ist, einen Thread mit einer Genauigkeit von Mikrosekunden schlafen zu legen. `Select` werden

Sets von Deskriptoren übergeben, die auf Aktivitäten überwacht werden. Sollten an einem der in `readfds` eingetragenen Deskriptoren Daten gelesen werden können, ohne dass ein (blockierend) zugreifender Thread schlafen gelegt wird, returniert die Funktion. Ebenso verhält es sich mit `writefds`: `select` returniert, falls an einem der überwachten Deskriptoren Daten ohne zu blockieren ausgegeben werden können. Das dritte Set, `exceptfds`, wird nur im Kontext von Netzwerkverbindungen verwendet. Sollten an einem der im Set spezifizierten Socketdeskriptoren sogenannte Out-of-Band-Daten vorliegen, kehrt die Funktion zurück. Der letzte Parameter der Funktion (`timeout`) dient der Zeitüberwachung. Sollten an keinem der in den Sets definierten Deskriptoren Daten ohne zu blockieren transferiert werden können, schläft `select` maximal für die in `timeout` angegebene Zeitspanne. Ist `timeout` null, schläft die Funktion ohne Zeitüberwachung. Aus Gründen der Effizienz muss im Parameter `nfds` beim Aufruf noch der höchste in den Sets spezifizierte Deskriptor plus eins übergeben werden.

Die übergebenen Sets von Deskriptoren (Datentyp `fd_set`) sind implementierungstechnisch betrachtet meistens als Bitfelder realisiert. Jedes Bit steht für einen Deskriptor. Ist das Bit gesetzt, soll der entsprechende Deskriptor überwacht werden. Um ein Set von Deskriptoren zu initialisieren, steht das Makro `void FD_ZERO(fd_set *set)` zur Verfügung. Es stellt sicher, dass alle Bits zu null gesetzt sind. Mit der Funktion `void FD_SET(int fd, fd_set *set)` wird dem Set ein Deskriptor hinzugefügt (Bit setzen). Ein Deskriptor wird über `void FD_CLR(int fd, fd_set *set)` wieder aus dem Set entfernt (ein einzelnes Bit wird gelöscht). Und um nach dem Aufruf von `select` festzustellen, über welchen Deskriptor Daten gelesen oder geschrieben werden können, verwenden Sie das Makro `int FD_ISSET(int fd, fd_set *set)` (siehe Beispiel 4-29).

Beispiel 4-29
Programmbeispiel select

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define max(x,y) ((x) > (y) ? (x) : (y))

int main( int argc, char **argv, char **envp )
{
   fd_set rds;
   struct timeval tv;
   int retval, nds=0;
   int dd = 0; /* watch stdin to see when it has input */

   FD_ZERO(&rds); /* initialize the deskriptor set */
```

```
   /* for every deskriptor you want to add... */
   FD_SET(dd, &rds); /* add the deskriptor you want to watch to the set */
   nds = max(nds, dd);

   tv.tv_sec = 5; /* wait up to five seconds. */
   tv.tv_usec = 0;

   retval = select(nds, &rds, NULL, NULL, &tv);

   if (retval) {
      if ( FD_ISSET(dd, &rds) )
         printf("Data is available now.\n");
   } else {
      printf("No data within five seconds.\n");
   }
   return 0;
}
```

Die Funktion `select` ist problematisch. Per definitionem soll ein dem `select` folgender `read`- oder `write`-Aufruf nicht blockieren. Dieses Verhalten müsste letztlich ein Treiber sicherstellen, was aber nur in Ausnahmefällen gewährleistet ist. Falls also mehrere Threads auf die gleiche Peripherie zugreifen, könnte `select` dem einen Thread signalisieren, dass das Lesen der Daten möglich ist, der andere Thread aber in dem Moment diese Daten bereits abholen, sodass der erste Thread, wenn er das `read` aufruft, doch schlafen gelegt wird.

Die modernere Variante, mehrere Ein- oder Ausgabekanäle zu überwachen, basiert auf der Thread-Programmierung. Für jeden Kanal (Filedeskriptor, Handle) wird ein eigener Thread aufgezogen, der dann blockierend per `read` oder `write` zugreift. Neben der einfacheren und übersichtlicheren Programmierung hat das auch den Vorteil, dass damit direkt das nebenläufige Programmieren (Stichwort Multicore-Architektur) unterstützt wird (Beispiel 4-30).

Beispiel 4-30 *Asynchroner Zugriff über Threads*

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>

static void *async_thread( void *arg )
{
   int fd=*((int*)(&arg)); // casting to avoid compiler warning
   char buf[64];

   read( fd, buf, sizeof(buf));
   pthread_exit( NULL );
}
```

```
int main( int argc, char **argv, char **envp )
{
   pthread_t async_thread_id;
   int fd;

   fd = open( "device.io", O_RDONLY );
   if( fd<0 ) {
      perror( "device.io" );
      return -1;
   }

   if( pthread_create(&async_thread_id,NULL,async_thread,
      (void*)(long)fd)!= 0 ) { // double cast to avoid warning
      fprintf(stderr,"creation of thread failed\n");
      return -1;
   }
   // perform other tasks
   // ...

   // synchronise with file I/O
   pthread_join( async_thread_id, NULL );
   // ...

   return 0;
}
```

Sind keine Zugriffe auf die Peripherie über den Deskriptor mehr notwendig, kann dieser per `int close(int fd)` wieder freigegeben werden. In vielen Anwendungen fehlt jedoch zu einem `open` das korrespondierende `close`. Das ist insofern nicht dramatisch, da das Betriebssystem beim Ende einer Applikation alle noch offenen Deskriptoren von sich aus wieder freigibt.

Manche Peripheriegeräte, wie beispielsweise eine Grafikkarte, transferieren nicht nur einzelne Bits oder Bytes, sondern umfangreiche Speicherbereiche. Diese Daten über einzelne `read`- und `write`-Aufrufe zu verschieben, wäre sehr ineffizient: Mit jedem Zugriff ist ein Kontextwechsel notwendig und außerdem wird mindestens in der Applikation ein Speicherbereich benötigt, in den beziehungsweise von dem die Daten zwischen Applikation und Hardware transferiert werden. Daher bieten Betriebssysteme und die zu den Geräten gehörenden Treiber oft die Möglichkeit, Speicherbereiche der Hardware (oder aber auch des Kernels) in den Adressraum einer Applikation einzublenden. Die Applikation greift dann direkt auf diese Speicherbereiche zu.

Mit Splice effizient kopieren

Auf klassischem Wege werden Daten kopiert, indem die Applikation einen Buffer reserviert und dann per `read` Daten von der Eingabequelle in diesen Buffer transferieren lässt. Typischerweise werden vom Kernel hierzu die Daten über den Gerätetreiber zunächst in den Kernel-Speicher gelesen und von dort in den von der Applikation bereitgestellten Buffer kopiert. Sind die Daten dort angelangt, übergibt die Applikation per `write` die Daten dem Kernel, der sie typischerweise noch einmal zwischenspeichert und dann den Gerätetreiber aktiviert, damit dieser den eigentlichen Schreibzugriff durchführt.

Um Daten von A nach B zu kopieren, werden diese also bis zu viermal angefasst und es gibt mehrere Kontextwechsel – nicht wirklich effizient.

Linux bietet als effiziente Alternative die Funktion `ssize_t splice(int fd_in, loff_t *off_in, int fd_out, loff_t *off_out, size_t len, unsigned int flags);` an. Diese kopiert ohne Umweg über das Userland direkt die Daten von der Eingabequelle an die Ausgabequelle. Ein- und Ausgabequellen sind dabei wie üblich als Filedeskriptoren referenziert.

Dazu öffnet die Applikation als Erstes die Gerätedatei (Funktion `open`), die den Zugriff auf die gewünschte Peripherie ermöglicht. Der zurückgelieferte Gerätedeskriptor `dd` (Device Descriptor) wird dann der Funktion `void *mmap(void *addr, size_t length, int prot, int flags, int dd, off_t offset)` übergeben. Ist der Parameter `addr` mit einer Adresse vorbelegt, versucht das Betriebssystem, den Speicher an diese Adresse einzublenden. Typischerweise ist `addr` aber auf null gesetzt und das System selbst sucht eine günstige, freie Adresse aus. Der Parameter `length` gibt die Länge des Speicherbereichs an, der aus Sicht der Peripherie ab dem Offset `offset` eingeblendet werden soll, `prot` spezifiziert den gewünschten Speicherschutz und ist entweder `PROT_NONE`, `PROT_EXEC`, `PROT_READ` oder `PROT_WRITE` (bitweise verknüpft). Das Argument `flags` legt fest, ob Änderungen an dem eingeblendeten Speicherbereich für andere Threads sichtbar sind, die den Bereich ebenfalls eingeblendet haben. Hierzu stehen die vordefinierten Werte `MAP_SHARED` und `MAP_PRIVATE` zur Verfügung.

Beispiel 4-31 *Speicherbereiche in den Adressraum einblenden*

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main( int argc, char **argv, char **envp )
{
   int dd;
   void *pageaddr;
   int *ptr;
```

```
   dd = open( "/dev/mmap_dev", O_RDWR );
   if (dd<0) {
      perror( "/dev/mmap_dev" );
      return -1;
   }
   pageaddr = mmap( NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, dd, 0 );

   printf("pageaddr mmap: %p\n", pageaddr );
   if (pageaddr) {
      ptr = (int *)pageaddr;
      printf("*pageaddr: %d\n", *ptr );
      *ptr = 55; /* access to the mapped area */
   }
   munmap( pageaddr, 4096 );

   return 0;
}
```

Um einen eingeblendeten Speicherbereich wieder freizugeben, ruft die Applikation `int munmap(void *addr, size_t length)` auf.

4.9 Bitoperationen

In vielen Fällen müssen Realzeitsysteme einzelne Bits einer Speicherzelle testen, setzen, rücksetzen oder invertieren. Die übrigen Bits sollen dabei unverändert bleiben. Diese Operationen lassen sich über bitweise UND-, ODER- und XOR-Operationen realisieren.

Testen von Bits

Um einzelne Bits zu testen, wird eine Bitmaske gebildet. Für jedes zu testende Bit wird in der Maske das korrespondierende Bit auf eins gesetzt, die übrigen Bits zu null. Diese Maske wird per bitweisem UND mit der zu testenden Variablen verknüpft, das Ergebnis schließlich mit der Maske verglichen. Ist das Ergebnis TRUE, war mindestens eines der relevanten Bits gesetzt. Mit dem Vergleich von Ergebnis und Maske kann man überprüfen, ob alle zu untersuchenden Bits gesetzt sind.

Beispiel 4-32
Testen von Bits

Ein 8-Bit-Register wird durch die Variable `register` repräsentiert. Es soll zunächst getestet werden, ob das zweite *und* das dritte Bit (also die Bits mit der Wertigkeit 2^1 und 2^2) gesetzt sind. Der Zustand der übrigen Bits ist irrelevant. Mit dem folgenden Code wird die Operation (Testen der Bits) ausgeführt:

```
maske = 0x06; /* Ist das 2. u. 3. Bit auf "1"? */
```

```
if ( (register&maske)==maske ) {
    printf("Beide Bits sind gesetzt\n");
} else {
    printf("Es sind nicht beide Bits gesetzt\n");
}
```

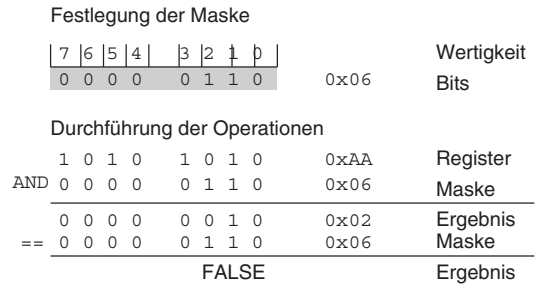


Abbildung 4-10
Test, ob alle zu prüfenden Bits gesetzt sind

Der Test, ob entweder das zweite oder das dritte Bit gesetzt ist, sieht folgendermaßen aus:

```
maske = 0x06; /* Ist das 2. o. 3. Bit auf "1"? */
if ((register&maske)!=0) {
    printf("Mindestens eines der Bits ist gesetzt\n");
} else {
    printf("Keines der Bits ist gesetzt\n");
}
```

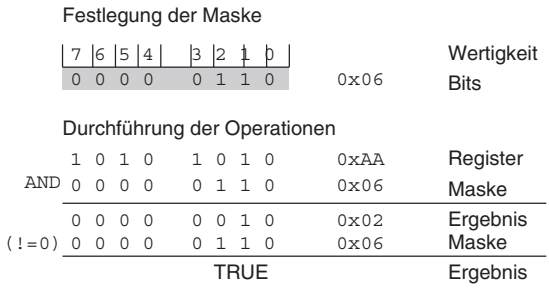


Abbildung 4-11
Test, ob mindestens ein Bit gesetzt ist

Setzen von Bits

Bits werden per logischem ODER gesetzt. Bits, die gesetzt werden sollen, werden bitweise ODER-verknüpft mit einer Eins, Bits, die ihren Wert (egal ob null oder eins) behalten sollen, werden bitweise ODER-verknüpft mit einer Null.

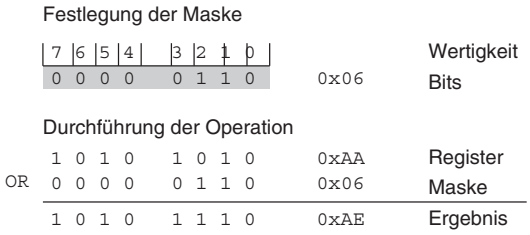
Beispiel 4-33
Setzen von Bits

Ein 8-Bit-Register wird durch die Variable `register` repräsentiert. In `register` soll das zweite und das dritte Bit (also die Bits mit der Wertigkeit 2^1 und 2^2) gesetzt werden, alle anderen Bits sollen ihren bisherigen Zustand behalten. Damit ergibt sich die Maske `0x06`. Mit dem folgenden Code wird die Operation (Setzen der Bits) ausgeführt:

```
register = register | 0x06; /* 2. u. 3. Bit auf eins setzen */
```

Falls `register` vor Ausführung der Operation den Wert `0xaa` hatte (jedes zweite Bit eins), hat `register` nach der Operation den Wert `0xae`.

Abbildung 4-12
Bits setzen



Rücksetzen von Bits

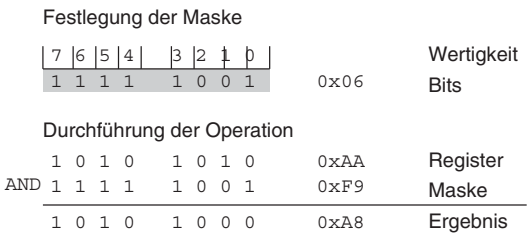
Per UND-Verknüpfung werden Bits zurückgesetzt. Sämtliche Bits, die unverändert bleiben sollen, werden bitweise UND-verknüpft mit einer Eins. Bits, die zurückgesetzt werden sollen, werden bitweise UND-verknüpft mit einer Null.

Beispiel 4-34
Rücksetzen von Bits

Ein 8-Bit-Register wird durch die Variable `register` repräsentiert. In `register` sollen das zweite und das dritte Bit (also die Bits mit der Wertigkeit 2^1 und 2^2) zu null gesetzt werden, alle anderen Bits sollen ihren bisherigen Zustand behalten. Damit ergibt sich die Maske `0xF9`. Mit dem folgenden Code wird die Operation (Rücksetzen der Bits) ausgeführt:

```
register = register & 0xF9; /* 2. u. 3. Bit null setzen */
```

Abbildung 4-13
Bits rücksetzen



Invertieren von Bits

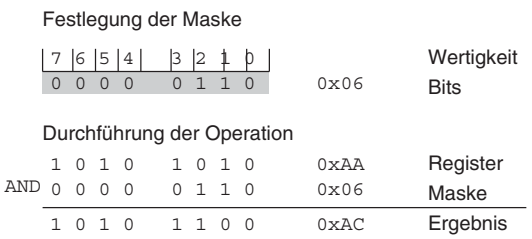
Einzelne Bits können per XOR (exklusives Oder) invertiert werden. In der dazu notwendigen Maske werden die zu invertierenden Bits zu eins gesetzt, die anderen bleiben null.

Beispiel 4-35
Invertieren von Bits

Ein 8-Bit-Register wird durch die Variable `register` repräsentiert. In `register` sollen das zweite und das dritte Bit (also die Bits mit der Wertigkeit 2^1 und 2^2) invertiert werden, alle anderen Bits sollen ihren bisherigen Zustand behalten. Damit ergibt sich die Maske `0x06`. Mit dem folgenden Code wird die Operation (Invertieren der Bits) ausgeführt:

```
register = register ^ 0x06; /* 2. u. 3. Bit invertieren */
```

Abbildung 4-14
Bits invertieren



4.10 Memory Management

Eine Realzeitapplikation sollte bezüglich Memory Management die folgenden drei Aspekte beachten:

1. Sie muss das Auslagern von Speicherseiten (Swapping) verhindern, indem sie beispielsweise `mlockall()` aufruft.
2. Sie beugt Verzögerungen vor, die entstehen können, wenn die Realzeitapplikation Daten auf dem Stack ablegt (lokale Variablen). Benötigt der Betriebssystemkern hierfür nämlich eine neue Page (Page-Fault) und sind sämtliche Pages belegt, muss er die Seite eines gerade nicht aktiven Jobs auslagern und die frei gewordene Seite der Realzeitapplikation zur Verfügung stellen.

   Um der sich dabei ergebenden Wartezeit vorzubeugen, reserviert die Realzeitapplikation direkt zu Beginn der Verarbeitung ausreichend Speicher auf dem Stack. In diesem Fall kommt es direkt zum Page-Fault, der Kernel lagert die Seiten anderer Jobs aus und teilt den frei gewordenen Speicherplatz der Realzeitapplikation zu. Diese Technik nennt sich Stack-Prefault.
3. Sie beugt Verzögerungen vor, die entstehen können, wenn die Realzeitapplikation Daten auf dem Heap ablegt (`malloc()`). Sind sämtli-

che Speicherseiten belegt, kommt es zum Page-Fault und der Kernel lagert die Page eines gerade nicht aktiven Jobs aus. Die frei gewordene Speicherseite wird der Realzeitapplikation zur Verfügung gestellt.

Daher reserviert die RT-Applikation direkt zu Beginn per `malloc()` den benötigten Speicher. Während des normalen Ablaufs (Hauptschleife) der RT-Applikation sollte `malloc()` vermieden werden.

Schutz vor Auslagern

Um das Auslagern (Swapping) der Speicherseiten, die zu einer Realzeitapplikation gehören, zu verhindern, stellen Unix-basierte Betriebssysteme die Systemcalls `int mlock(const void *addr, size_t len)`, `int munlock(const void *addr, size_t len)`, `int mlockall(int flags)` und `int munlockall(void)` zur Verfügung. Mit `mlock()` werden die Speicherseiten, die zum übergebenen Adressbereich gehören, der bei `addr` beginnt und `len` Byte lang ist, markiert, sodass sie von der Speicherverwaltung des Kernels nicht mehr ausgelagert werden. Die Freigabe erfolgt später über `munlock()`. In Realzeitapplikationen wird jedoch vorwiegend die Variante `mlockall()` verwendet, die je nach Flag entweder die aktuell vom Prozess verwendeten Speicherseiten (`MCL_CURRENT`) oder auch die zukünftigen Speicherseiten (`MCL_FUTURE`) markiert (siehe Beispiel 4-36). In den meisten Fällen dürfte die Kombination `MCL_CURRENT|MCL_FUTURE` Anwendung finden.

Ist per `mlockall()` der Schutz vor Auslagern auch für zukünftig vom Job verwendete Speicherseiten aktiviert, werden Systemcalls, die Speicher reservieren (`mmap()`, `sbrk()` oder `malloc()`), in dem Fall abgebrochen, bei dem kein ausreichender Speicherplatz mehr zur Verfügung steht.

```
if (mlockall(MCL_CURRENT|MCL_FUTURE)==-1) {
   perror("mlockall failed");
   exit(-2);
}
```

Beispiel 4-36 *Schutz vor Auslagern*

Prefault

Mit *Prefault* wird die Technik eines Realzeitprogrammierers umschrieben, mit der er den Betriebssystemkern veranlasst, später benötigte Speicherseiten vorzeitig in den Adressraum der Realzeitapplikation einzubinden. Der Trick besteht darin, direkt zu Beginn der RT-Applikation sämtlichen Speicher zu reservieren, der später benötigt wird. Das betrifft sowohl den Stack (lokale Variablen, Beispiel 4-37) als auch den Heap (`malloc()`, Beispiel 4-38).

Speicher auf dem Stack kann am einfachsten mit der Funktion `void *alloca(size_t size)` reserviert werden. Allerdings ist diese Funktion nur begrenzt portabel. Alternativ kann – wie in Beispiel 4-37 gezeigt – ein ausreichend großes Feld auf dem Stack reserviert werden.

Wichtig ist, den Speicher nicht nur anzufordern, sondern auch einmal auf den Speicher zuzugreifen. Um zu verhindern, dass die Speicherseite direkt nach der Anforderung für einen anderen Job wieder freigeräumt wird, sollte zuvor `mlockall()` aufgerufen werden.

Um die vorgestellten Funktionen aufrufen zu können, benötigt die Applikation Rootrechte (Superuser- oder Adminrechte). Es sei an dieser Stelle noch einmal daran erinnert, dass aus Gründen der Sicherheit und gemäß dem Programmierprinzip »Least Privilege« die Rootrechte nur für das Sperren der Seite aktiviert und direkt danach wieder zurückgenommen werden sollten.

Beispiel 4-37 *Prefault-Stack*

```
#define MAX_SAFE_STACK (8x1024)

void stack_prefault(void)
{
   unsigned char dummy[MAX_SAFE_STACK];

   memset(dummy, 0, MAX_SAFE_STACK);
   return;
}
...
int main( int argc, char **argv, char **envp )
{
   ...
   stack_prefault();
   ...
```

Beispiel 4-38 *Prefault-Heap*

```
#define MAX_SAFE_HEAP (64x1024)

int main( int argc, char **argv, char **envp )
{
   ...
   rt_heap = malloc( MAX_SAFE_HEAP );
   if (rt_heap==NULL) {
      perror("prefault-heap");
      return -1;
   }
   memset(rt_heap, 0, MAX_SAFE_HEAP);
   ...
}
```

5 Realzeitarchitekturen

Moderne Realzeitsysteme bestehen aus Hardware, Systemsoftware und Applikationen. Diese müssen ausgewählt, in ihren Eigenschaften konfiguriert und so zusammengestellt werden, dass sie die Realzeitanforderungen einhalten.

Wesentliche Unterscheidungsmerkmale der Hardware zeigen sich in der Leistungsfähigkeit, dem Stromverbrauch, den Möglichkeiten der Peripheriekopplung, der Robustheit, der Größe und natürlich im Preis. Aus Sicht der Realzeitanwendung ist insbesondere entscheidend, ob die CPU einen oder mehrere Kerne besitzt (Singlecore oder Multicore), welchen Befehlssatz die CPU verarbeitet (x86, ARM), ob es sich um eine 32- oder eine 64-Bit-CPU handelt und über welche Bussysteme Peripherie angeschlossen werden kann.

Bei der Systemsoftware hat der Entwickler die Möglichkeit, eine einfache Laufzeitumgebung einzusetzen, ein Realzeitbetriebssystem zu verwenden oder ein Standardbetriebssystem auszuwählen. In manchen Einsatzszenarien bietet sich zudem die Kombination Realzeitkernel plus Standardbetriebssystem an. Mit Linux steht ohnehin ein Zwitter aus Realzeitbetriebssystem und Standardbetriebssystem zur Verfügung, der reichlich Konfigurationsmöglichkeiten bietet, um zeitliche Anforderungen zu realisieren.

Bei den Applikationen wird man in den wenigsten Fällen auf Standardapplikationen zurückgreifen können. Typischerweise wird die Applikation selbst erstellt, um dabei auch die Möglichkeiten der Systemsoftware und der Hardware programmtechnisch zu nutzen. Entscheidend ist, ob die Applikation singlethreaded oder multithreaded erstellt wird. Letzteres ist moderner und ermöglicht, aktuelle Multicore-Hardware zu nutzen. Bei komplexer Aufgabenstellung ist die damit verbundene Aufteilung in logische Blöcke intuitiver, bringt jedoch Probleme mit der Inter-Prozess-Kommunikation und dem notwendigen Schutz kritischer Abschnitte mit sich.

Zentrales Element beim Aufbau eines Realzeitsystems ist die Systemsoftware, die die folgenden Grundstrukturen für den Aufbau moderner Realzeitsysteme vorgibt:

- Realzeitsystem ohne spezielle Systemsoftware,
- Realzeitsystem basierend auf Standard-Systemsoftware,
- Realzeitsystem, das dediziert eingebaute Realzeiteigenschaften nutzt (Threaded Interrupts),
- Realzeitsystem, bei dem Teile der Applikation in den Kernel verlagert werden (Userland-to-Kernel),
- System mit einem Realzeitkernel auf Singlecore-Basis,
- Realzeitsystem auf Multicore-Basis (Multikern-Ansatz),
- System, das auf die Kombination von Realzeit-Betriebssystem und Standardbetriebssystem setzt (Multikernel-Ansatz).

5.1 Realzeitsysteme ohne spezielle Systemsoftware

Bei dieser Architekturvariante setzt die Applikation ohne Zwischenschicht direkt auf der Hardware auf. Sie kümmert sich damit selbst um die Prozessorinitialisierung und die Bedienung sämtlicher Interrupts. Die Programmierung erfolgt in Assembler und in der Programmiersprache C. Hierfür wird ein Entwicklungssystem mit Cross-Compiler und Cross-Linker benötigt. Entwickler müssen gute Hardware-Kenntnisse haben und Erfahrungen in der systemnahen Programmierung mitbringen. Da es keine spezielle Systemsoftware gibt, existieren auch keine standardisierten Programmierschnittstellen. So müssen beispielsweise Funktionen zum Auslesen von Zeiten selbst geschrieben werden. Da alles selbst codiert werden muss, ist der Aufwand für komplexe Funktionalitäten sehr hoch.

Ohne Systemsoftware gibt es auch kein Multitasking. Daher wird die Applikation singlethreaded entworfen. Allerdings wird beim Softwareentwurf Parallelverarbeitung häufig durch Implementierung unabhängiger Verarbeitungsstränge in Interrupt-Service-Routinen eingeplant (Poor Man's Multithreading).

Das Zeitverhalten bei Realzeitsystemen, die ohne spezielle Systemsoftware auskommen, wird allein durch die Applikation bestimmt und ist damit in weiten Teilen berechenbar. Latenzzeiten ergeben sich im Wesentlichen durch die Länge der Interrupt-Service-Routinen (Interrupt-Latenzzeit). Diese sind wiederum von der Aufteilung der Reaktion in Primär- und Sekundärreaktion durch den Entwickler abhängig.

Sie wählen diese Architektur, wenn Ihr Realzeitsystem aus einem einfachen Mikrocontroller (Singlecore) aufgebaut wird. Dieser Mikrocontroller hat typischerweise eine Verarbeitungsbreite von 8 Bit, manchmal auch von 16 Bit. Die Aufgabenstellung selbst ist nicht komplex, oft handelt es sich um nicht vernetzte Systeme. Die zeitlichen Anforderungen können sowohl weich als auch hart sein, aufgrund der meist schwachen Prozessorleistung liegen sie im zwei- bis dreistelligen Millisekun-

denbereich. Kürzere Zeitanforderungen lassen sich auch einhalten, wenn Teile der Software in Hardware ausgelagert werden, beispielsweise über ein *Field Programmable Gate Array* (FPGA).

Primär- und Sekundärreaktion

Die Realzeitsteuerung verarbeitet Rechenzeitanforderungen fast immer wie in Abbildung 5-1 dargestellt zweistufig.

- Die Primärreaktion ist meist in Form einer Interrupt-Service-Routine ausgeprägt, die aufgrund einer Rechenzeitanforderung aufgerufen wird. Dadurch hat sie eine kurze Latenzzeit. Im Rahmen der ISR wird zumindest der Interrupt quittiert und die Sekundärreaktion aktiviert. Zeitkritische Aufgabenteile werden ebenfalls während der Primärreaktion abgewickelt.
- Die Sekundärreaktion ist typischerweise als Task ausgeprägt, je nach Realzeitarchitektur manchmal auch als Tasklet. In der Sekundärreaktion werden die zeitlich weniger kritischen Aufgaben abgearbeitet.

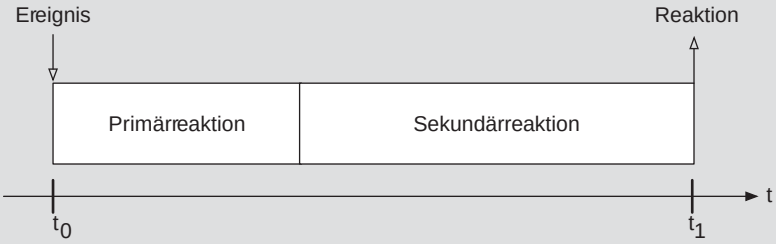


Abbildung 5-1 *Primär- und Sekundärreaktion*

Der Systemarchitekt verteilt die Aufgaben, die aufgrund einer Rechenzeitanforderung anfallen, auf die beiden Teile. Die Erledigung von Aufgaben in der Primärreaktion (ISR) führt zu einer kurzen Reaktionszeit für die zugehörige Rechenzeitanforderung. Allerdings werden dadurch die allgemeinen Latenzzeiten schlechter. Tendenziell versucht der Architekt daher, die Primärreaktion auf das absolute Minimum zu beschränken und die eigentlichen Arbeiten in der Sekundärreaktion durchzuführen.

5.2 Realzeitsysteme basierend auf einem Standard-OS

Sind die zeitlichen Anforderungen weich oder im dreistelligen Millisekundenbereich, kann ein Standardbetriebssystem eingesetzt werden. Dadurch ist man allerdings auf eine standardisierte Hardware-Plattform festgelegt. Meist dürfte dies ein PC sein, häufig in der robusteren Ausprägung ein Industrie-PC. Linux bietet zudem die Möglichkeit, auf andere Hardware-Plattformen und Prozessoren zu setzen. In diesem Fall

wird ein separater Entwicklungsrechner eingesetzt, der die Cross-Entwicklungsumgebung beherbergt.

Der Vorteil dieser Realzeitarchitektur besteht darin, dass der Entwickler bereits mit seinem Betriebssystem, mit der Entwicklungsumgebung und mit den Schnittstellen vertraut ist. Es stehen viele Bibliotheken mit vorgefertigten und auch komplexen Routinen zur Verfügung. Standardbetriebssysteme unterstützen häufig aktuelle und moderne Funktionalitäten. Unter Umständen beeinflussen lange Interrupt-Service-Routinen von Standard-Hardware-Komponenten die Reaktionszeiten der Tasks (Sekundärreaktion).

Realzeitsysteme, die auf Standardbetriebssystemen beruhen, sind allein aufgrund von deren Verbreitung Angriffen (siehe Abschnitt 6.2) ausgesetzt. Insbesondere wenn Windows als Betriebssystem verwendet wird, dürfen sie nicht im sicherheitskritischen Umfeld eingesetzt werden.

Sie wählen diese Architektur bei weichen oder harten Realzeitanforderungen im dreistelligen Millisekundenbereich, falls eine Standard-Hardware (häufig eine PC-Plattform) zur Verfügung steht. Sie wählen diese Architektur ebenso, wenn Cutting-Edge-Eigenschaften, wie beispielsweise NAT für IPv6, gewünscht werden, die im einzusetzenden Betriebssystem bereits implementiert sind.

5.3 Threaded Interrupts (Realzeiterweiterungen für Standardbetriebssysteme)

Standardbetriebssysteme bieten mehrere Möglichkeiten, das Zeitverhalten von Applikationen deterministischer zu gestalten. Hierzu gehören insbesondere die Wahl eines prioritätengesteuerten Schedulings, die Vergabe von Realzeitprioritäten und das Ausschalten von Swapping.

Man hat Linux auf dem Weg vom Standardbetriebssystem hin zu einem Standard-Realzeitbetriebssystem unter dem Namen PREEMPT-RT mit mehreren Patches versehen, die nach und nach auf der einen Seite das Realzeitverhalten verbessert und auf der anderen Seite weitergehende Realzeiteigenschaften implementiert haben [QuKu2012-63]. Je nach eingesetzter Linux-Version kann der zugehörige PREEMPT-RT-Patch Realzeiteigenschaften nachrüsten. Ein aktueller Linux-Kernel der Version 3.0 und höher ist bereits von Haus aus mit den wichtigsten Eigenschaften versehen, die nur noch eingesetzt beziehungsweise aktiviert und konfiguriert werden müssen.

Eine der sehr nützlichen Erweiterungen stellen Threaded Interrupts dar. Bei dieser Technik werden Interrupt-Service-Routinen in zwei Teile geteilt, in eine Basis-ISR und einen Thread. Die Basis-ISR ist für alle Interrupts identisch und versetzt – falls der zugehörige Interrupt auftritt – den zugeordneten, hochprioren Thread in den Zustand lauffähig. Damit wird eine sehr kurze Primärreaktion erreicht. Die tatsächliche Reaktion wird mit der Sekundärreaktion abgehandelt.

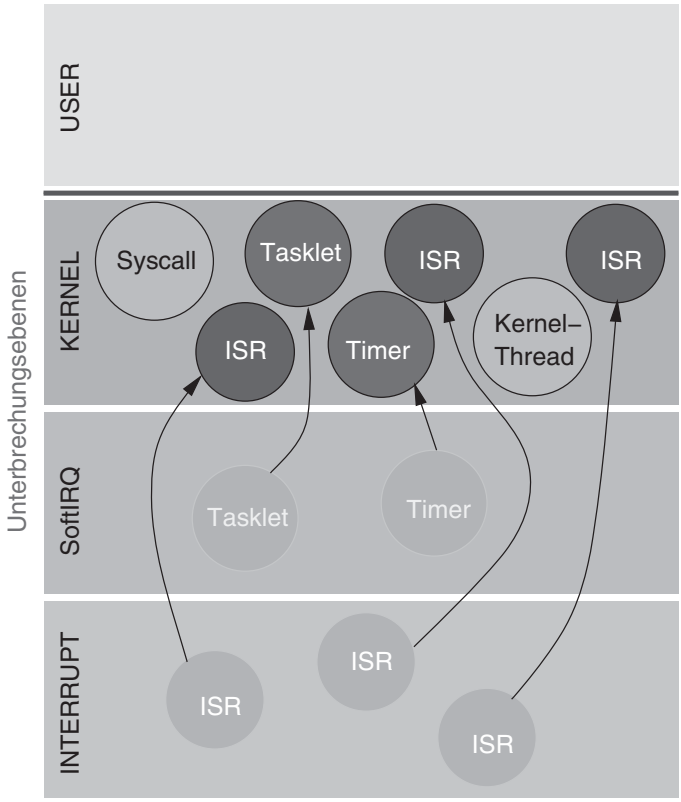
Linux gibt den Threads standardmäßig die Realzeitpriorität 50, also im gesamten Prioritätenspektrum die Priorität 90 (siehe Abschnitt 3.3). Der Thread schließlich enthält den eigentlichen produktiven Code (Sekundärreaktion), der den Interrupt als solchen bearbeitet. Diese Verlagerung gibt dem Systemarchitekten die Möglichkeit, die Interrupts untereinander, insbesondere auch in Beziehung zur Realzeitapplikation, zu priorisieren. Falls notwendig, wird damit die Realzeitapplikation sogar dem Interrupt vorgezogen. Als Threads werden nicht nur Interrupt-Service-Routinen, sondern auch davon abgeleitete Routinen wie Kernel-Timer und Tasklets abgearbeitet. Die Abarbeitung des ISR-Codes im Prozesskontext bedingt natürlich mehr Kontextwechsel. Daher bietet sich diese Technologie weniger für leistungsschwache Hardware an.

Um Threaded Interrupts zu aktivieren, wird einem aktuellen Linux-Kernel beim Booten der Parameter `threadirqs` übergeben. Auf einem aktuellen Ubuntu kann der Parameter in der Datei `/etc/default/grub` eingetragen und hinterher das Skript `update-grub` aufgerufen werden.

Setzt der Entwickler auf diese Realzeitarchitektur, muss er beim Entwurf nicht nur die eigenen Threads, sondern eben auch die sonstigen Systemthreads (ISRs, Timer, Tasklets) berücksichtigen. Die Verteilung der Prioritäten ist so zu gestalten, dass nicht nur die Realzeitbedingungen für die erstellte Realzeitapplikation, sondern auch diejenigen für die Systemthreads eingehalten werden. Das ist nicht trivial, da man die minimal und maximal zulässigen Reaktionszeiten der Systemthreads ebenso wenig kennt wie deren Laufzeiten. Die Laufzeiten wiederum werden benötigt, um die sich ergebenden Verzögerungen zu bestimmen (Stichwort Realzeitnachweis).

Der eigenen Realzeitapplikation sollte demnach auch nur dann eine höhere Priorität zugeteilt werden, falls deren Verarbeitungszeit kurz ist. Diese geht dann auf Einprozessorsystemen als Verdrängungszeit (Wartezeit) in die Reaktionszeit der Systemthreads ein.

Abbildung 5-2
Threaded Interrupts



Diese Architektur ist zu wählen, wenn ausreichend Rechnerleistung zur Verfügung steht und die Anforderungen an das Zeitverhalten im unteren Millisekundenbereich, eventuell niedriger, liegen. Die Architektur bietet sich vor allem auch auf Singlecore-Maschinen an.

5.4 Userland-to-Kernel

Wenn die zeitlichen Anforderungen auf Applikationsebene (im Userland) nicht eingehalten werden können, kann der Systemarchitekt überlegen, zeitkritische Teile der Applikation (Sekundärreaktion) in den Kernel zu verlagern. Allein bedingt durch den Wegfall beziehungsweise die Verkürzung der Kontextwechselzeiten sind die Latenzzeiten kürzer und der Overhead geringer.

Im Wesentlichen können Codesequenzen in folgende Kernel-Komponenten verlagert werden:

- Interrupt-Service-Routinen
- Soft-IRQs

- Kernel-Threads
- Gerätetreiber

Latenzzeiten je nach Hardware im Bereich von wenigen Mikrosekunden lassen sich bei der Abarbeitung von Applikationsteilen im Kontext einer Interrupt-Service-Routine realisieren. Die Verarbeitungszeit der ISR (Primärreaktion) darf dabei nicht zu lang werden, verlängert diese doch direkt die Latenzzeiten der übrigen Systemteile, insbesondere auch der übrigen Interrupt-Service-Routinen.

Eine Verlagerung in Soft-IRQs (siehe Abschnitt 4.3.6) führt dazu, dass die Codesequenzen direkt nach aktivierten Interrupt-Service-Routinen ablaufen. Auf dieser Ebene lassen sich auch relativ leicht zyklisch abzuarbeitende Programmteile realisieren.

Die einfachste Portierung von Teilen der Applikation wird über priorisierbare Kernel-Threads erreicht.

Zeitkritische Teile der Applikation werden dann sinnvoll in Gerätetreiberfunktionen integriert, wenn Realzeitapplikation und Gerätetreiber ohnehin zueinander gehören.

Um diese Realzeitarchitektur zu verwirklichen, ist für den Entwickler eine Einarbeitung in die Kernel-Programmierung notwendig [QuKu2011]. Die in den Betriebssystemkern zu verlagernden Codesequenzen werden als sogenanntes Kernel-Modul realisiert. Allerdings kann der Code nicht eins zu eins vom Userland übertragen werden. Innerhalb des Kernels dürfen keine Floating-Point-Operationen verwendet werden; Bibliotheksfunktionen stehen quasi nicht zur Verfügung. Die Dienste des Kernels (Systemcalls) können genutzt werden, sind aber anders benannt und die Parametrierung unterscheidet sich im Detail von den Pendants des Userlands. Codesequenzen der Interrupt-Ebene und SoftIRQ-Ebene sind weiter eingeschränkt: Funktionen beziehungsweise Funktionalitäten wie das Schlafenlegen sind tabu.

Zu beachten ist weiterhin, dass die Trennung von zeitunabhängigen und zeitkritischen Teilen und vor allem die Verlagerung in den Kernel mit den dazu notwendigen Modifikationen zu einem nicht portierbaren und vor allem stark auf das System abgestimmten Design führt. Es besteht keine saubere Trennung mehr zwischen Applikation und Betriebssystemkern. Es besteht die Gefahr von Systemabstürzen durch Programmierfehler, die sich im Kernel ungeschützt bemerkbar machen können.

Mithilfe dieses Ansatzes lässt sich häufig der Einsatz eines (proprietären) Realzeit-OS vermeiden. Sie wählen diese Architektur, wenn Sie harte Realzeitanforderungen im unteren Milli- oder im Mikrosekundenbereich erfüllen müssen, diese im Userland nicht einhalten können und andere Methoden, wie beispielsweise Threaded Interrupts, nicht ausreichen. Die Architektur ist geeignet für Single- und auch für Multicore-Maschinen.

5.5 Realzeitbetriebssystem

Systeme, die neben den funktionalen Anforderungen auch zeitlichen Anforderungen genügen müssen, basieren klassischerweise auf einem Realzeitbetriebssystem. Dieses ermöglicht – abhängig von der verwendeten Hardware – Zeitanforderungen im einstelligen Millisekundenbereich, manchmal auch darunter, einzuhalten. Dazu bietet der Betriebssystemkern typischerweise ein prioritätengesteuertes Scheduling. Die kleinste Scheduling-Einheit sind Threads. Das Realzeitbetriebssystem weist zumeist einen kleinen Footprint auf, was bedeutet, dass es auch auf Hardware ablauffähig ist, die nur wenig Speicher und geringe Prozessorleistung zur Verfügung stellt. Im Idealfall ist die Interrupt-Latenzzeit allein durch die Hardware bestimmt. Durch die im Vergleich zu einem Standardsystem geringe Verbreitung ist es seltener Angriffen von sogenannten Script-Kiddies ausgesetzt.

Nachteilig bei diesem Ansatz ist die hohe Einarbeitungszeit, da Entwickler sich erst in ein neues Betriebssystem mit seinen Eigenheiten und vor allem oftmals proprietären Schnittstellen einarbeiten müssen. Oft wird eine eigene Entwicklungsumgebung benötigt, die Lizenzkosten verursacht. Für jede eingesetzte Instanz des Realzeitbetriebssystems ist ohnehin eine sogenannte Laufzeitlizenz zu bezahlen.

Ein klassisches Realzeitbetriebssystem besteht aus mehreren Komponenten. Neben dem Userland, welches Bibliotheken und wichtige Systemprogramme enthält, gibt es den Kernel und ein sogenanntes Board Support Package (BSP). Das BSP enthält alle notwendigen Anpassungen, damit das System auf einer bestimmten Hardware ablauffähig ist. Einige Hersteller liefern auch die Cross-Entwicklungswerkzeuge dazu.

Realzeitbetriebssysteme sind nicht immer topaktuell, was die neuesten Funktionalitäten betrifft. Oft dauert es einige Zeit, bis die Hersteller in der Lage sind, aktuelle Trends zu implementieren und ausreichend zu testen. Einige der Realzeitbetriebssysteme bieten ähnliche Möglichkeiten wie Linux, die Primär- und Sekundärreaktion in unterschiedlichen Systemschichten auszuführen.

Sie wählen diese Architektur, wenn Sie harte Echtzeitanforderungen, typischerweise kombiniert mit einer proprietären, auf die Anforderungen hin optimierten Hardware (meist Singlecore-CPU, Mikrocontroller) einsetzen wollen. Für den Einsatz ist das deterministische Zeitverhalten möglicherweise elementar, auf allerneueste Funktionalitäten können Sie verzichten.

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.

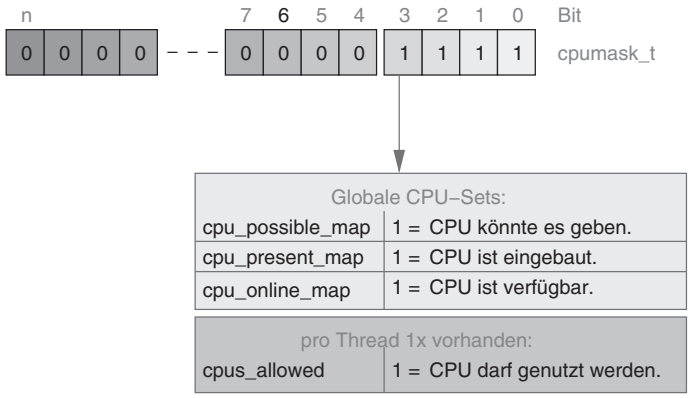


Abbildung 5-3 *Datenstruktur zur Verwaltung von mehreren Prozessorkernen*

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.

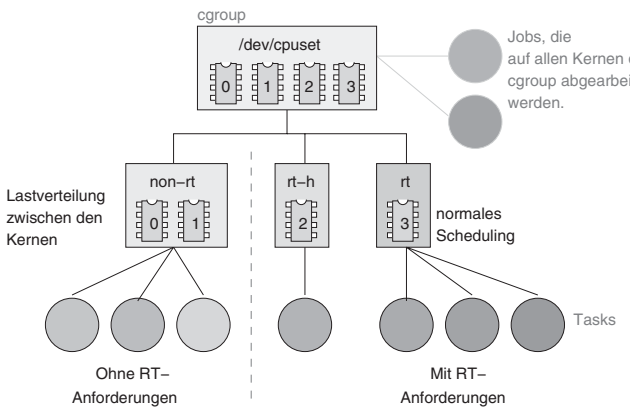


Abbildung 5-4
Hierarchische Prozessorganisation

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.

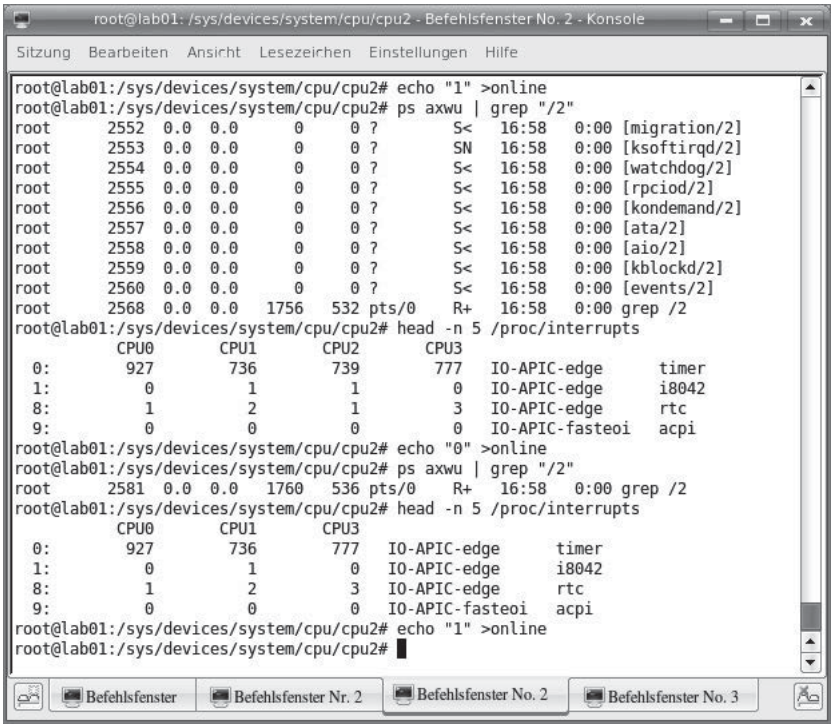


Abbildung 5-5
Deaktivieren von Rechnerkernen

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

es sich um eine Form der Virtualisierung, wobei der Realzeitkernel den Hypervisor darstellt. Der Realzeitkernel kontrolliert insbesondere das Sperren und Freigeben von Interrupts. Das Verfahren ist in der Implementierung zwar vergleichsweise aufwandsarm, nachteilig dabei ist aber, dass sämtliche Tasks, die unter dem Standardbetriebssystem laufen, nicht realzeitfähig sind. Insbesondere ist es nicht möglich, dass eine Task des Realzeitkerns eine Komponente des Standardbetriebssystems nutzt (z.B. TCP/IP), ohne dass diese Realzeit-Task ihre Echtzeitfähigkeit (ihren Determinismus) verliert. Implementierungen finden sich sowohl für Linux (RT-Linux, RTAI, Xenomai) als auch für Varianten des Windows-Betriebssystems.

RT-Linux

RT-Linux ist am *Department of Computer Science* der Universität New Mexico entwickelt worden. RT-Linux selbst ist zunächst ein kleiner Realzeitkern, der Linux in einer eigenen Task ablaufen lässt. Linux ist dabei die *Idletask*, die nur dann aufgerufen wird, wenn keine andere Realzeit-Task lauffähig ist.

Jeder Versuch der Linux-Task, Interrupts zu sperren (um selbst nicht unterbrochen zu werden), wird vom Realzeitkern abgefangen. Dazu emuliert der Realzeitkern die komplette Interrupt-Hardware. Versucht Linux selbst, einen Interrupt zu sperren, so merkt sich der Realzeitkern dies, sperrt den Interrupt jedoch nicht. Tritt nun der Interrupt auf, wird vom Realzeitkern entweder ein Realzeit-Interrupt-Handler (Realtime Handler) aufgerufen oder der Interrupt wird als *pending* markiert. Gibt die Linux-Task Interrupts wieder frei, emuliert RT-Linux den als *pending* markierten Interrupt für die Linux-Task. Damit können Interrupts die Linux-Task in jedem Zustand unterbrechen und Linux als solches ist nicht in der Lage, irgendwelche Latenzzeiten zu verursachen.

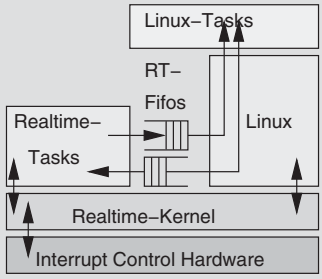


Abbildung 5-6 *RT-Linux-Systemarchitektur [Yod99]*

Interessant ist das Konzept, Standard- und Realzeitbetriebssystem parallel zu fahren, nur dann, wenn zwischen beiden ein Informationsaustausch möglich ist. RT-Linux bietet dazu zwei Möglichkeiten an:

1. Real-Time-Fifos und
2. Shared-Memory

Die Variante RT-Linux ist mit einem Patent versehen, sodass der Einsatz der Software bedenklich ist. Alternativ wird daher heute auf die Lösungen RTAI (Real Time Application Interface) oder Xenomai gesetzt, die dieses Problem nach derzeitigem Stand nicht mitbringen. Beide Varianten setzen primär auf eine Modifikation des Linux-Kernels namens Adeos.

Adeos/ipipe (Adaptive Domain Environment for Operating Systems) ermöglicht als sogenannter Nano-Kernel mehreren Betriebssystemen, die gleiche Hardware zu nutzen (Abbildung 5-7). Basisbegriffe sind dabei Domains und die Interrupt- beziehungsweise Event-Pipeline (I-pipe). Jedes Betriebssystem – im Realzeitumfeld sind dies typischerweise ein Realzeitkernel wie Xenomai und ein Standard-Linux – wird als Domain geführt. Die Root-Domain ist Linux. Jede Domain ist priorisiert, wobei die Root-Domain nicht die höchste Priorität haben muss. Für die Bearbeitung von Interrupts und Events stellt Adeos eine Interrupt-Pipe zur Verfügung (I-pipe). In dieser Pipe reihen sich die Domains gemäß ihrer Priorität ein. Ein Realzeitkernel wie RTAI oder Xenomai hat typischerweise sehr hohe Priorität und bekommt daher den Interrupt als Erstes zugestellt. Ist der Interrupt irrelevant für die jeweilige Domain, wird er von Adeos an die nächste Stufe der Pipeline weitergereicht.

Tabelle 5-1 *Von RTAI und Xenomai unterstützte Interfaces*

RTAI	Xenomai
POSIX 1003.1b	POSIX 1003.1b
RTDM	RTDM
	VxWorks
	PSoS+
	Vrtx

Die beiden Realzeitkernel RTAI und Xenomai nutzen Adeos, um neben dem Standard-Linux das Einhalten harter Realzeitanforderungen zu garantieren. Dabei unterscheiden sich die beiden Lösungen bezüglich ihrer Projektziele, der Interfaces und der unterstützten Hardware-Plattformen. Primäres Ziel unter RTAI ist es, möglichst kurze Latenzzeiten zu erreichen. Xenomai demgegenüber steht für hohe Portabilität. Entsprechend sind auch die unterstützten Schnittstellen vielfältiger. Während RTAI nur die beiden Interfaces RTDM (Realtime Driver Model) und POSIX 1003.1b unterstützt, hat Xenomai neben dem eigenen, nativen API noch sogenannte Skins für VxWorks, PSoS+ und Vrtx implementiert (Tabelle 5-1). Damit können Realzeitapplikationen, die beispielsweise für VxWorks geschrieben sind, mit geringem Aufwand unter Linux ablaufen.

RTAI unterstützt Hardware-Plattformen, die auf x86-, PowerPC-, Arm- oder Mips-Prozessoren bestehen; Xenomai unterstützt Arm, Blackfin, Nios II, PowerPC und x86.

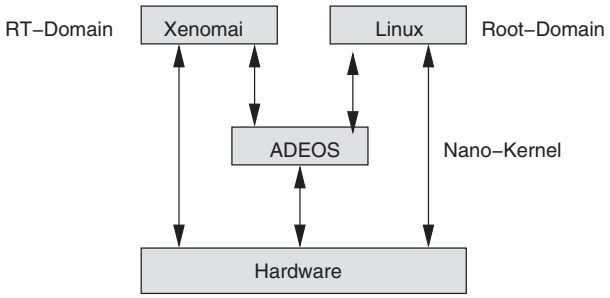


Abbildung 5-7
ADEOS

Ein Multikernel-Ansatz, wie ihn RTAI und Xenomai darstellen, hat den generellen Nachteil, dass gleich mehrere Betriebssysteme installiert, konfiguriert, aufeinander abgestimmt und vor allem gepflegt werden müssen. Je nach eingesetztem Betriebssystem und Version ist das eingesetzte Standardbetriebssystem weiterhin nicht realzeitfähig. Der Entwickler muss sich in zwei Systeme und deren Interfaces einarbeiten. Er teilt die zu lösende Aufgabe in die zeitlich kritischen und in die zeitlich unkritischen Bereiche ein. Typischerweise wird das Standardsystem (Linux oder auch Windows) für die Benutzerschnittstelle (GUI) eingesetzt.

Diese Architekturvariante wird vorwiegend im Bereich der eingebetteten Systeme eingesetzt, wenn harte Realzeitanforderungen im Mikrosekundenbereich erfüllt werden müssen. Auch wenn Multicore-Plattformen unterstützt werden, wird die Lösung primär für Singlecore-Anwendungen eingesetzt. Die Verknüpfung mit dem Standardsystem ermöglicht es dem Anwender, bekannte Benutzerschnittstellen/Oberflächen zu präsentieren und Realzeitdaten in normale Anwendungen (beispielsweise Office) zu integrieren.

5.8 Besonderheiten beim Entwurf moderner Realzeitsysteme

Der Entwurf eines Realzeitsystems ähnelt in seiner Struktur zunächst dem Entwurf eines Systems ohne zeitliche Anforderungen. Er orientiert sich typischerweise an einem V-Modell oder einem Spiralmodell, bei denen die einzelne Entwurfsphasen immer wieder in neuer Verfeinerung durchlaufen werden.

Unterschiede ergeben sich aufgrund der zusätzlichen, nicht funktionalen Anforderungen, beispielsweise bezüglich Realzeit, Safety, Security, Reliability, der oftmals notwendigen Parallelentwicklung von Hard- und

Software (Hardware/Software Codesign) und den Erschwernissen des Debuggens und Testens eingebetteter, zeitkritischer Software.

Anforderungen zusammenstellen

Die Anforderungen werden für die Anforderungsspezifikation (Requirement Specification) zusammengetragen. Neben den funktionalen Anforderungen müssen im Realzeitbereich die zeitlichen Anforderungen erfasst werden. Dazu sind die einzelnen Rechenzeitanforderungen als solche zu identifizieren, für jede Rechenzeitanforderung die minimal und die maximal zulässige Reaktionszeit (t_{Dmin} und t_{Dmax}), die maximale Auftrittshäufigkeit (minimale Prozesszeit t_{Pmin}) und schließlich die zeitlichen Abhängigkeiten untereinander, die durch die t_A ausgedrückt werden.

In der Kategorie sonstige Anforderungen sind im Realzeitumfeld vor allem Anforderungen an Safety und an Security zu definieren.

Der Entwurf

Zur Realisierung der funktionalen und zeitlichen Anforderungen sind die benötigten Hard- und Software-Komponenten auszuwählen beziehungsweise zu entwerfen.

Hierzu ist es sinnvoll, zunächst Funktionsblöcke zu definieren, die mit den funktionalen Anforderungen korrespondieren und denen dann zeitliche Anforderungen zugeordnet werden. Dabei sind bereits einige Entwurfsprinzipien zu beachten:

- Trennung von Oberfläche und Abläufen.
- Trennung der zeitkritischen von den nicht zeitkritischen Abläufen.
- Berücksichtigung von Sicherheitsaspekten, beispielsweise das Prinzip, einzelnen Codesequenzen nur die Privilegien zuzuordnen, die sie auch benötigen (Least Privilege).

Anhand der Funktionsblöcke kann die Hardware ausgewählt werden. Dabei schätzt der Entwickler bereits grob ab, ob die Funktionsblöcke in Realzeit verarbeitet werden können oder nicht. Ist das für einzelne Blöcke beispielsweise nicht möglich, kann über eine Realisierung des Blocks in Form von Hardware, beispielsweise von Field Programmable Gate Arrays (FPGA), nachgedacht werden. Alternativ muss eine generell leistungsfähigere Hardware in Betracht gezogen werden.

Für die Auswahl der Hardware ist darüber hinaus die Ankopplung von Peripherie relevant. Gerade im Bereich der eingebetteten Systeme werden häufig Feldbusse wie beispielsweise CAN (Controller Area Network) eingesetzt.

Hand in Hand mit der Hardware wird die Systemsoftware ausgewählt, schließlich muss diese die Hardware unterstützen. Unter Umständen ist zudem eine Portierung der Systemsoftware (Firmware, Bootloader, Betriebssystem) einzuplanen. Je nach Plattform sind ein Bootloader und das Betriebssystem (siehe Abschnitt 3.1) auszuwählen.

Moderne Realzeitsysteme setzen auf Multithreading. Die bereits identifizierten Funktionsblöcke werden demnach auf möglichst jeweils einzelne Threads abgebildet. Dadurch ergibt sich Inter-Thread-Kommunikation, die zu planen ist: Welche Daten werden wann zwischen welchen Einheiten wie ausgetauscht? Dem Schutz der sich dabei ergebenden kritischen Abschnitte ist besondere Aufmerksamkeit zu schenken.

Auf Basis der bisherigen Ergebnisse ist die Realzeitarchitektur zu entwerfen. Dabei muss nicht zwangsläufig eine der erwähnten Architekturen in Reinform vorkommen, sondern es sind durchaus Kombinationen denkbar, beispielsweise ein Multicore-Ansatz kombiniert mit Threaded Interrupts.

Im nächsten Schritt werden die Threads auf Prozessorkerne verteilt und systemweit Prioritäten vergeben. Anschließend können die Verarbeitungszeiten abgeschätzt oder aber ausgemessen werden. Idealerweise werden die Thread-Gruppen so auf die Kerne verteilt, dass sie untereinander keine gemeinsame Ressourcen haben. Dadurch treten keine Blockierzeiten zwischen Prozessorkernen auf und die Berechnung der Blockierzeiten (Abschnitt 8.3.1) bleibt gültig.

Mit Kenntnis der ursprünglichen Zeitanforderungen und der Verarbeitungszeiten ist der Realzeitnachweis für jede Rechenzeitanforderung und für jeden Prozessorkern getrennt durchzuführen. Können dabei einzelne Realzeitanforderungen nicht eingehalten werden, ist der Entwurf anzupassen. Eventuell führen andere Algorithmen zu einer kürzeren Laufzeit. Das Umverteilen der Jobs zwischen den Rechnerkernen bei einem Multicore-Ansatz könnte ebenso Abhilfe bringen wie eine leistungsstärkere Hardware. Vielleicht sind aber auch die zeitlichen Anforderungen doch nicht so streng, wie sie ursprünglich definiert wurden? Das Anpassen ist so lange durchzuführen, bis alle funktionalen und alle zeitlichen Anforderungen erfüllt werden können.

Als letzter Schritt der Entwurfsphase muss der Test geplant werden. Hierzu sind für jedes Modul Testpattern inklusive der erwarteten Ergebnisse aufzustellen. Auch für das Gesamtsystem ist ein Testplan zu entwerfen.

Zur Dokumentation des Entwurfes können beispielsweise Datenflussdiagramme (Abschnitt 7.1) eingesetzt werden, die einen Überblick über das Gesamtsystem geben. Die Abläufe innerhalb der einzelnen Funktionsblöcke werden per Struktogramm beschrieben (Abschnitt 7.2). Die Nebenläufigkeit des Systems kann schließlich mit einem Petri-

netz modelliert werden (Abschnitt 7.3). In vielen Fällen hat sich auch UML (Unified Modelling Language) etabliert (Abschnitt 7.5).

Die Realisierung

Zur Realisierung werden die definierten Funktionsblöcke in Code umgesetzt. Im Realzeitbereich wird dazu sehr häufig die Programmiersprache C verwendet, die allen Unkenrufen zum Trotz dank der Erzeugung sehr schnellen Codes nach wie vor am häufigsten im Bereich der Realzeitsysteme eingesetzt wird. Vorteilhaft ist vor allem, dass ein erfahrener Programmierer weiß, wie spezifische Sprachkonstrukte in Maschinencode umgesetzt werden. Dieses Wissen ist elementar, da moderne Compiler eine Reihe von Optimierungen vornehmen, die zu ungewöhnlichen und selten auftretenden Fehlern führen können. Für komplexere Applikationen bietet sich auch C++ an, das die Implementierung von objektorientierten Entwürfen unterstützt.

Volatile

Das Schlüsselwort volatile verhindert die Optimierung eines Variablenzugriffs durch den Compiler. Typischerweise gehen Compiler davon aus, dass sich Variablen außerhalb des aktuellen Programmkontextes nicht ändern. Insbesondere beim Zugriff auf Hardware ist das aber nicht korrekt. Hier kann sich der Wert eines Registers sehr wohl ändern. Durch das Schlüsselwort wird mit jedem Zugriff wirklich auf die dahinterliegende Speicherzelle zugegriffen und nicht nur auf die Kopie in einem Register.

Ein typisches Codefragment ist das Überprüfen eines bestimmten Zustandes:

```
static volatile int status = 0;

static void wait_for_hw( void )
{
     while (status==0)
             ;
}
```

Liegt eine Host-Target-Entwicklung vor, werden auf dem Host-System Cross-Entwicklungswerkzeuge in Form von Compiler, Linker, Bibliotheken und Debugger benötigt. Eventuell muss sogar zunächst eine komplette Entwicklungsumgebung gebaut werden. Im Linux-Umfeld bieten sich hierfür Generatoren wie `buildroot` oder `OpenEmbedded` an.

In jedem Fall wird eine Versionsverwaltung benötigt. Klassisch ist `svn`; aktuelle Projekte nehmen `git`.

Für den Komponententest wird auf die während des Entwurfs vorbereiteten Testfälle zurückgegriffen. Eventuell muss der Entwickler sich eine Testumgebung programmieren, die Funktionsblöcke mit den notwendigen Parametern aufruft und die Ergebnisse verifiziert.

Fehlersuche

Die Fehlersuche bei einem Realzeitsystem ist problematisch:

- Das Debuggen hat Einfluss auf das Zeitverhalten.
- Im Embedded-Umfeld ist häufig ein Remote-Debugging notwendig.
- Bei einigen Realzeitarchitekturen befinden sich Teile der Realzeitanwendung im Kernel, hier ist ein Kernel-Debugging notwendig.

Bei der Fehlersuche innerhalb des Realzeitsystems stellt sogenanntes Tracing manchmal die einzige Möglichkeit dar, zeitrelevante Fehler zu finden. Dazu können systemimmanente Tracing-Mechanismen oder selbst implementierte genutzt werden. Systemimmanente Tracing-Mechanismen protokollieren typischerweise die aufgerufenen Systemcalls, wie das folgende Beispiel des Kommandos `strace` zeigt:

Beispiel 5-2 *Systemimmanentes Tracing*

```
quade@felicia:~>strace echo "hello"
execve("/bin/echo", ["echo", "hello"], [/* 17 vars */]) = 0
brk(0)                                  = 0x1b74000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f9ef66e4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=134707, ...}) = 0
mmap(NULL, 134707, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9ef66c3000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\360\1\0\0\0\0\0"...,
832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1638120, ...}) = 0
...
```

Über die Option `-p` kann sich `strace` an bereits laufende Threads binden und die Systemcalls mitprotokollieren.

Noch mehr Möglichkeiten bieten eigene Traces, die über das Einstreuen von Ausgaben (`fprintf`) in den Quellcode realisiert werden. Sinnvoll ist es, jede Ausgabe mit einem Zeitstempel zu versehen, eventuell auch den Namen der Quellcodedatei und die zugehörige Zeilennummer mit ausgeben zu lassen. Der Zeitstempel ermöglicht den zeitlichen Abgleich, wenn später Traces von unterschiedlichen Threads zusammengeführt werden müssen (Beispiel 5-3). Sinnvollerweise stellt der Entwickler zudem direkt ein Trace-Makro zur Verfügung, das per Define zum Generierungszeitpunkt entweder ein- oder ausgeschaltet werden kann.

Beispiel 5-3 *Selbst implementiertes Tracing*

```
#include <stdio.h>
#include <sys/time.h>

int main( int argc, char **argv, char **envp )
{
   struct timeval tv;

   gettimeofday( &tv, NULL );
   fprintf(stderr,"%ld s %ld us: %s %s line %d\n",
      tv.tv_sec, tv.tv_usec,
      __FILE__, __FUNCTION__, __LINE__ );
   return 0;
}
```

Oftmals gibt es im Bereich der eingebetteten Systeme keine Möglichkeit, per `printf()` Ausgaben durchzuführen. Hier muss sich der Entwickler mit der primitiven Form zufriedengeben, bei der eine oder auch mehrere LEDs programmgesteuert den aktuellen Zustand reflektieren. Stehen auch keine LEDs zur Verfügung, kann manchmal das Ausgangssignal einer Output-Leitung angesteuert werden. Per Oszilloskop lassen sich auf diese Art sogar ausgesprochen genaue Differenzzeitmessungen durchführen.

Traces werden unter dem Stichwort Kernel-Event-Tracer häufig auch eingesetzt, um die Interaktion zwischen Applikation und Kernel (ISRs, Treiberfunktionen) deutlich zu machen. VxWorks bietet hierfür beispielsweise das Werkzeug WindView an.

Je nach Applikation und Fehlerbild lässt sich auch ein normaler Debugger, wie ihn unter Linux der `gdb` darstellt, einsetzen. Ähnlich wie `strace` kann der `gdb` ebenfalls die Kontrolle von bereits gestarteten Threads durch Angabe der Thread-ID übernehmen.

Hilfreich ist die Möglichkeit des Post-Mortem-Debuggens. Durch geeignete Konfiguration wird das System dazu veranlasst, im Fall eines Programmabsturzes einen Speicherabzug der von der Task verwendeten Speichersegmente anzulegen, den sogenannten *core*. Zur Freischaltung und Konfiguration wird in den Userlimits per `ulimit`-Kommando die maximale Größe der Speicherabzugsdatei (Core File Size) in Blöcken festgelegt, so wie es in Beispiel 5-4 zu sehen ist.

Beispiel 5-4 *Post-Mortem-Debugging*

```
quade@felicia:/tmp>ulimit -c 100000
quade@felicia:/tmp>./faulty_app
Speicherzugriffsfehler (Speicherabzug geschrieben)
quade@felicia:/tmp>gdb faulty_app core
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
...
```

```
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Core was generated by `./faulty_app'.
Program terminated with signal 11, Segmentation fault.
#0  0x00000000004004cf in main (argc=1, argv=0x7ffff2163e98,
    envp=0x7ffff2163ea8) at faulty_app.c:7
7               c = *ptr;
(gdb)
```

Befinden sich Teile der Realzeitapplikation im Kernel (Userland-to-Kernel-Architektur), muss ein Kernel-Debugging durchgeführt werden. Da hierbei der Kernel angehalten werden muss, steht keine Ablaufumgebung mehr zur Verfügung, die Tastatureingaben entgegennimmt, Daten auf den Monitor zaubert, auf Speicherinhalte zugreift oder die den Kernel später weiterarbeiten lässt.

Technisch wird dieses Problem dadurch gelöst, dass komplexe Funktionen auf ein zweites System ausgelagert werden. Dieses hat ein funktionierendes Speicher- und Dateimanagement und übernimmt das Durchwühlen des Quellcodes nach Variablen, Datenstrukturen, Funktionen und Codezeilen. Für den zu debuggenden Kernel wird nur noch ein sogenannter Debug-Server benötigt, der einfache Kommandos, wie beispielsweise Speicherzellen lesen, schreiben oder Breakpoints setzen, im zu untersuchenden System ausführen kann.

Im Linux-Kernel stellt der `kgdb` einen solchen Debug-Server dar. Dieser kommuniziert mit einem Frontend (Debug-Host) über die serielle Schnittstelle, mit der Target und Host ausgestattet sein müssen. Als Frontend wird `gdb` eingesetzt. Damit Kernel-Debugging funktioniert, wird der zum Zielsystem passende `gdb` ebenso benötigt wie der Code des Betriebssystemkerns inklusive Symbolinformationen und der Quellcode von Kernel und zu debuggenden Komponenten.

Linux bietet mit dem `kdb` alternativ ein lokales Frontend an, das zwar bereits komplexere Kommandos ausführen kann, aber kein Debugging auf Hochsprachenniveau ermöglicht. Mit dem `kdb` lassen sich Breakpoints setzen, Speicherzellen auslesen und modifizieren, der Stack-Trace ausgeben oder die Liste der Jobs anzeigen. Um den `kdb` einzusetzen, ist dieser zunächst freizuschalten. Dazu dient das folgende Kommando:

```
root@host:~# echo kms,kbd > /sys/module/kgdboc/parameters/kgdboc
```

So scharf gestellt wird ein Kernel-Panic ebenso in den Debugger springen wie das Aktivieren über eine magische Tastatursequenz oder der schreibende Zugriff auf die Datei `/proc/sysrq-trigger`:

```
root@host:~# echo g >/proc/sysrq-trigger
```

Genauere Informationen zum Umgang mit dem Kernel-Debugger, insbesondere auch in einer virtualisierten Umgebung, finden sich unter [QuKu2012-62].

Bei einer Host-Target-Entwicklung wird, falls vom Zielsystem unterstützt, ebenfalls ein Remote-Debugger eingesetzt. Auf dem Zielsystem (Target) läuft der Debug-Server, der seine Kommandos meist über eine serielle Schnittstelle vom Debug-Host bekommt, ausführt und dann die Ergebnisse zurück an den Debug-Host sendet.

Der Debug-Host benötigt neben dem Debugger-Frontend, der Code der jeweiligen Zielplattform debuggen kann (Cross-Debugger), die zu untersuchenden Programme (Executables), die Objektdateien und den zugehörigen Quellcode.

Je eher Fehler in der Nähe der Hardware gesucht werden, desto eher kommen auch hardwaregestützte Debugging-Verfahren in Betracht, insbesondere Incircuit-Emulatoren (ICE) und JTAG (Joint Test Action Group).

Primär ist JTAG entwickelt worden, um integrierte Schaltungen (IC) auf ihre Funktion testen zu können, wenn diese bereits auf der Zielhardware verbaut wurden. Integrierte Schaltungen mit JTAG-Interface besitzen zusätzliche Schaltungsteile, die im Normalbetrieb völlig abgetrennt von den anderen Funktionsblöcken des IC sind. Diese Schaltungsteile können über eigene Anschlussleitungen (PIN) aktiviert werden. In diesem Fall ist es möglich, die Hardware des IC zu kontrollieren.

Außer zur Überprüfung der Hardware lässt sich JTAG im Bereich der eingebetteten Systeme einsetzen für:

- Programmierung von Flash-Speicher
- Einzelschritt-Debugging (Breakpoints setzen/rücksetzen, run/stop, single step, watchpoints setzen)
- Testen der digitalen Ein- und Ausgänge

JTAG ist damit auch eine einfache Form eines Incircuit-Emulators. Bei einem ICE wird ein Baustein in der Schaltung, typischerweise die CPU, durch den Emulator ersetzt. Dieser ist in der Lage, die CPU mit all ihren Signalen zu emulieren. Um dies zu realisieren, setzt der ICE häufig eine Spezialvariante der zu debuggenden CPU ein. Mit einem ICE kann man sich darüber hinaus einen Überblick über CPU-interne Zustände verschaffen.

Da es sich bei dem Incircuit-Emulator um sehr leistungsfähige Hardware handelt, ist der Debugger zum einen sehr spezifisch und zum anderen sehr teuer. Das schränkt seine Verbreitung ein.

Das Zeitverhalten einer Realzeitapplikation kann auch per Profiling ausgemessen werden. Unter Profiling versteht man das Ausmessen des Zeitverhaltens einer Applikation. Es wird gemessen, wie häufig eine

Funktion aufgerufen wird und wie lang die Abarbeitung einer Funktion dauert.

Profiling wird typischerweise über zwei Technologien realisiert. Bei der Instrumentalisierung wird der Code modifiziert, sodass die gesuchten Informationen beim Ablauf vom Code selbst generiert werden. Damit hat das Profiling einen – allerdings geringen – Einfluss auf das Zeitverhalten. Bei der zweiten Methode wird in regelmäßigen, kurzen Abständen der Program Counter (PC) der CPU ausgelesen. Aus der dort abgelegten Adresse kann berechnet werden, welche Funktion gerade abgearbeitet wird. Die Information lässt auch Rückschlüsse über die Laufzeit der Funktion zu.

Unter Linux wird die Instrumentalisierung durch die Option `-pg` beim Kompilieren und Linken aktiviert. Die Auswertung schließlich erfolgt mit dem Programm `gprof`.

Da bei parallelen Anwendungen Fehler nicht immer durch normales, einfaches Debuggen gefunden werden können, bedarf es spezieller und für Parallelisierung optimierter Werkzeuge. Um die Anwendung auch auf ihre Skalierbarkeit zu testen und später zu optimieren, sollte man die Anwendung auf möglichst unterschiedlichen Systemen testen.

Profiling bietet auch das Werkzeug `callgrind` aus dem `Valgrind`-Paket, eine aus dem Open Source stammende Werkzeugsammlung. Valgrind selbst wird für die Suche nach Speicherlecks genutzt, bietet darüber hinaus aber auch mit callgrind und helgrind zwei Tools an, die bei Synchronisationsproblemen eingesetzt werden können.

Callgrind ist ein sogenanntes Profiling-Werkzeug, welches Funktionsaufrufe innerhalb einer Anwendung protokolliert und als Aufrufgraph (call-graph) ausgibt. Damit lässt sich der Programmpfad genau verfolgen. Mit Helgrind lassen sich Synchronisationsprobleme in parallelen Anwendungen finden. Dazu zählen Deadlocks, Race Conditions oder der fehlerhafte Einsatz der POSIX Pthread API an sich wie z.B.:

- Freigeben eines invaliden oder zuvor nicht angeforderten Locks,
- rekursives Anfordern eines Mutex.

Sychronisationsfehler in einer Anwendung zu finden ist sehr schwierig, da diese nur unter bestimmten Bedingungen auftreten. Auf dem Entwicklungsrechner treten sie unter Umständen nie oder nur sehr selten auf und aus ein oder mehreren korrekt funktionierenden Programmläufen kann nicht belastbar auf eine Fehlerfreiheit geschlossen werden.

Tools wie helgrind helfen dem Entwickler zwar bei der Suche nach Synchronisationsfehlern, können aber keine vollständige Testabdeckung realisieren. Auch sind False-Positiv-Meldungen dieser Tools nicht selten. Für die Analyse wird also ein erfahrener Entwickler benötigt.

Alle Valgrind-Werkzeuge unterstützen von Haus aus Threads, sofern die native Threading-Bibliothek von Linux zur Entwicklung verwendet wurde (POSIX-Threads).

Die Integration

Bei der Integration werden die unter Umständen unabhängig voneinander realisierten Komponenten auf dem Zielsystem installiert und konfiguriert.

Der Systemtest

Der abschließende Systemtest soll sicherstellen, dass sowohl die funktionalen als auch die zeitlichen Anforderungen eingehalten werden. Dazu werden die Testfälle, die bereits beim Entwurf festgelegt wurden, abgearbeitet.

Um die zeitlichen Anforderungen zu überprüfen, wird das Realzeitsystem unter verschiedenen Lastsituationen, insbesondere unter hoher Last, getestet.

Last kann auf zweierlei Arten erzeugt werden:

1. Verarbeitet das Realzeitsystem Signale aus einem technischen Prozess, so wird versucht, eine Situation herbeizuführen, bei der möglichst sämtliche Signale gleichzeitig mit höchster Frequenz auftreten.
2. Das Realzeitsystem wird mit einem (nutzlosen) Hintergrundprozess belastet. Hierbei ist zu differenzieren, ob eine CPU-Last, eine Netzwerklast, eine Last des Dateisystems oder alles zusammen erzeugt werden soll.

Die CPU kann sehr gut mithilfe eines Benchmarks wie Prime95 unter Last gesetzt werden. Die Software hierfür ist unter [http://www.mersenne.org] downloadbar. Eine einfachere Lösung ist es, den Prozessor beispielsweise rechnen (Beispiel 5-5) oder den Kernel Zufallszahlen bestimmen zu lassen (Beispiel 5-6). Bei Letzterem werden vom Kernel diverse Hashoperationen durchgeführt. Wichtig ist, die Zufallszahlen über `/dev/urandom` auszulesen. Beim Lesen von `/dev/random` blockiert der Kernel den zugreifenden Thread, wenn er die Zufälligkeit der Zahlen nicht garantieren kann. Wird ein solches Skript zigfach parallel gestartet, lässt sich auch eine Multicore-Maschine auslasten.

```
#!/bin/bash
while true
do
   a=$((a*2));
done
```

Beispiel 5-5
Lasterzeugung durch Berechnungen im Userland

Beispiel 5-6 *Lasterzeugung durch Berechnungen im Kernel*

```
#!/bin/bash

cat /dev/urandom >/dev/null
```

Eine grobe Aussage über die CPU-Last liefert das Programm `uptime` mit den Werten, die unter *load average* aufgeführt werden. Vereinfacht dargestellt gibt der erste Wert die Anzahl Jobs wieder, die zu einem Zeitpunkt rechenbereit sind. Auf einer Singlecore-Maschine kann daher eine Last von eins abgearbeitet werden, auf einer Quadcore-Maschine eine Last von vier. Um eine reale Lastsituation zu erzeugen, sollte die Last ein Vielfaches der Anzahl vorhandener CPU-Kerne betragen.

Eine hohe Last im Netzwerksubsystem lässt sich mithilfe des Kommandos `netcat` realisieren. Per `netcat` werden im einfachsten Fall Nullen von einem (Client) zum anderen Rechner (Server) kopiert. Auf dem Server wird dazu `netcat` mit der Option `-l <portadresse>` gestartet. Da die übertragenen Daten typischerweise auf den Bildschirm ausgegeben werden, sollten Sie diese durch geschicktes Umleiten wegwerfen.

```
user@server:~>netcat -l 12345 >/dev/null
```

Auf dem Client liest `netcat` die Daten vom normalen Eingabeinterface. Das lässt sich nutzen, um beispielsweise unendlich viele Nullen zu übertragen.

```
user@client:~>cat /dev/zero | netcat <serverip> 12345
```

Festplattenzugriffe lassen sich auf einem Unix-artigen System mithilfe des Kommandos `dd` erzeugen. Beispielsweise wird in einer Endlosschleife die komplette Festplatte (lesen von `/dev/sda`) ausgelesen. Die Daten werden typischerweise direkt wieder verworfen (Ausgabe nach `/dev/null`), um die Festplatte nicht unnötig zu beschreiben. Eventuell wird das Kommando zeitversetzt und mehrfach parallel gestartet.

```
root@client:~>dd if=/dev/sda of=/dev/null bs=4k
```

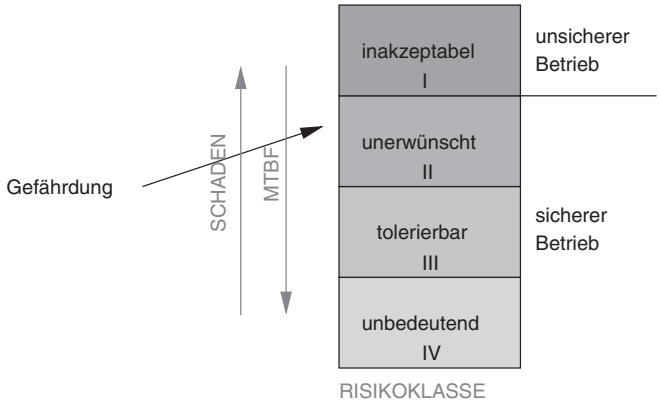
Sollen Schreibzugriffe erzeugt werden, kann dies ebenfalls mit `dd` geschehen. Die zu schreibenden Daten dürfen jedoch nicht aus Datenblöcken bestehen, die nur Nullen enthalten. Diese werden nämlich gesondert behandelt. Ebenso muss verhindert werden, dass die Daten nur im Cache verbleiben und auch dass durch die Lasterzeugung die Festplatte nicht voll geschrieben wird (wenngleich das zu testende Realzeitsystem auch mit dieser Situation zurechtkommen muss).

6 Safety und Security

Realzeitsysteme werden häufig in sicherheitskritischen Anwendungen eingesetzt. Ein Ausfall oder ein Fehlverhalten des Realzeitsystems bedroht dann Menschen, Anlagen, Produktionsgüter und die Umwelt und führt zu hohen Kosten. Daher ist es notwendig, das System so zu konstruieren, dass von ihm keine Bedrohung ausgeht. Das System muss betriebssicher sein. Man spricht allgemein von der Betriebssicherheit beziehungsweise im englischsprachigen Raum von Safety.

Während die Betriebssicherheit den Schutz der Umgebung vor dem System sicherstellt, handelt es sich bei Angriffssicherheit (englisch Security) um den Schutz des Systems vor der Umgebung. Es muss sichergestellt werden, dass unbefugte Personen keinen Zugriff auf die Systeme bekommen, keine sensiblen Informationen ausspionieren oder beispielsweise eine Anlage in einen ungewünschten Zustand versetzen.

6.1 Grundbegriffe der Betriebssicherheit (Safety)



MTBF Häufigkeit des Auftretens (Maß der Zuverlässigkeit)
SCHADEN Schwere der Auswirkungen

Abbildung 6-1
Einsortierung von Gefährdungen in Risikoklassen

Realzeitsysteme dürfen ihre Umwelt nicht gefährden und müssen daher betriebssicher ausgelegt werden. Die Betriebssicherheit selbst ist nach

der zwischenzeitlich zurückgezogenen DIN/VDE 31000-2 definiert als »eine Sachlage, bei der das Risiko nicht größer als das Grenzrisiko ist«. Als Grenzrisiko ist dabei das größte, noch vertretbare Risiko zu verstehen. Man spricht auch vom Restrisiko beziehungsweise vom akzeptierten Risiko. Dabei wird das Risiko typischerweise in Risikoklassen eingeteilt (zum Beispiel Risikoklasse I inakzeptabel, II unerwünscht, III tolerierbar und IV unbedeutend). In welche Klasse eine identifizierte Gefährdung fällt, wird auf Basis der Häufigkeit des Auftretens und der Schwere der Auswirkungen mithilfe der sogenannten Fehlerbaumanalyse festgelegt [Winne2009] (Abbildung 6-1).

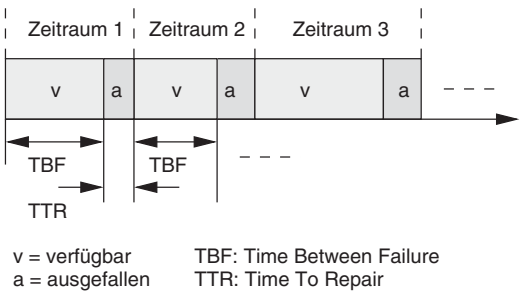
Einer Risikoklasse wiederum ist vereinfacht dargestellt eine Ausfallwahrscheinlichkeit zugeordnet, in der das System nicht verfügbar (unverfügbar) ist oder aber eine Fehlfunktion aufweist.

Dabei kann sich gemäß der Theorie ein Realzeitsystem nur in einem von den beiden Zuständen befinden: Es ist nicht verfügbar oder es ist verfügbar.

Verfügbarkeit. Wahrscheinlichkeit, dass ein System innerhalb eines spezifizierten Zeitraums funktionstüchtig (verfügbar) ist. Die Verfügbarkeit wird mit dem Buchstaben `p` gekennzeichnet.

Unverfügbarkeit. Wahrscheinlichkeit, dass ein System innerhalb eines spezifizierten Betrachtungszeitraums funktionsuntüchtig (unverfügbar) ist. Die Unverfügbarkeit wird mit dem Buchstaben `q` gekennzeichnet. Sie wird auch Ausfallwahrscheinlichkeit genannt.

Abbildung 6-2
Zeitverlauf des Systemzustandes



Der Betrachtungszeitraum, in dem ein System verfügbar ist, wird als *Time Between Failure* (TBF) bezeichnet. Der Betrachtungszeitraum, in dem ein System unverfügbar ist, wird als *Time To Repair* (TTR) oder auch Reparaturzeit bezeichnet (Abbildung 6-2).

Summiert man die Zeiträume, in denen das System verfügbar ist, auf und bildet durch Division mit der Anzahl der Zeiträume das arithmetische Mittel, erhält man die *Mean Time Between Failure* (MTBF). Das Gleiche gilt für die Reparaturzeiten (Gleichung 6-1). Das arithmetische Mittel wird hier *Mean Time To Repair* (MTTR) genannt.

$$MTBF = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} TBF_i$$

$$MTTR = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} TTR_i$$

Gleichung 6-1
MTBF und MTTR

Der Kehrwert der MTBF ist die Ausfallrate λ, der Kehrwert der MTTR die Reparaturrate ρ.

$$\lambda = \frac{1}{MTBF} \; ; \qquad \rho = \frac{1}{MTTR}$$

Gleichung 6-2
Ausfall- und Reparaturrate

Unter Kenntnis der MTBF und der MTTR beziehungsweise der Ausfallrate λ und der Reparaturrate ρ lässt sich gemäß der Formeln aus Gleichung 6-3 die Dauerverfügbarkeit, also die Wahrscheinlichkeit, dass sich das System im Zustand verfügbar befindet, ausrechnen. Das Gleiche gilt für die Dauerunverfügbarkeit.

$$p = \frac{MTBF}{MTTR + MTBF} \quad \text{(Verfügbarkeit)}$$

$$q = \frac{MTTR}{MTTR + MTBF} \quad \text{(Unverfügbarkeit)}$$

$$\text{mit} \quad p + q = 1$$

Gleichung 6-3
Dauerverfügbarkeit und Dauerunverfügbarkeit

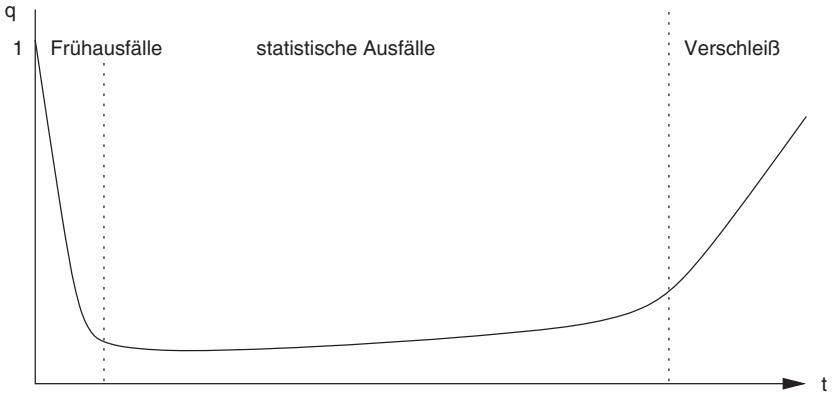


Abbildung 6-3
Badewannenkurve

Die Dauerverfügbarkeit steht für die Verfügbarkeit über einen (unendlich) langen Zeitraum hinweg. Das ist ungenau. In der Realität hängt die Verfügbarkeit von der Zeit ab. Generell unterscheidet man drei Ausfallphasen, wie in Abbildung 6-3 zu sehen ist. Dabei ist die Wahrscheinlichkeit für einen Ausfall bei einem neuen System besonders hoch. Man

spricht dabei von den Frühausfällen. Läuft ein neues System jedoch erst einmal fehlerfrei, fällt es zunächst nur noch mit geringer Wahrscheinlichkeit aus, was als Phase der statistischen Ausfälle bezeichnet wird. Mit zunehmender Alterung und damit einhergehendem Verschleiß kommt es gegen Ende der Lebensdauer eines Systems wieder zu vermehrten Ausfällen, den sogenannte Verschleißausfällen.

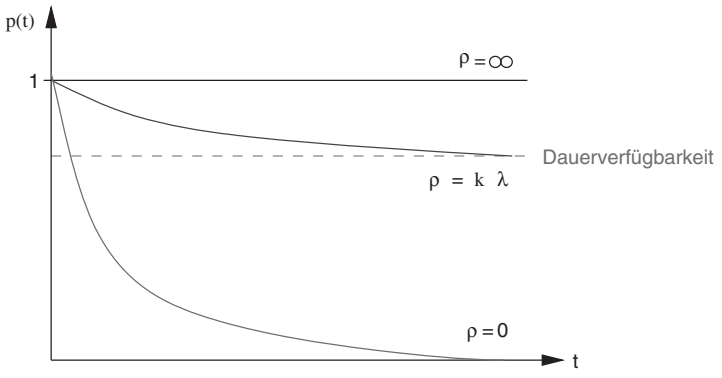
In der Phase der statistischen Ausfälle lässt sich die Verfügbarkeit beziehungsweise die Unverfügbarkeit in Abhängigkeit von der Zeit gemäß Gleichung 6-4 berechnen.

Gleichung 6-4
Verfügbarkeit und Unverfügbarkeit in Abhängigkeit von der Zeit

$$p(t) = \frac{\rho_i}{\rho+\lambda} + \frac{\lambda}{\rho+\lambda} e^{-(\rho+\lambda)t}$$
$$q(t) = 1 - p(t)$$

Abbildung 6-4 visualisiert die Verfügbarkeitsfunktion p(t) grafisch. Bei einer normalen Reparaturrate stellt sich auf lange Zeit betrachtet die Dauerverfügbarkeit ein. Wird das System nicht repariert, ist es sehr wahrscheinlich, dass das System nach langer Zeit ausgefallen ist. Wird das System quasi in Nullzeit repariert (Reparaturrate ist unendlich), bleibt die Verfügbarkeit bei hundert Prozent.

Abbildung 6-4
Verfügbarkeit für ausgewählte Reparaturraten



Reicht die berechnete Verfügbarkeit zur Erreichung einer vorgegebenen Risikoklasse nicht aus, müssen risikomindernde Maßnahmen ergriffen werden. Diese können organisatorischer Art (beispielsweise vorbeugende Wartung) oder aber auch technischer Art (beispielsweise Materialauswahl oder Redundanzkonzepte) sein.

Betriebskonzepte

Abhängig von der technischen Realisierung lassen sich Systeme, die Redundanzen aufweisen, unterschiedlich betreiben. Falls bei einem Ausfall keine oder kaum Schäden zu erwarten und vielleicht sogar tolerierbar

sind, werden Redundanzen für die Steigerung der Verfügbarkeit eingesetzt. Sind die Schäden demgegenüber nicht tolerierbar, werden Redundanzen zur Steigerung der Betriebssicherheit eingesetzt und das System in den sogenannten sicheren Zustand überführt. Das ist beispielsweise bei einem Flugzeug der Fall, dessen kritische Steuersysteme mehrfach vorhanden sind. Fällt eines der redundanten Systeme aus, wird der sicherheitsbewusste Pilot nicht weiter fliegen, sondern den nächsten Flughafen ansteuern und landen, um wieder einen sicheren Zustand (*Fail Safe*) zu erreichen. Der Betreiber eines Webshops allerdings wird das redundante System nutzen, um den Betrieb fortzusetzen und damit die Ausfallzeit zu überbrücken. Natürlich lassen sich durch geeignete Redundanzen sowohl die Betriebssicherheit als auch die Verfügbarkeit erhöhen. In diesem Fall spricht man von *Fail Operational*.

Sind die Verfügbarkeiten redundanter Komponenten bekannt, kann die Gesamtverfügbarkeit des Systems abhängig davon, ob es im Hinblick auf Sicherheit oder im Hinblick auf Verfügbarkeit betrieben wird, ausgerechnet werden.

Verfügbarkeitsbetrieb

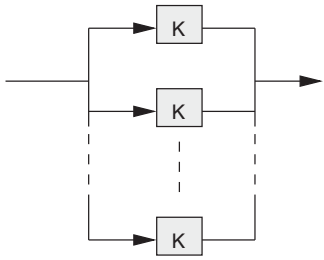


Abbildung 6-5
Verfügbarkeitsbetrieb

Fällt beim Verfügbarkeitsbetrieb ein System aus, wird es durch ein redundantes System ersetzt und der Betrieb wird fortgesetzt. Unabhängig von der physikalischen Realisierung wird dieser Umstand in einem sogenannten Verfügbarkeitsersatzschaltbild als Parallelschaltung (siehe Abbildung 6-5) dargestellt.

Die Verfügbarkeit im Fall des Verfügbarkeitsbetriebs wird über die Unverfügbarkeit der einzelnen Komponenten berechnet (siehe Gleichung 6-5), wobei die Unverfügbarkeiten miteinander multipliziert werden.

Gleichung 6-5
Verfügbarkeit beim Verfügbarkeitsbetrieb

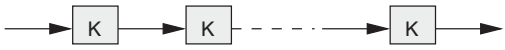
$$q_{gesamt} = \prod_{i=1}^{n} q_i$$

$$p_{gesamt} = 1 - q_{gesamt}$$

Sicherheitsbetrieb

Beim Sicherheitsbetrieb wird das redundante System genutzt, um bei einem Ausfall das System in den sicheren Zustand zu überführen (fail safe). Unabhängig von der physikalischen Realisierung wird dieser Umstand im Verfügbarkeitsersatzschaltbild als Serienschaltung (siehe Abbildung 6-6) dargestellt.

Abbildung 6-6
Sicherheitsbetrieb



Die Verfügbarkeit berechnet sich beim Sicherheitsbetrieb über das Produkt der Einzelverfügbarkeiten (siehe Gleichung 6-6). In diesem Fall entspricht außerdem die Gesamtausfallrate der Summe der Einzelausfallraten.

Gleichung 6-6
Verfügbarkeit beim Sicherheitsbetrieb

$$p_{gesamt} = \prod_{i=1}^{n} p_i$$

$$q_{gesamt} = 1 - p_{gesamt}$$

$$\lambda_{gesamt} = \sum_{i=1}^{n} \lambda_i$$

6.2 Angriffssicherheit (Security)

Realzeitsysteme sind zunehmend vernetzt und auch die Realzeitsysteme, die technische Anlagen steuern, sind nicht nur durch firmeninterne Netze, sondern sogar zum Zweck der Fernwartung über das Internet erreichbar. Doch auch nicht vernetzte Realzeitsysteme werden zum Angriffsziel, wie der Computervirus Stuxnet, der 2010 den Betrieb iranischer Atomanlagen massiv störte, eindrucksvoll zeigt. Daher sind Realzeitsysteme vor äußeren Zugriffen durch geeignete Maßnahmen zu schützen.

Realzeitsysteme sind gefährdet durch

- Sabotage,
- Spionage und
- Missbrauch.

Durch geeignete Maßnahmen müssen also die Verfügbarkeit, die Vertraulichkeit und die Integrität (die drei Schutzziele) gewährleistet werden. Derartige Schutzmaßnahmen müssen in das Realzeitsystem integriert werden. Man spricht von geräteimmanenten Schutzvorrichtungen. Diese sollten darüber hinaus durch strukturelle Abwehrmaßnahmen ergänzt werden. Dies sind Maßnahmen, die sich durch die Integration des Realzeitsystems in das Gesamtsystem, beispielsweise in das Firmennetz oder durch die Anbindung an das Internet, ergeben.

Betreiber von Realzeitsystemen haben zumeist auf die geräteimmanenten Schutzvorrichtungen keinen Einfluss und müssen daher verstärkt auf strukturelle Abwehrmaßnahmen setzen.

6.2.1 Geräteimmanente Schutzvorrichtungen

Die geräteimmanenten Schutzvorrichtungen moderner Realzeitsysteme bestehen aus folgenden Maßnahmen:

- Authentifizierung
- Verschlüsselung von Daten und Kommunikation
- Filtern eingehender und abgehender Datenströme über eine (eingebaute) Firewall
- minimalistischer Funktionsumfang
- Isolationstechniken

Authentifizierung

Zugang zu kritischen Teilen des Realzeitsystems sollen nur autorisierte Personen erhalten. Dazu ist eine Authentifizierung des Users notwendig. Klassischerweise wird hierzu häufig eine Username-/Passwort-Authentifizierung verwendet, die auf der Idee beruht, dass nur die berechtigte Person das Passwort kennt.

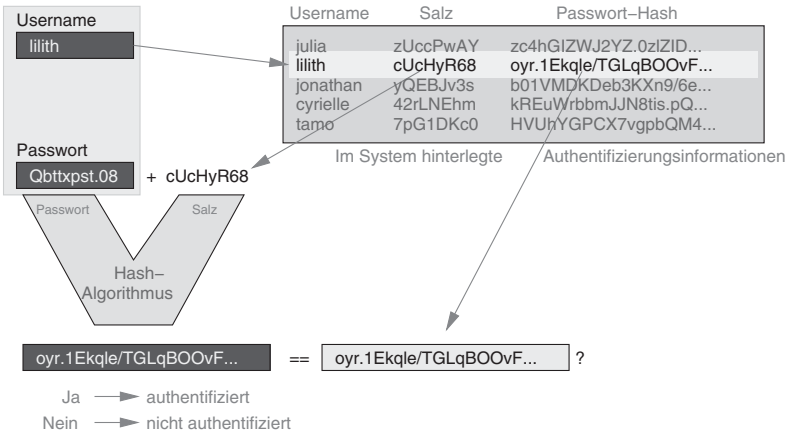


Abbildung 6-7 *Authentifizierung mithilfe von Passwort-Hashes*

Bei der technischen Realisierung dieser Maßnahme gibt es Folgendes zu beachten:

- Es darf kein Default-Passwort im System hinterlegt werden. Ein solches Default-Passwort wird früher oder später bekannt und hebelt den kompletten Authentifizierungsmechanismus aus.
- Die geheimzuhaltende Authentifizierungsinformation (das Passwort) darf auf keinen Fall im Klartext im System hinterlegt werden! Stattdessen wird das Passwort zusammen mit einem sogenannten Salz gehasht. Mit *Salz* wird ein meist zufälliger String bezeichnet, der an das Passwort angehängt wird und dieses damit verlängert. Das jeweils eingesetzte Salz muss man sich natürlich merken (abspeichern). Zum Hashen wird eine Hashfunktion eingesetzt, die das Passwort zusammen mit dem Salz auf eine eindeutige, endliche Bitfolge (zum Beispiel 160 Bit), dem Passwort-Hash, abbildet. An die Hashfunktion werden zwei Anforderungen gestellt. Erstens darf aufgrund der Bitfolge kein Rückschluss auf Passwort und Salz möglich sein (keine Umkehrbarkeit). Zweitens muss die Hashfunktion kollisionsfrei sein, sodass zwei unterschiedliche Passwort-Salz-Kombinationen auch auf zwei unterschiedliche Hashwerte abgebildet werden.
- Die Verwendung der Passwort-Hashes stellt sicher, dass ein Angreifer, der Zugriff auf die Authentifizierungsdatenbank hat, nicht direkt sämtliche Klartextpasswörter kennt. In diesem Fall bleibt ihm nur der Versuch, die Passwörter per Brute-Force-Angriff zu knacken. Das typischerweise zufällig gewählte Salz verlängert ein Passwort und erschwert damit den Brute-Force-Angriff.
- Die Authentifizierung per Username/Passwort ist in vier Schritten aufzubauen (Abbildung 6-7):

1. Mit dem vom User eingegebenen Namen wird die zum Namen gehörende Authentifizierungsinformation ausgelesen. Diese besteht aus dem Namen, dem Salz und dem zugehörigen Hashwert.
2. Aus dem vom User eingegebenen Passwort und dem Salz wird ein String gebildet. Dieser String wird gehasht und ergibt den Passwort-Hash.
3. Direkt anschließend muss zunächst der Speicher, in dem das Passwort im Klartext abgelegt ist, mit Zufallswerten überschrieben werden.
4. Der aus Salz und Passwort berechnete Passwort-Hash wird mit demjenigen aus der Authentifizierungsinformation verglichen. Bei Gleichheit ist der User als derjenige, der das Passwort kennt, authentifiziert. Andernfalls wird der Zugriff verweigert.

Basiswissen Kryptografie

Eine wichtige Methode der Spionageabwehr ist die Verschlüsselung von Daten, die Kryptografie.

Bei der Verschlüsselung wird eine Information, oft auch als Klartextnachricht bezeichnet, mit einem Schlüssel, dem Key, per Algorithmus verknüpft. Ergebnis dieser Verknüpfung ist der Ciphertext. Sind Verschlüsselungsalgorithmus, der sogenannte Cipher, und der Key gut gewählt, ist ohne Schlüssel die Klartextnachricht nicht wiederherzustellen.

Die Kryptografie unterscheidet zwei Verschlüsselungsmechanismen, die symmetrische und die asymmetrische Verschlüsselung.

Wird der gleiche Schlüssel sowohl zum Ver- als auch zum Entschlüsseln verwendet, spricht man von symmetrischer Verschlüsselung. Symmetrische Cipher sind typischerweise sehr schnell, aber im Einsatz nicht immer unproblematisch. Soll beispielsweise eine Nachricht verschlüsselt werden, müssen Sender und Empfänger der Nachricht den Schlüssel kennen. Der Schlüssel ist damit ein *Shared Secret*, ein gemeinsames Geheimnis. Wenn jedoch mehr als zwei Personen das Geheimnis teilen, gilt es nicht mehr als geheim; bei einem Verrat ist der Schuldige nicht mehr eindeutig identifizierbar. Das zweite Problem betrifft den Schlüsseltransport. Hier muss sichergestellt werden, dass kein Dritter während des Transportes den Schlüssel mitlesen kann.

Bei der asymmetrischen Verschlüsselung werden für Ver- und Entschlüsselung unterschiedliche Schlüssel – die jedoch mathematisch zusammenhängen – eingesetzt. Man spricht vom Schlüsselpaar. Der zum Verschlüsseln eingesetzte Key wird als *Public Key* bezeichnet, zum Entschlüsseln benötigt man den zugehörigen *Private Key*. Der Name deutet es bereits an: Der Public Key ist öffentlich, unterliegt keinerlei Geheim-

haltung und darf nicht nur, sondern soll sogar publiziert werden. Der private Schlüssel dagegen ist zu schützen und darf auf keinen Fall in falsche Hände geraten. Da die Schlüssel vergleichsweise komplex sind, sind diese in Dateien abgespeichert. Da jeder, der den privaten Schlüssel besitzt, die mit dem zugehörigen öffentlichen Schlüssel verschlüsselten Inhalte wiederherstellen und ein Diebstahl des Schlüssels nicht immer verhindert werden kann, wird der private Schlüssel häufig selbst noch einmal verschlüsselt. Hierfür kann dann ein symmetrisches Verfahren eingesetzt werden, denn das zum Verschlüsseln notwendige Geheimnis muss nur der Besitzer des Private Key kennen. Bei dem zum Verschlüsseln des Schlüssels eingesetzten Key handelt es sich um ein Passwort, welches nicht abgespeichert wird.

Public-Key-Verfahren lassen sich nicht nur zum Verschlüsseln, sondern auch zum digitalen Unterschreiben einsetzen.

Der öffentliche Schlüssel gehört zu einer Identität, beispielsweise einer Person. Diese Identität ist typischerweise über Informationen wie Name und Wohnort bekannt. Werden Public Key und die Angaben zur Identifizierung des Schlüsselbesitzers miteinander kombiniert und schließlich digital unterschrieben, spricht man von einem digitalen Zertifikat. Die digitale Unterschrift des Zertifikats schützt dieses vor Manipulationen. Im einfachsten Fall unterschreibt der Besitzer selbst sein Zertifikat (Self Signed Certificate). Typischerweise wird das Zertifikat aber von einer vertrauenswürdigen Instanz, einer *Certification Authority* (CA), ausgestellt. Allerdings ist die Frage, ob es überhaupt eine vertrauenswürdige Instanz geben kann beziehungsweise gibt!

Die Cipher (Verschlüsselungsalgorithmen) selbst lassen sich gemäß ihrer internen Arbeitsweise unterscheiden. Streamcipher verschlüsseln Nachrichten beliebiger Länge und typischerweise Zeichen für Zeichen. Blockcipher verarbeiten Nachrichten blockweise. Ein Block besteht beispielsweise aus 32 Zeichen, entsprechend 256 Bit. Nachrichten, die nicht dem Vielfachen einer Blockgröße entsprechen, werden mit Nullen aufgefüllt.

Eines der bekanntesten symmetrischen Verschlüsselungsverfahren ist AES. Hierbei handelt es sich um einen Blockcipher, der beispielsweise mit Blöcken der Länge 256 (AES-256) arbeitet.

Da bei Blockcipher identische Klartextblöcke zu gleichen Ciphertexten führen, werden verschiedene Betriebsarten unterschieden (Abbildung 6-8). Die beschriebene Betriebsart, bei der jeder Klartextblock mit dem Schlüssel verknüpft wird, nennt sich Electronic Code Block (ECB). Beim sogenannten Cypher Block Chaining (CBC) wird der jeweilige Block nicht nur mit dem Schlüssel (der typischerweise genau die Länge eines Blocks hat) verschlüsselt, sondern zusätzlich noch mit dem vorherigen Block. Da es beim ersten Block noch keinen Vorgänger gibt, ver-

wendet man einen Block, der mit Zufallswerten gefüllt ist. Dieser Block heißt Initialisierungsvektor (IV). Nachteilig am CBC-Verfahren ist, dass die Ver- und Entschlüsselung nur sequenziell erfolgen können.

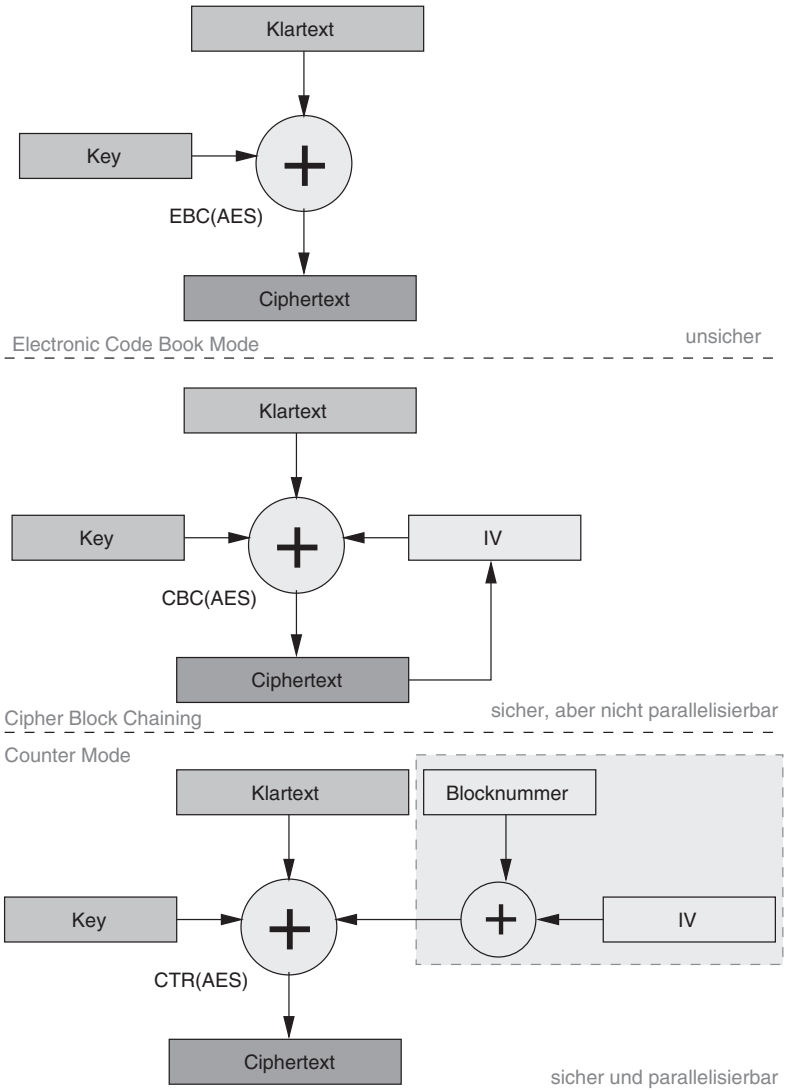


Abbildung 6-8
Blockcipher-Betriebsarten

Um auch die parallele Verschlüsselung zu ermöglichen, wird beim Counter Mode (CTR) der jeweilige Nachrichtenblock mit dem Schlüssel und einem Initialisierungsvektor verknüpft. Der Inhalt des für jeden Block unterschiedlichen IV ergibt sich aus einem einmal festgelegten (Zufalls-)Wert und der Blocknummer durch sinnvolle, mathematische Verknüpfung.

Da symmetrische Verschlüsselungsverfahren höhere Geschwindigkeiten ermöglichen, asymmetrische aber mehr Funktionalität, werden beide Verfahren in der Praxis häufig kombiniert eingesetzt. Das asymmetrische Verfahren ermöglicht die Authentifizierung und realisiert den sicheren Schlüsseltransport für die symmetrische Verschlüsselung, die ihrerseits den eigentlichen Datenstrom vor unautorisierten Zugriffen sichert.

Datenverschlüsselung

Die Verschlüsselung von Daten sollte grundsätzlich beim Entwurf eines Realzeitsystems eingeplant werden. Typischerweise wird hierfür ein symmetrischer Blockcipher wie AES-256 eingesetzt. Die Verschlüsselung direkt in die Applikation einzubauen, macht die Applikation auf der einen Seite zwar unabhängig vom darunter liegenden Betriebssystem, ist auf der anderen Seite aber aufwendig. Betriebssysteme wie Linux bieten die Möglichkeit, einzelne Verzeichnisse oder sogar den kompletten Hintergrundspeicher zu verschlüsseln.

Die Komplettverschlüsselung des Hintergrundspeichers ist komplizierter, insbesondere wenn sie bei einem eingebetteten System realisiert werden soll. Als Vorteil erweist sich der Schutz der Daten, falls das Gerät ausfällt und beispielsweise zur Reparatur eingeschickt werden muss. Während das Gerät jedoch in Betrieb ist, sind die Daten zumindest teilweise ohnehin entschlüsselt. Damit ist zu dieser Zeit zumindest theoretisch ein unautorisierter Zugriff auf die Daten möglich.

Bei der Teilverschlüsselung werden einzelne Dateien oder Verzeichnisse, beispielsweise ein Homeverzeichnis, verschlüsselt. Ein Vorteil dieses Verfahrens ist, dass bei geeigneter Systemauslegung das Verzeichnis durch das Einbinden (Mounten) dann entschlüsselt wird, wenn darauf Zugriffe erfolgen. Wird es danach gleich wieder ausgehängt, ist die Zeit, in der sensible Daten abgegriffen werden können, sehr kurz.

Einzelne Dateien lassen sich beispielsweise durch Einsatz von GnuPG sichern.

Für die Teilverschlüsselung von Verzeichnissen steht plattformübergreifend die Software Truecrypt zur Verfügung. Linux selbst bietet das Overlay-Verzeichnis *ecryptfs* an.

Verschlüsselung der Kommunikation

Die Sicherung einer Kommunikationsverbindung besteht nicht nur in der Verschlüsselung der Daten, sondern zusätzlich auch in der Authentifizierung der Kommunikationspartner. Dazu sind zertifikatsbasierte Verfahren (asymmetrische Verschlüsselung) unabdingbar.

Ähnlich wie bei der Datenverschlüsselung gibt es auch bei der Kommunikation die Möglichkeit, auf der einen Seite die Verschlüsselung di-

rekt in eine Applikation einzubauen oder auf der anderen Seite die Mechanismen des Betriebssystems zu nutzen.

Für die »In-Application-Verschlüsselung« bietet sich das Protokoll TLS beziehungsweise der Vorgänger SSL an. Mit OpenSSL und GnuTLS stehen zwei Bibliotheken zur Verfügung, die nur noch in die Applikation einzubinden sind. Der eigentliche Datenaustausch ist dabei erwartungsgemäß ähnlich wie der ohne Verschlüsselung. Unterschiede ergeben sich vor allem bei der Initialisierung – hier muss schließlich der Zugriff auf Zertifikate und (private) Schlüssel realisiert werden. Neben der Bibliothek bringt OpenSSL Werkzeuge mit, um Schlüsselpaare und Zertifikate zu erzeugen und zu verwalten. GnuTLS und OpenSSL unterscheiden sich im Wesentlichen bezüglich der zugehörigen Lizenz.

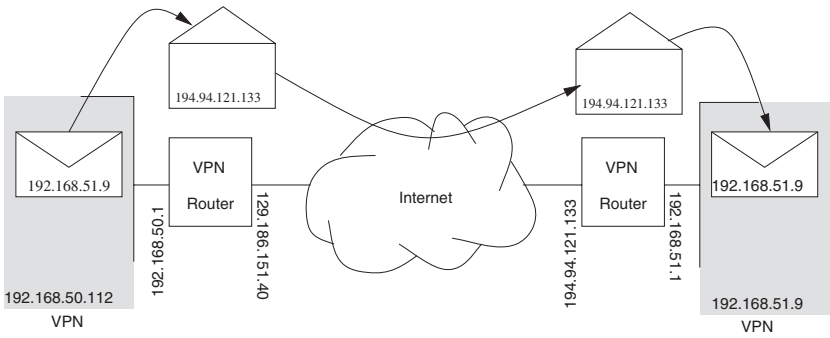


Abbildung 6-9
Tunneln der Daten über ein Virtual Private Network

Per Virtual Private Network (VPN) kann der gesamte Datenverkehr von und zu einem Realzeitsystem gesichert werden. Neben der Verschlüsselung und der Authentifizierung ermöglichen sie zudem durch sogenanntes Tunneling den sicheren Transport der Daten über unsichere Netze, wie beispielsweise über das Internet (Abbildung 6-9).

Zur Realisierung eines VPN auf dem System selbst stehen insbesondere die beiden Varianten `IPSec` und `OpenVPN` zur Verfügung. Bei IPSec werden die Daten im Betriebssystemkern selbst verschlüsselt. Das Austauschen der Authentifizierungsinformationen ist wie die gesamte Konfiguration selbst sehr komplex, sodass zum Einsatz des Protokolls sehr viel Know-how gehört.

Deutlich einfacher im Einsatz ist OpenVPN. Im Unterschied zu IPSec findet die Verschlüsselung im Userland statt, die zu schützenden Daten werden in normale UDP-Pakete verpackt und zwischen den OpenVPN-Endpunkten ausgetauscht. Die notwendigen Modifikationen am Betriebssystemkern fallen erheblich geringer aus als bei IPSec, da hier nur ein virtuelles Netzwerkinterface implementiert werden muss, welches die Daten der Applikationen entgegennimmt und an die OpenVPN-Software im Userland weiterreicht. OpenVPN liefert alle

notwendigen Elemente, inklusive der Werkzeuge zur Generierung der Zertifikate, sodass der Einstieg und die Konfiguration vergleichsweise einfach sind.

Datenfilterung (Firewall)

Internet-Protokolle

Kommunikationsprotokolle der IP-Familie dienen zur Vernetzung von Netzwerken (Inter-Net) und Rechnern beziehungsweise Geräten. Jedes Netz bekommt hierzu eine Netzadresse, die Rechner im Netz eine Rechneradresse. Die Kombination aus Netz- und Rechneradresse wird IP-Adresse genannt. Diese besteht aus 32 Bit (bei IPv6 sind es 128 Bit). Protokolle wie IP und ICMP basieren – neben der Information über den Protokolltyp – auf einer Absender- und einer Ziel-IP-Adresse und ermöglichen damit die Host-zu-Host- (Rechner-zu-Rechner-) Kommunikation.

Damit Applikationen untereinander kommunizieren können, erweitern Protokolle wie TCP und UDP die IP-Adressinformationen um eine Absender- und eine Ziel-Portnummer. Die Portnummer repräsentiert die Applikation, wobei bekannten Serverapplikationen wie einem Webserver auch bekannte Portnummern (80) zugeordnet werden.

Die Bedeutung der verschickten Daten für die Applikation ist durch die Applikationsprotokolle, wie beispielsweise http, smtp, ssh oder ftp, festgelegt.

Die Firewall eines vernetzten Systems hat zwei Aufgaben: Zum einen filtert sie den ein- und ausgehenden Datenverkehr, zum anderen reduziert sie die Last. Diese Lastreduktion ist wiederum eine Schutzvorrichtung gegen einen Denial-of-Service-Angriff, bei dem versucht wird, das Gerät in eine Überlast-Situation zu bringen.

Firewalls sind meist als sogenannte Paketfilter realisiert, die neben der Flussrichtung (wird ein Paket gesendet oder wird eines empfangen) auf Basis der Adressinformationen eines IP-Paketes (siehe Kasten oben "Internet-Protokolle") entscheiden, ob ein Paket weiterverarbeitet oder abgelehnt (weggeworfen) wird. Dazu werden verschiedene Regeln aufgestellt, die von der Firewall bei jedem Paket der Reihe nach abgearbeitet werden.

Firewalls werden gemäß der Strategie aufgebaut, alle Pakete, die nicht ausdrücklich als erlaubt ausgefiltert werden, zu verbieten (Default-Strategie Deny).

Das Programm, mit dem unter Linux Filterregeln gesetzt und ausgelesen werden können, heißt `iptables`. Das in Beispiel 6-1 gezeigte Skript konfiguriert die Linux-Firewall so, dass nur sichere Verbindungen zum Webserver per https (Port 443) möglich sind, der auf dem Rechner selbst läuft. Dazu werden zunächst sämtliche Regeln der Firewall gelöscht. Anschließend wird für den ankommenden, herausgehenden und durchgehenden Datenverkehr die Default-Policy auf gesperrt gesetzt

und schließlich gezielt der lokale Webserver für Datenverkehr, der an den Zielport 443 (dport = destinationport) gerichtet ist beziehungsweise von diesem abgeht (sport = sourceport), freigegeben.

Generell sollte die Firewall einfach und übersichtlich aufgebaut werden. Sie sollte ausschließlich die Pakete durchlassen, die benötigt werden. Typischerweise sind das die Dienste wie Domain Name Service (DNS, Port 53), https-Server (Port 443) und eventuell Remotezugriff über SSH (Port 22).

Beispiel 6-1
Einfaches Skript zur Firewall-Konfiguration

```
#!/bin/bash

# Grundzustand herstellen
iptables -F
iptables -X
iptables -t nat -F
iptables -t nat -X
iptables -t mangle -F
iptables -t mangle -X
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP

# Zugriffe auf den lokalen Webserver per https erlauben
iptables -A INPUT -p tcp --dport 443 -j ACCEPT
iptables -A OUTPUT -p tcp --sport 443 -j ACCEPT
```

Minimalistischer Funktionsumfang

Software muss das tun, was man von ihr erwartet; Software darf nicht mehr tun, als man erwartet, und Software muss das, was sie tut, richtig tun.

Da es keine fehlerfreie Software gibt, ist es ein Ziel, möglichst wenig Software einzusetzen. Das betrifft die eigenen Applikationen, die sich auf die notwendige und geforderte Funktionalität beschränkt, als auch das System als Ganzes.

Daher sind insbesondere bei einem eingebetteten System sämtliche Komponenten, die standardmäßig installiert, aber nicht benötigt werden, zu deinstallieren. Wird bei einem Embedded-Linux beispielsweise Busybox als Userland eingesetzt, ist Busybox so zu konfigurieren, dass nur die notwendigen Teile einkompiliert sind.

Werden Applikationen oder Dienste auf dem System nur zeitweise benötigt, sollten diese auch nur in dieser Zeit aktiviert sein.

Security by Isolation

Unter Isolation versteht man in der IT-Sicherheit eine Technik, bei der Rechner, Dienste und Dateien dem Zugriff durch Sperren oder Abschließen entzogen werden.

Rechner, die nicht mit dem Internet verbunden sind, können auch nicht über das Internet angegriffen werden. Auf (externe) Festplatten, die stromlos geschaltet sind, kann ebenfalls nicht zugegriffen werden.

Dass ein System sicherer wird, je weniger Zugangsmöglichkeiten es besitzt, leuchtet ein; Isolationstechniken stellen daher die erste Wahl bei den Maßnahmen zur Absicherung dar! Unglücklicherweise haben Isolationstechniken häufig den Nachteil, dass sie Sicherheit auf Kosten von Funktionalität beziehungsweise Bedienbarkeit realisieren.

Folgende Isolationstechniken bieten sich an:

- Physikalische Isolation
- Verwendung alternativer Plattformen
- Virtualisierung

Bei der physikalischen Isolation wird das zu schützende System vor der Außenwelt abgeschirmt. Dazu wird es in einem abgeschlossenen Raum untergebracht, zu dem nur die autorisierten Personen Zugriff erhalten. Des Weiteren weist das System möglichst keine Schnittstellen auf, über die es angegriffen werden kann. So werden alle USB-Schnittstellen unbrauchbar gemacht, DVD-Laufwerke entfernt, WLAN- und Bluetooth-Adapter ausgebaut.

Systeme, die eine hohe Verbreitung haben, werden häufiger als außergewöhnliche Systeme angegriffen. Daher sollte der Systementwickler sowohl alternative Hardware-Plattformen und alternative System- und Applikationssoftware auswählen.

Als Drittes bietet sich Virtualisierung zur Isolation an. Einzelne Applikationen oder aber auch eine Gruppe von Applikationen werden in eine virtuelle Maschine verlagert. Wird die separierte Applikation kompromittiert, hat das zunächst keinerlei Auswirkungen auf die Daten und Dienste in den übrigen Systemteilen.

Die Virtualisierung bringt neben sicherheitstechnischen Vorteilen auch die einfache Wiederherstellung (Recovery) mit sich. Dazu ist von der Datei, welche die virtuelle Maschine repräsentiert, regelmäßig ein Backup anzulegen.

6.2.2 Strukturelle Abwehrmaßnahmen (Security by Structure)

Der Schutz vor Sabotage, Spionage und Missbrauch fällt im Umfeld der Automatisierungstechnik durch Vorgaben der Systemhersteller besonders schwer. Diese garantieren die Funktionalität nur unter der Maßga-

be, dass nur ausdrücklich durch den Hersteller autorisierte Veränderungen vorgenommen werden dürfen. Konkret bedeutet dieses, dass der Betreiber eines Systems keinen Patch einspielen darf, der nicht vom Systemhersteller freigegeben wurde. Das schließt insbesondere auch Fehlerkorrekturen ein, die die Systemsoftware, in Abgrenzung zur Anwendungssoftware, betreffen. Der Systemhersteller selbst verfügt jedoch kaum über die benötigten Ressourcen, um zeitnah die Fehlerkorrekturen zu überprüfen und die Freigaben zu erteilen. Damit bleiben Anlagen der Automatisierungstechnik typischerweise über viele Jahre ungepatcht und mit bekannten Fehlern behaftet. Umso wichtiger ist es, diese Systeme bereits durch die Art der Einbettung in ihre Umgebung abzusichern.

Zu den strukturellen Methoden gehören

- die Einordnung der kritischen Systeme in ein eigenes Subnetz,
- die Wahl geeigneter IP-Adressen,
- die Sicherung durch eine Firewall und
- der Einsatz von VPN-Technologie.

Um Systeme einfacher schützen zu können, sollten diese in Subnetze verlagert werden, die eine eigene Netzadresse bekommen. Das macht die Absicherung generell einfacher und übersichtlicher.

Das Internet-Protokoll unterscheidet zwischen privaten und öffentlichen Netzadressen. Während eine öffentliche IP-Adresse offiziell genau ein einziges Mal vergeben wird, kommen private IP-Adressen mehrfach vor. Daher können und dürfen diese Adressen im Internet auch nicht weitergeleitet werden; Rechner mit einer privaten IP-Adresse sind aus dem Internet zunächst nicht erreichbar.

Diesen Umstand nutzt man sicherheitstechnisch aus. Durch die Vergabe einer privaten IP-Adresse ist das System im Internet nicht sichtbar und kann zunächst – solange kein NAT aktiv ist – von außen nicht angegriffen werden.

Um trotz privater IP-Adresse auf das Internet zugreifen zu können, wird die Technik Network Address Translation (NAT) eingesetzt. Dank SNAT kann das Realzeitsystem beispielsweise notwendige Updates über das Internet laden. Für den Zugriff auf Statusinformationen beispielsweise kann das Realzeitsystem kontrollierbar per DNAT über das Internet zugänglich gemacht werden.

Soll bei einem System eine Fernwartung möglich sein, bietet sich hierfür – unabhängig von der privaten IP-Adresse – die bereits beschriebene VPN-Technologie an, da diese eine Authentifizierung ermöglicht.

Network Address Translation (NAT)

Unter Network Address Translation versteht man das Ändern von Ziel- oder/und Absenderadressen in IP-Paketen. Es dient dazu, Rechnern mit einer privaten IP-Adresse den Zugang zum Internet zu ermöglichen beziehungsweise umgekehrt Rechnern aus dem Internet Zugriff auf einen Rechner im privaten Netz zu geben. Hierzu werden zwei Arten von NAT unterschieden: Source NAT (SNAT) und Destination NAT (DNAT).

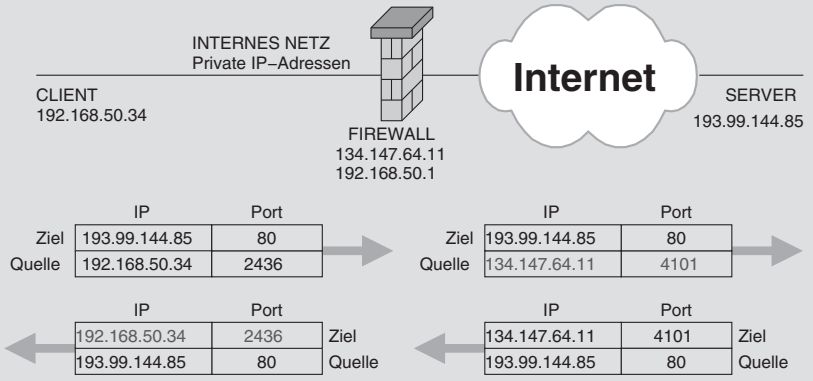


Abbildung 6-10 *SNAT ermöglicht trotz privater Netzadresse Zugang zum Internet.*

SNAT ermöglicht Systemen mit einer privaten IP-Adresse den Zugang zum Internet. Dazu wird die (private) Quelladresse (IP-Adresse plus Port) auf dem Router, der zwischen internem Netz und dem Internet vermittelt, gegen eine öffentliche Quelladresse (Absenderadresse) ausgetauscht. Antworten aus dem Internet kommen dadurch zunächst wieder bei dem Router an, der anhand der Ziel-Portadresse merkt, dass es sich um ein »genattetes« Paket handelt. Er tauscht dieses Mal die (öffentliche) Zieladresse gegen die private Adresse aus und leitet das modifizierte Paket in das interne Netz weiter.

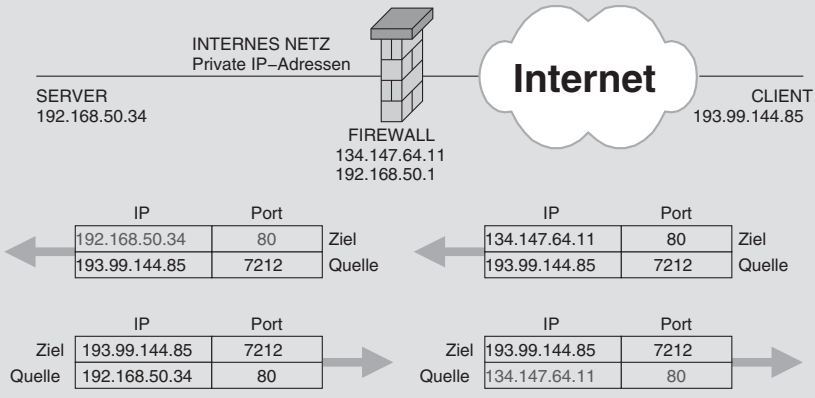


Abbildung 6-11 *DNAT ermöglicht den Zugriff auf Server mit privater IP-Adresse.*

DNAT, auch als Port-Forwarding bezeichnet, ermöglicht es, aus dem Internet Rechner im privaten Netz zu erreichen. Dazu tauscht der Router, der zwischen dem internen Netz und dem Internet vermittelt, die Zieladresse (IP-Adresse und Port) eines an ihn gerichteten Paketes gegen die private Adresse (IP-Adresse und Port) aus. Im Anschluss leitet er das Paket in das interne Netz. Antwortet der interne Rechner, wird – wie bei SNAT – die Absenderadresse ausgetauscht, bevor das Paket den Router in Richtung Internet verlässt.

7 Formale Beschreibungsmethoden im Überblick

Im Rahmen der Konstruktion eines Realzeitsystems werden Zustandsgrößen und Stellwerte des technischen Prozesses (zum Beispiel Temperaturwerte oder Ventile) auf logische Einheiten, auf Tasks abgebildet (Abbildung 7-1).

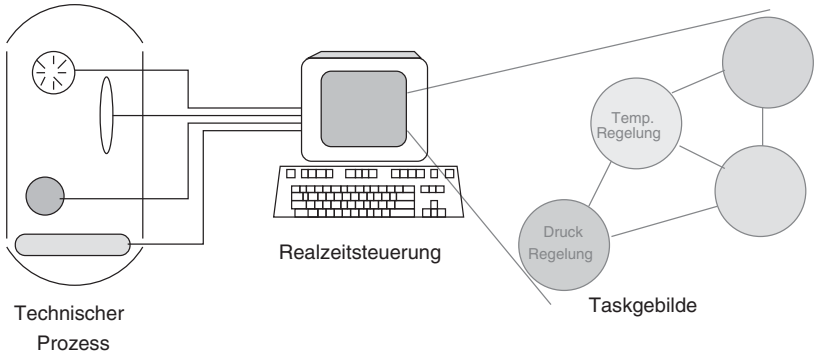


Abbildung 7-1
Rechenprozesse bilden den technischen Prozess ab.

Das sich dabei ergebende Taskgebilde muss formal beschrieben werden. Nur so sind eine Planung, ein Review und später auch eine Wartung und Reparatur möglich. Ein sinnvolles Vorgehensmodell ist dabei ein Top-down-Ansatz. Zunächst ist das Taskgebilde als solches, also welche Tasks existieren, welche Sensor- und welche Aktorwerte werden verarbeitet und welche Daten werden untereinander ausgetauscht, zu beschreiben. Hierbei sind beispielsweise Datenflussdiagramme sehr hilfreich, die um Kontrollflüsse ergänzt werden. Informationen über zeitliche Abläufe enthält das Datenflussdiagramm übrigens nicht.

Ist bekannt, welche Software-Komponenten das Realzeitsystem ausmachen, werden die einzelnen Tasks als solche beschrieben. Hierzu bieten sich die klassischen Struktogramme an, die sich leicht in strukturierten Code umsetzen lassen.

Petrinetze ermöglichen die Modellierung der Abläufe. Die Modellanalyse schließlich ermöglicht unter anderem Rückschlüsse auf mögliche Deadlocks im System.

Insbesondere bei komplexen Systemen wird auf UML (Unified Modelling Language) zurückgegriffen. UML bietet 13 unterschiedliche Diagrammtypen an, die für die Modellierung jedoch nicht alle benötigt werden. Ein auf UML basierender Entwurf wird typischerweise mithilfe von rechnergestützten Werkzeugen (Tools) durchgeführt, die bei der Erstellung der zum Teil komplexen Diagrammformen behilflich sind. So umfangreich die Modellierungsmöglichkeiten von UML sind, so umfangreich ist auch der Einarbeitungsaufwand in die im Realzeitumfeld wenig intuitive Beschreibungssprache.

Der Entwurf von Realzeitsystemen inklusive der geeigneten Beschreibungsformen ist ein Thema für sich. Wir haben hier aus der Fülle Datenflussdiagramme, Struktogramme, Petrinetze und einige UML-Diagramme ausgewählt, um damit einen Überblick beziehungsweise Einstieg zu geben.

7.1 Daten- und Kontrollflussdiagramm

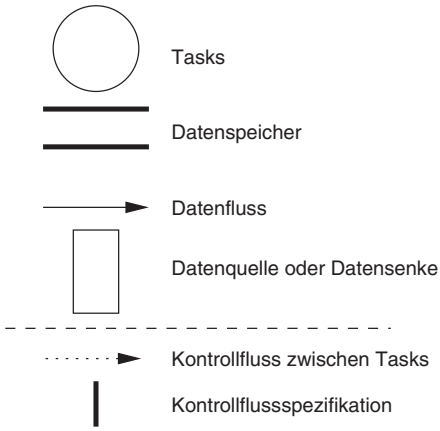


Abbildung 7-2
Grundelement von DFD und CFD

Zur Darstellung des Taskgebildes werden Datenflussdiagramme (siehe Abbildung 7-2) eingesetzt. Diese sind für die Darstellung im Realzeitumfeld noch um weitere Elemente, die den Kontrollfluss definieren, erweitert worden. Folgende grafische Elemente können verwendet werden:

Tasks

Tasks (Jobs, Rechenprozesse, Threads) werden als Kreise dargestellt. Sie verarbeiten eingehende Daten (ankommende Kanten) zu Ausgaben (abgehende Kanten). Für ihre Aufgabe benötigen sie auch lokale Daten, die aber für die Umgebung nicht sichtbar sind.

Tasks können schrittweise verfeinert werden. In diesem Fall erhalten sie zur Identifikation eine Kennung (meist ein einfacher Buchstabe). Unter dieser Kennung wird dann ein eigenes Datenflussdiagramm angefertigt. Auf der untersten Verfeinerungsebene werden zur Beschreibung der Task (und ihrer Algorithmen) sogenannte Prozessspezifikationen angefertigt.

Datenspeicher

Für mehrere Tasks sichtbare (also globale) Daten werden durch zwei parallele Striche dargestellt. Sie stellen einen temporären Aufenthaltsort für Daten dar. Dieser wird benötigt, wenn der Entstehungszeitpunkt der Daten verschieden ist vom Nutzungszeitpunkt. Daten werden geschrieben und gelesen.

Datenflüsse

Die eigentlichen Datenflüsse werden durch gerichtete Kanten (Pfeile) dargestellt. Als Beschriftung dient die Bezeichnung der über die Kante fließenden Daten, nicht die Operation auf die Daten.

Datenquellen und Datenspeicher

Als Rechtecke oder ohne jegliche Umrahmung werden Datenquellen und Datensenken dargestellt; dies sind beispielsweise Sensoren und Aktoren.

Kontrollflüsse

Der gestrichelte Pfeil deutet an, dass zwischen zwei Tasks eine Kontrollinformation (Event) ausgetauscht wird. Eine derartige Information kann beispielsweise das Starten oder Beenden einer Task, das Senden eines Events oder eines Signals, das Schlafenlegen und das Aufwecken einer Task oder schließlich das Betreten und Verlassen eines kritischen Abschnittes sein. Kontrollflüsse beginnen bei Tasks und enden entweder in einer Kontrollflussspezifikation oder bei einer anderen Task.

Kontrollflussspezifikation

Kontrollflüsse, die zwischen mehr als zwei Tasks bestehen, werden über eine Kontrollflussspezifikation realisiert. Diese repräsentiert beispielsweise ein Semaphor. Der Name der Kontrollflussspezifikation sollte sinnvollerweise die Bedeutung widerspiegeln. Zum Namen wird eine textuelle Beschreibung erstellt, die die beteiligten Instanzen und die genaue Bedeutung wiedergibt.

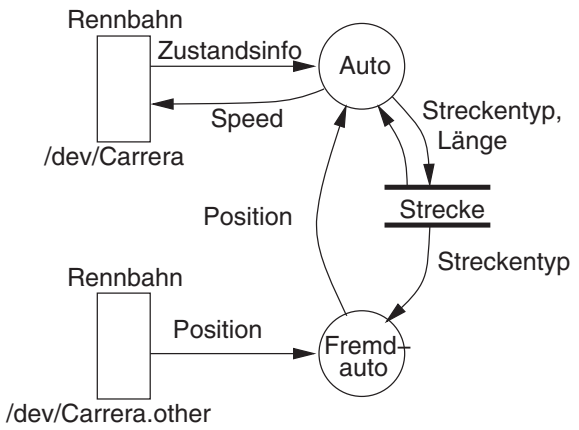


Abbildung 7-3
Datenflussdiagramm einer Carrerabahnsteuerung

Abbildung 7-3 stellt das Datenflussdiagramm einer Software dar, die ein per Computer zu fahrendes Carrerabahnauto steuert. Die vom Auto zu fahrende Strecke selbst ist in Segmente eingeteilt. Die Task *Auto* bekommt von der Rennbahn Zustandsinformationen (Segmenttyp) und berechnet daraus Geschwindigkeitswerte (Speed). Aufgrund der Zustandsinformationen lässt sich die Länge eines Streckensegments berechnen und zusammen mit dem Streckentyp in einem Datenspeicher ablegen.

Die Position des gegnerischen Fahrzeugs erhält die Task *Auto* über die Task *Fremdauto*. Diese bekommt Positionsangaben in Form von Segmenttypen ebenfalls von der Rennbahn, allerdings über das Interface `/dev/Carrera.other`. Zur genauen Bestimmung des gegnerischen Fahrzeugs auf der Strecke wird diese Information mit der Streckeninformation abgeglichen.

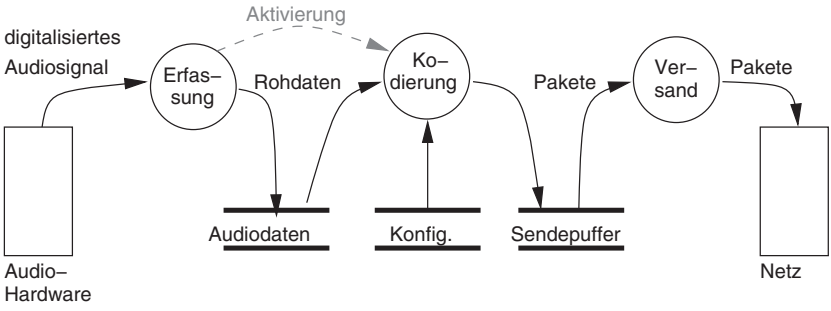


Abbildung 7-4
DFD-Audiostreaming

Abbildung 7-4 stellt das vereinfachte DFD eines Audiostreaming-Systems dar. Hier ist die Task *Erfassung* für die zeitgenaue Aufnahme der digitalisierten Audiosignale zuständig. Für die zeitgenaue Aufnahme der Daten ist eine Entkopplung dieser Aufgabe von der Weiterverarbeitung

notwendig. Aus diesem Grund werden die Daten in einem Datenspeicher abgelegt und nicht direkt der Task *Kodierung* übergeben. Diese Task liest zu Beginn eine Konfigurationsdatei, in der das zu verwendende Kodierungsverfahren samt seiner Parameter beschrieben ist. Entsprechend dieser Informationen werden die Daten, die sich im Datenspeicher *Audiodaten* befinden, kodiert, paketiert und in dem Sendepuffer abgelegt. Auch das Senden muss aus Gründen der Realzeitfähigkeit über einen Datenspeicher entkoppelt werden. Das Datenflussdiagramm enthält einen Kontrollfluss. Die Task *Erfassung* aktiviert die Task *Kodierung*, sobald genügend Rohdaten zur Verfügung stehen.

Anhand eines reinen Datenflussdiagramms sind keinerlei dynamische Abläufe sichtbar. Das DFD gibt eine statische Sicht der Dinge wieder. Dynamik kann partiell über die Erweiterung der Kontrollflüsse ausgedrückt werden. Kontrollflüsse zeigen an, welche Verarbeitungseinheit welche Verarbeitung auslöst.

Folgende Regeln sind bei der Erstellung eines Datenflussdiagramms zu beachten:

1. Jeder Datenfluss muss mit mindestens einer Task verbunden sein.
2. Direkte Datenflüsse zwischen Datenquellen und Datensenken (auch untereinander) sind nicht erlaubt.
3. Direkte Datenflüsse zwischen Datenspeichern sind nicht möglich.
4. Datenflüsse müssen immer gerichtet sein (Pfeilspitzen).
5. Unterschiedliche Datenflüsse zwischen zwei Tasks werden als separate Pfeile dargestellt.
6. Jede Task, jeder Datenfluss und jeder Datenspeicher müssen bezeichnet sein.
7. Beim Datenflussnamen muss es sich um ein Substantiv handeln, nicht um ein Verb. Der Name darf inhaltlich keine Verarbeitung und keinen Vorgang beschreiben!
8. Taskbeschreibungen der Art »Daten verarbeiten«, »Datensatz erzeugen« oder »Daten aufbereiten« sind zu vermeiden.
9. Kontrollflüsse werden mit Verben beschrieben.
10. Der Name der Kontrollflussspezifikation drückt deren Bedeutung aus.

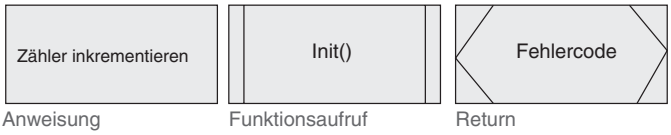
7.2 Struktogramme

Struktogramme dienen zur strukturierten Darstellung von Codesequenzen. Sie ersetzen die früher verwendeten Programmablaufpläne beziehungsweise Flussdiagramme.

Programme setzen sich aus Anweisungen (Aktionen) zusammen und aus den Kontrollstrukturen, die den Ablauf des Programms festlegen:

Bedingungen und Schleifen. Als besondere Anweisung sind Funktionsaufrufe (entsprechen dem Sprung in ein Unterprogramm) und der Rücksprung aus einer Funktion (Return) zu kennzeichen.

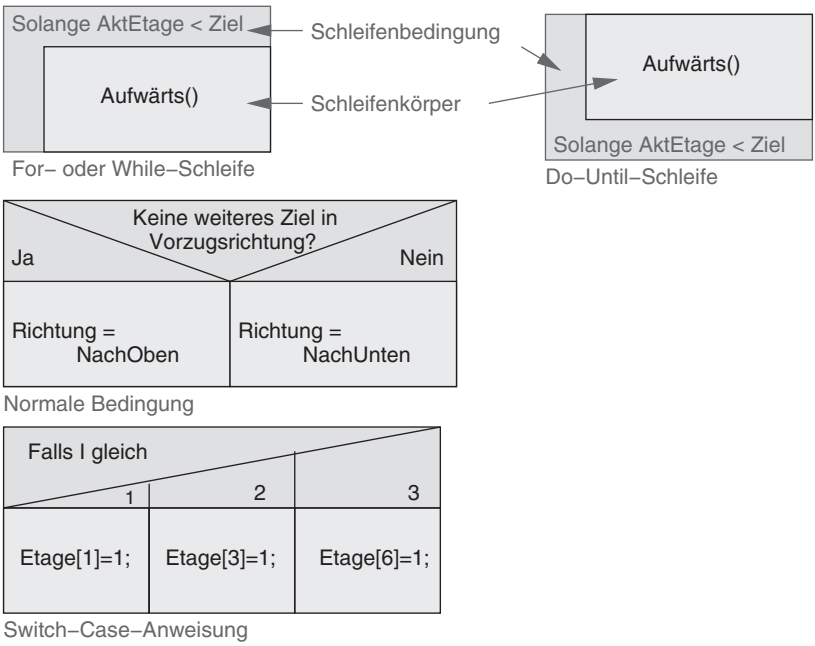
Abbildung 7-5
Darstellung von Anweisungsblöcken



Bedingungen kommen in Programmiersprachen in verschiedenen Konstrukten vor:

- als einfache Bedingung (IF-Abfrage),
- als kopfgesteuerte Schleife (der Schleifenkörper wird unter Umständen keinmal durchlaufen, Anweisungen `for` oder `while`),
- als fußgesteuerte Schleife (der Schleifenkörper wird mindestens einmal durchlaufen, Anweisungen `do - while`),
- als Mehrfachbedingung (SWITCH-CASE-Anweisung).

Abbildung 7-6
Darstellung von Bedingungen und Schleifen



Um eine Funktion mithilfe eines Struktogramms zu beschreiben, werden die einzelnen Blöcke aneinandergesetzt, wie in Abbildung 7-7 exemplarisch gezeigt.

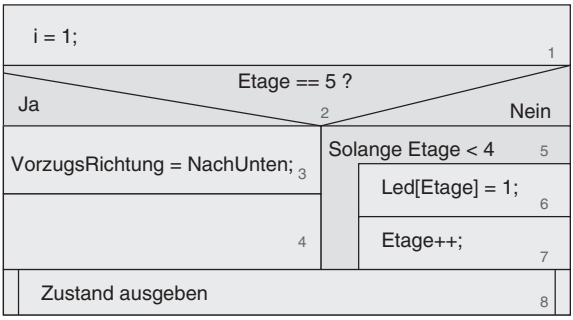


Abbildung 7-7
Verknüpfung der Struktogrammblöcke

Die im Struktogramm angegebenen Aktionen (Anweisungen) werden von oben nach unten abgearbeitet.

Falls beispielsweise `etage=2` wäre, ergäbe sich der folgende Ablauf (1, 2, 5, 6, 7, 5, 6, 7, 8).

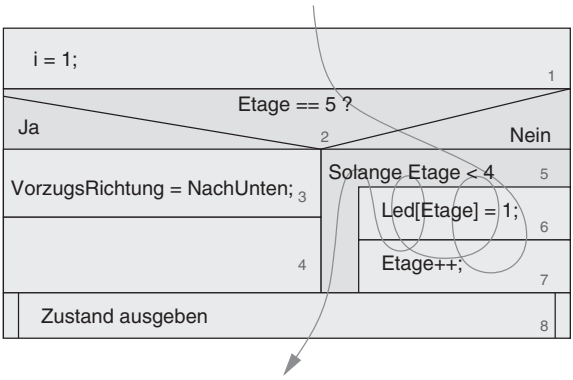


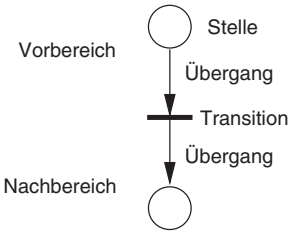
Abbildung 7-8
Beispiel für ein Struktogramm

7.3 Beschreibung nebenläufiger Prozesse (Petrinetze)

Ein Petrinetz ist ein Modell zur Beschreibung nebenläufiger (paralleler) Prozesse. Eng verwandt mit den Zustandsautomaten, ermöglicht es die Analyse der modellierten Systeme:

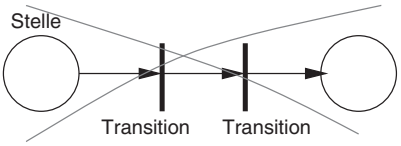
- Erreichbarkeit von Systemzuständen (Wie kann bei einem gegebenen Zustand ein zweiter Systemzustand erreicht werden?).
- Lebendigkeit beziehungsweise Terminierung (Werden die im System vorhandenen Ressourcen aufgebraucht?).
- Totale Verklemmung (Stillstand des Systems).
- Partielle Verklemmung (Stillstand von Systemteilen).

Abbildung 7-9
Petrinetz



Ein Petrinetz ist ein gerichteter Graph, der aus Knoten und Übergängen besteht. Bei Petrinetzen unterscheidet man zwei Knotenarten: die Stellen und die Transitionen. Stellen – üblicherweise als Kreise dargestellt – entsprechen Zuständen, Transitionen, die als Balken oder Rechtecke visualisiert werden, entsprechen den Aktionen bzw. Ereignissen. Die Knoten (Stellen und Transitionen) des Netzes werden durch Übergänge (Kanten) miteinander verbunden, wobei eine Kante jeweils nur von einer Stelle zu einer Transition führen darf beziehungsweise von einer Transition zu einer Stelle. Kanten zwischen gleichartigen Knoten (z.B. von Stelle zu Stelle) sind nicht erlaubt (Abbildung 7-10).

Abbildung 7-10
Zwei Transitionen direkt hintereinander sind nicht erlaubt.



Formal betrachtet ist ein Petrinetz ein Quadrupel PN = (S,T,F,M), wobei S die nicht leere Menge der Stellen, T die nicht leere Menge der Transitionen, F die Flussrelation (Übergänge) und M die nicht leere Menge der Marken ist.

Markenfluss. Anders als reine Zustandsautomaten lässt sich bei Petrinetzen das dynamische Verhalten modellieren. Dazu existiert eine oder mehrere Marken (Markierung) `M`, die gemäß der Schaltbedingungen von Stelle zu Transition »fließen«, dort zerstört werden, um danach entsprechend der nachfolgenden Anzahl von Übergängen neu erzeugt zu werden. Die Anzahl der Marken im Netz ist damit nicht konstant. Es fließen aus einer Transition so viele Marken, wie es Übergänge zu den nachfolgenden Stellen gibt. Werden die Marken in einem System so lange reduziert, bis keine schaltfähige Transition mehr existiert, ist das Petrinetz terminiert.

Schaltbedingungen. Wechselt eine oder mehrere Markierungen die Stelle, spricht man davon, dass die zwischen den Stellen befindliche Transition schaltet oder feuert. Eine Transition schaltet dann, wenn *alle* sogenannten Vorbedingungen erfüllt sind. Unter Vorbedingung wird dabei das Vorhandensein von jeweils mindestens einer Marke pro Über-

gang in der Stelle bezeichnet, durch die diese Stelle mit der betrachteten Transition verbunden ist.

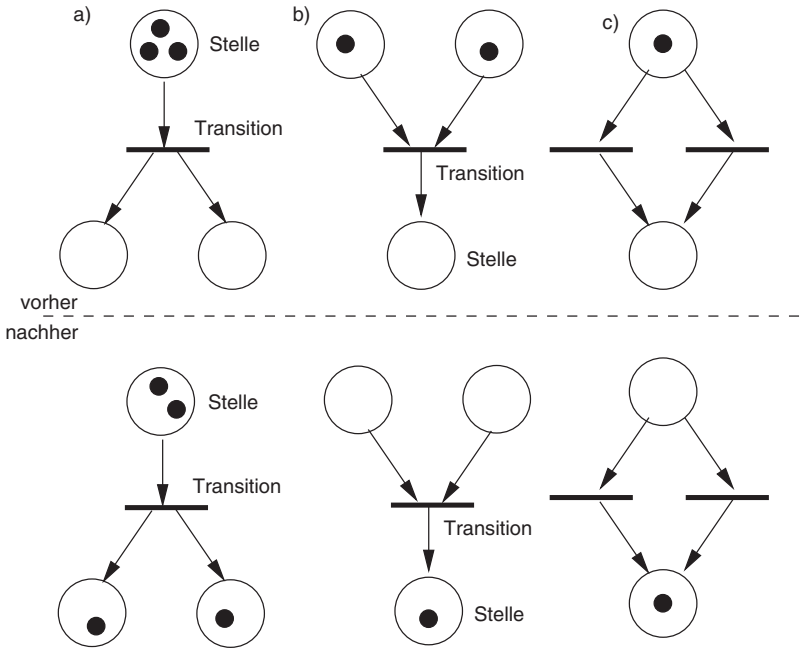


Abbildung 7-11
Unterschiedliche Übergänge bei Petrinetzen

In Abbildung 7-11 sind die unterschiedlichen Übergangsmöglichkeiten verdeutlicht. Im Teilbild a) wird ein einfacher Übergang dargestellt. Über die Transition fließt von den drei im Vorbereich existenten Marken eine einzelne. Da die Transition jedoch zwei Nachbedingungen (zwei abgehende Kanten, die in zwei Stellen führen) besitzt, wird die Marke verdoppelt. Im Teilbild b) kann die Transition schalten, weil sich in den Vorbedingungen jeweils eine Marke befindet. Da es nur eine Nachbedingung gibt, besitzt das Netz am Ende des Schaltvorganges auch nur noch eine Marke. Im Teilbild c) schließlich ist ein nicht deterministischer Übergang dargestellt. Für die beiden angegebenen Stellen ist die Vorbedingung erfüllt. Es ist aber nicht geklärt, welche der beiden Transitionen schalten wird.

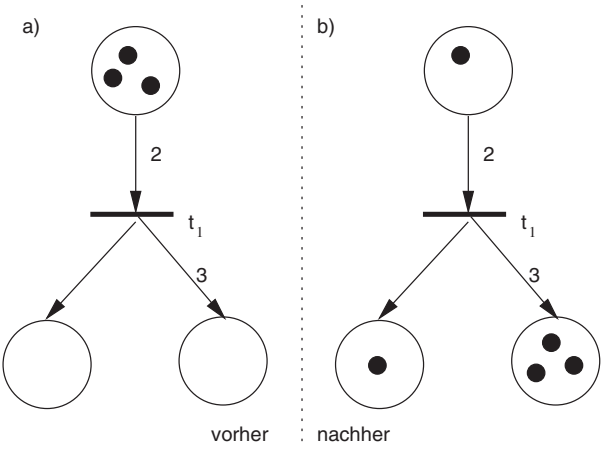


Abbildung 7-12
Stellen-/ Transitionsnetz

Petrinetze, bei denen immer genau eine Marke über einen Übergang fließt, heißen Bedingungs-/Ereignisnetz. Demgegenüber bieten die sogenannten Stellen-/Transitionsnetze die Möglichkeit, Kanten zu gewichten. Dazu wird jedem Übergang eine Zahl zugeordnet, die die Anzahl der Marken angibt, die über diesen Übergang beim Schalten der Transition (beziehungsweise zum Schalten der Transition) transferiert werden. Wird in einem Stellen-/Transitionsnetz für eine Kante keine Gewichtung angegeben, hat die Kante das Gewicht 1. In Abbildung 7-12 schaltet die Transition t_1 mit der Gewichtung zwei nur dann, wenn im Vorbereich mindestens zwei Markierungen vorhanden sind. In diesem Fall werden dem Vorbereich zwei Marken entnommen und im Nachbereich bekommt die linke Stelle eine Marke, die rechte Stelle jedoch drei Marken, da die Gewichtung der rechten Kante mit drei angegeben ist.

Modellbildung. Eine Modellbildung wird vereinfacht, wenn man zunächst die folgenden Schritte durchführt:

1. Die verwendeten Ressourcen (Betriebsmittel) sind zu identifizieren. Ressourcen sind beispielsweise Werkstücke, Werkstückträger oder auch Semaphore oder gemeinsame Speicherbereiche.
2. Betriebsmittel werden durch jeweils eine Stelle modelliert, die Anzahl der Ressourcen oft über die Anzahl der Marken in dieser Stelle (Anfangszustand).
3. Weitere Stellen ergeben sich, in dem die dynamischen Abläufe durchgespielt beziehungsweise die Bedingungen, die zu Zustandswechseln führen, betrachtet werden.

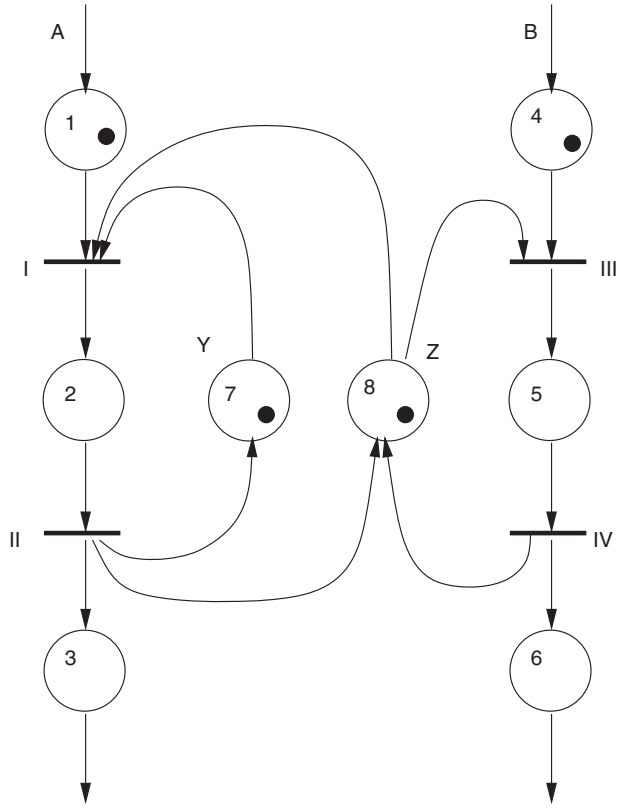


Abbildung 7-13
Petrinetz einer Fertigungsstraße [Abel1990]

Im Folgenden soll beispielhaft ein technischer Prozess in Form eines Petrinetzes modelliert werden.

Auf den beiden Fertigungsstraßen A und B kommen Werkstücke an, die durch zwei Handhabungssysteme (Y und Z, Roboter) bearbeitet werden. Zur Bearbeitung eines Werkstücks auf der Fertigungsstraße A sind beide Handhabungssysteme notwendig, zur Bearbeitung eines Werkstückes auf der Fertigungsstraße B jedoch nur der Roboter Z.

Aus der Beschreibung lassen sich die folgenden Ressourcen ableiten:

- Werkstück auf Fertigungsstraße A
- Werkstück auf Fertigungsstraße B
- Handhabungssystem Y
- Handhabungssystem Z

Die Fertigungsstraßen für sich allein werden hier nicht als Betriebsmittel modelliert, da sie anscheinend ständig zur Verfügung stehen.

Aus dieser Aufstellung lässt sich die Ausgangssituation (hier aus vier Stellen bestehend) modellieren. Werden jetzt die beschriebenen Bedingungen berücksichtigt, ergibt sich das in Abbildung 7-13 dargestellte Petrinetz.

7.4 Netzwerkanalyse

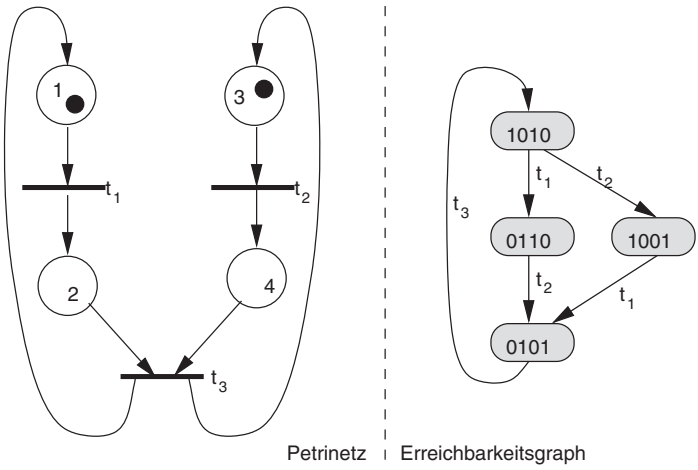
Mechanismen der Netzwerkanalyse ermöglichen die Identifikation von Verklemmungen. Dazu wird aufgrund des Petrinetzes ein sogenannter Erreichbarkeitsgraph aufgestellt.

Ein Erreichbarkeitsgraph ist ein gerichteter Graph, dessen Knoten erreichbare Markierungen `M` des zugeordneten Petrinetzes `PN` sind.

Konstruktion des Erreichbarkeitsgraphen. Ausgehend von der Anfangsmarkierung werden alle möglichen schaltfähigen Transitionen betrachtet. Die beim Schalten der Transition möglichen erreichbaren Markierungen werden als Knoten des Graphen übernommen. Dieses Vorgehen wird ebenfalls bei den neu entstandenen Knoten durchgeführt, und zwar so lange, bis keine Transition mehr gefunden wird, die noch nicht durch entsprechende Kanten im Graphen abgebildet wurde. Ein Kennzeichen des Erreichbarkeitsgraphen besteht darin, dass die Knoten des Graphen unterschiedliche Markierungen besitzen.

Die Knoten im Erreichbarkeitsgraphen, die nur eingehende, aber keine ausgehenden Kanten haben, stellen mögliche Verklemmungen dar.

Abbildung 7-14
Synchronisation per Petrinetz modelliert [Abel1990]



Im Folgenden soll die Erstellung des Erreichbarkeitsgraphen exemplarisch vorgestellt werden (Abbildung 7-14).

Das Petrinetz besteht aus vier Stellen, die initial folgendermaßen markiert sind:

- Stelle 1 besitzt eine Markierung.
- Stelle 2 besitzt keine Markierung.
- Stelle 3 besitzt eine Markierung.
- Stelle 4 besitzt keine Markierung.

Damit ergibt sich der erste Knoten zu `M={1,0,1,0}`. Von `M` sind sowohl die Transition t_1 als auch t_2 möglich. Schaltet die Transition t_1, wäre der nächste Knoten im Erreichbarkeitsgraph gegeben durch `M={0,1,1,0}`. Das Schalten der Transition t_2 führt zum Knoten `M={1,0,0,1}`.

Die Transition t_3 kann erst schalten, wenn die Stelle Zwei und die Stelle Vier besetzt ist. Je nach Knoten im Erreichbarkeitsgraphen muss daher die Transition t_2 oder t_1 schalten. Beide Transitionen führen dann aber in denselben Knoten `M={0,1,0,1}`. Hier kann die Transition t_3 schalten und damit wird der Initialzustand wieder eingenommen.

Der resultierende Erreichbarkeitsgraph ist in Abbildung 7-14 dargestellt. Ein weiteres Beispiel für ein Petrinetz mit zugehörigem Erreichbarkeitsgraph ist in Abbildung 7-15 dargestellt.

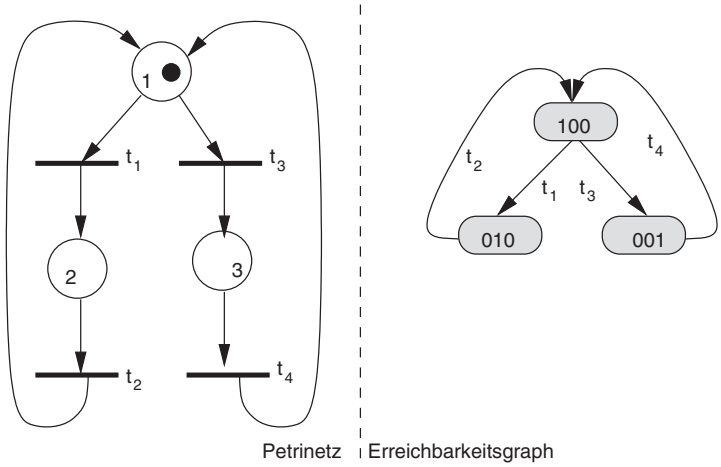


Abbildung 7-15
Zugriff auf ein gemeinsames Betriebsmittel [Abel1990]

Bei der Verklemmung wird die

- totale Verklemmung von der
- partiellen Verklemmung

unterschieden.

Im Folgenden ein weiteres Beispiel zur Verklemmung.

Kritische Abschnitte sind dann mehrfach betretbar, wenn während des Zugriffes innerhalb des kritischen Abschnittes keine Veränderungen an den Daten vorgenommen werden. Solange also kein Rechenprozess schreibend auf Daten innerhalb eines solchen Abschnittes zugreift, können mehrere Rechenprozesse gefahrlos lesend zugreifen.

Dieses Verhalten eines Lese-/Schreiblocks soll mithilfe normaler Semaphore nachgebildet werden, sodass zwei Rechenprozessen der lesende Zugriff oder einem Rechenprozess der schreibende Zugriff erlaubt ist. Dazu wird ein Semaphor mit 2 vorinitialisiert. Ein Prozess, der nur

lesend auf den kritischen Abschnitt zugreift, alloziert das Semaphor wie gewohnt einmal, ein Rechenprozess, der schreibend zugreifen möchte, alloziert dagegen das Semaphor zweimal. Die folgenden Codesequenzen verdeutlichen den Vorschlag:

Sequenz zum Lesen	Sequenz zum Schreiben
`P(S1)`	`P(S1)`
	`P(S1)`
`... // kritischer Abschnitt`	`... // kritischer Abschnitt`
`V(S1)`	`V(S1)`
	`V(S1)`

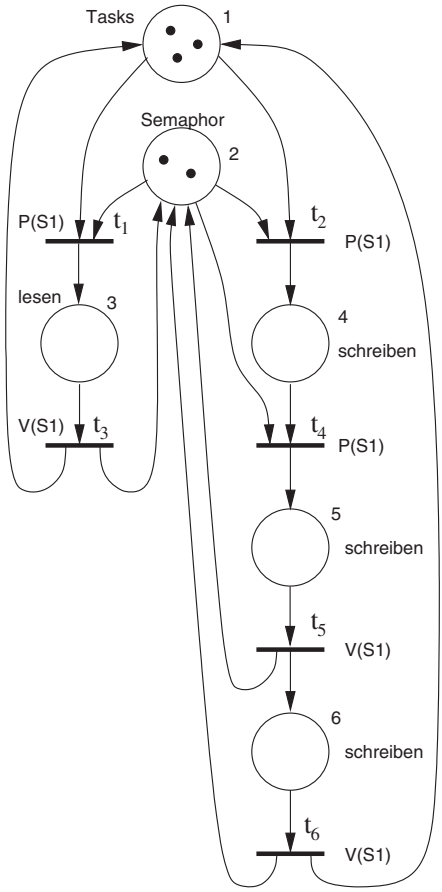


Abbildung 7-16
Deadlock-Untersuchung mithilfe eines Petrinetzes

Mithilfe eines Petrinetzes (siehe Abbildung 7-16) soll geklärt werden, ob dieser Vorschlag praktikabel ist. Dazu werden die beschriebenen Vorgänge in einem Bedingungs-/Ereignisnetz modelliert. Betriebsmittel sind

hier die zugreifenden Prozesse (von denen zur Vereinfachung maximal drei aktiv sein sollen) und das Semaphor.

Auf Basis des Petrinetzes lässt sich der Erreichbarkeitsgraph aufstellen:

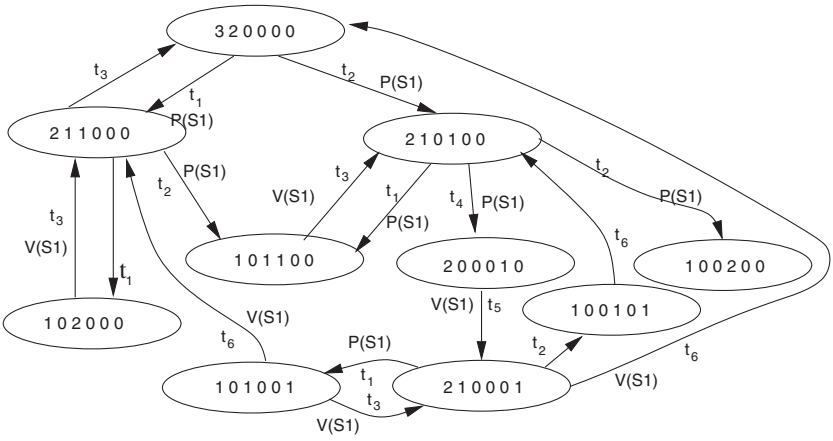


Abbildung 7-17
Erreichbarkeitsgraph

Da es im Erreichbarkeitsgraphen einen Knoten ohne abgehende Kanten gibt (100200), kann es zu einer Verklemmung kommen.

7.5 UML

UML wurde entwickelt, um bei Modellierung, Spezifizierung und Visualisierung komplexer Softwaresysteme geeignete Ausdrucksformen zu bieten. UML 2 – von der OMG in 2004 als Version 2 verabschiedet – besitzt 13 Diagrammtypen, die in jeder Phase des Entwurfs und der Entwicklung verwendet werden können. Die Diagrammtypen können in die Bereiche Struktur-, Verhaltens- und Interaktionsdiagramme unterschieden werden.

Grundlage der Modellierungen in UML ist der objektorientierte Ansatz (OOAD: Object-Oriented Analysis and Design). Das bedeutet nicht, dass für die Implementierung nur objektorientierte Sprachen infrage kommen. UML definiert nur die Notation, nicht die Methodik, das Vorgehensmodell oder die Implementierung. Eine auf UML basierende Sprache, die ohne den objektorientierten Ansatz auskommt, ist SysML. Die Diagramme in SysML sind angelehnt an UML, erlauben aber auch die Einbeziehung von Hardware und die Beschreibung gemischter Hard- und Softwaresysteme.

UML bietet somit eine Reihe von verschiedenen Modellierungselementen ([Rupp2012]). Welches der Diagramme in welcher Analyse- und Entwicklungsphase sinnvoll eingesetzt werden kann, hat der Standard

mit Absicht nicht definiert, denn er will keine Methode und kein Vorgehensmodell fest vorgeben. Dadurch ergeben sich dem Entwickler vielfältige Möglichkeiten, mithilfe der Diagramme ein System zu modellieren. Die Wahlmöglichkeiten benötigen jedoch Erfahrung, um die optimale Darstellungsart zu finden.

Prinzipiell unterscheidet man zwei Klassen von Modellierungstechniken:

- *Strukturmodelle* beschreiben die Beziehung zwischen den Komponenten des Systems statisch (»Teile und Herrsche«).
- *Verhaltensmodelle* beschreiben das Verhalten einer einzelnen Komponente sowie die dynamische und somit auch zeitliche Beziehung der Komponenten untereinander.

7.5.1 Strukturdiagramme

Strukturdiagramme werden zum Modellieren von klar definierten Komponenten und deren Schnittstellen benutzt. Die Stärke der Stukturdiagramme liegt darin, die Hierarchie der Komponenten sowie die statische Verbindung der Komponenten zueinander abzubilden.

Als ein Beispiel für ein Strukturdiagramm dient das Komponentendiagramm aus UML 2. Damit lassen sich zum einen die strukturellen Komponenten und deren Schnittstellen definieren sowie die statischen Verbindungen (Datenflüsse) zwischen den Komponenten modellieren. Komponenten können auch hierarchisch geschachtelt werden.

Die Hardware-Komponente repräsentiert Radumdrehungsmelder, Uhrzeit sowie Spannungs- und Strommesser. Die von der Komponente nach außen angebotene Schnittstelle wird von der Komponente ausgehend mit einem Vollkreis angegeben. Rechtecke an der Komponente markieren die Ports der Komponente. Benötigt eine Komponente eine externe Schnittstelle, so wird dies durch das Halbkreis-Symbol repräsentiert, welches an der Schnittstelle (Vollkreis) andockt.

Der Impuls einer Radumdrehung wird beispielsweise an die Komponente *V* zur Berechnung der Geschwindigkeit gegeben sowie auch an die Komponente *Monitoring* weitergeleitet. Die Monitoring Komponente zeichnet über die Geschwindigkeit hinaus auch zurückgelegte Kilometer und weitere Langzeitdaten (Strom-/Spannungsverlauf) auf. Ebenfalls benötigen beide Komponenten eine Zeitbasis für ihre Berechnung.

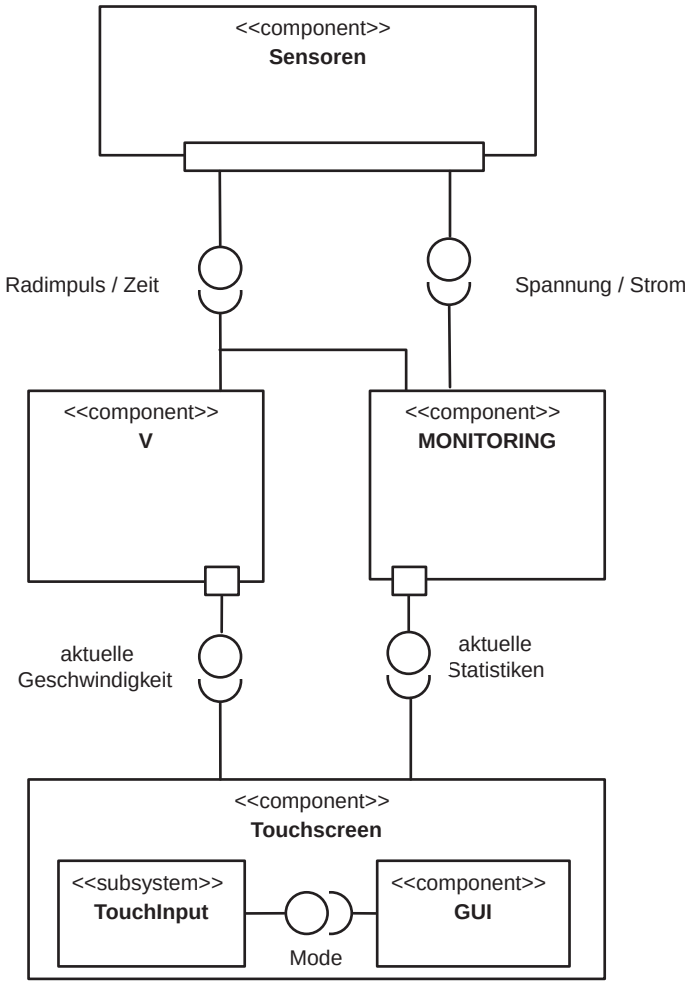


Abbildung 7-18
Strukturdiagramm Fahrradcomputer

Die GUI-Komponente benötigt die angebotenen Schnittstellen von *V* und *Monitoring*, um dem Benutzer die aktuellen Daten darzustellen. Sie ist in der Komponente *Touchscreen* enthalten. Teil der Touchscreen-Komponente ist das Subsystem *Touchinput*, über das der Benutzer den Modus für das GUI steuert. Je nach gwähltem Modus werden andere Daten von der GUI angezeigt.

7.5.2 Verhaltensdiagramme

Ein Beispiel für Verhaltensdiagramme ist das Anwendungsfalldiagramm (Use-Case), welches bei einer Systemanalyse mit dem Ziel benutzt wird, die Funktionalität und Systemgrenzen des zu entwickelnden Systems zu definieren. Das Anwendungsfalldiagramm ermöglicht eine Art Black-Box-Sicht auf das System.

Wichtige Elemente des Anwendungsfalldiagramms:

- der eigentliche Anwendungsfall (Use-Case) in Form einer Elipse,
- die Akteure (Strichmännchen), die den Anwendungsfall triggern bzw. instanziieren,
- das System, als umschließender rechteckiger Kasten.

Durch die Akteure und die Use-Cases wird klarer, wie mit dem System interagiert wird und welche Funktionalitäten das System bietet.

In Abbildung 7-19 ist dazu am Beispiel des Fahrradcomputers ein Anwendungsfalldiagramm dargestellt.

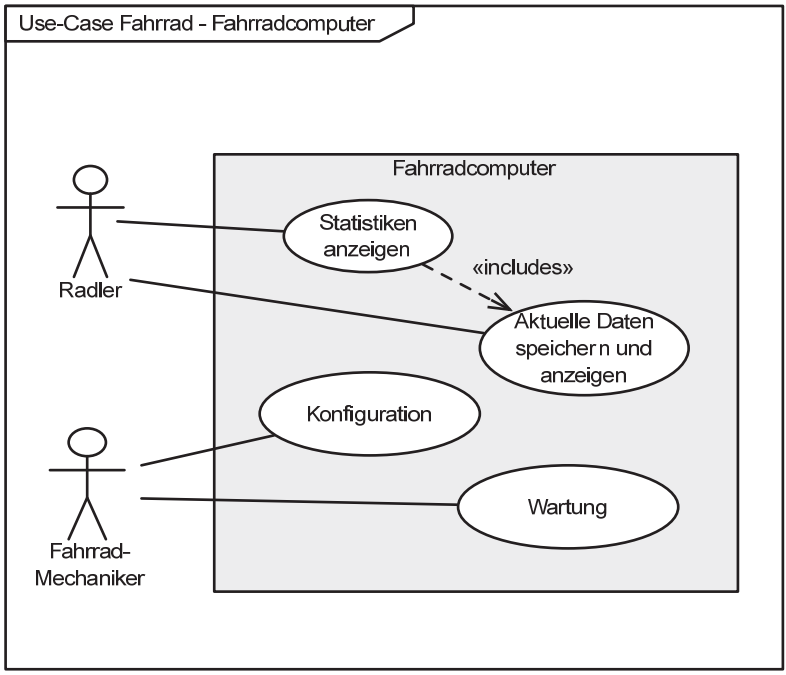


Abbildung 7-19 *Anwendungsfall-diagramm Fahrradcomputer*

Als mögliche Akteure unterscheiden wir den normalen Benutzer (Radler) und den Mechaniker (Konfigurator). Natürlich kann ein einzelner Benutzer auch beide Rollen annehmen.

Der Radler benutzt den Tacho, um sich während der Fahrt aktuelle Daten über Geschwindigkeit, zurückgelegte Kilometer, Batteriespan-

nung usw. anzeigen zu lassen. Eine weitere Aktion ist die Ausgabe von Statistiken auf dem Display, wie z.B. Fahrleistung im Jahr.

Der Akteur Mechaniker konfiguriert und wartet das System. Damit während der Fahrt korrekte Daten angezeigt werden, muss bei der Konfiguration vorab z.B. der Radumfang eingestellt werden. Der Austausch der Batterien ist ein Beispiel für die Wartung des Systems.

Ein weiteres Verhaltensdiagramm ist das Aktivitätsdiagramm. Die Wurzeln des Aktivitätsdiagramms sind unschwer in Struktogrammen und Petrinetzen zu erkennen. Aktivitätsdiagramme bieten sich an, um Algorithmen darzustellen. Vergleichbar mit Petrinetzen wandert ein Token durch das Aktivitätsdiagramm und kann je nach Verzweigungen und Zusammenführungen sequenzielle und parallele Abläufe darstellen.

Eine Aktivität ist definiert als eine Menge von Abläufen, die sich in der Realität unter bestimmten Randbedingungen abspielen. Ein Aktivitätsdiagramm zeigt eine oder auch mehrere Aktivitäten.

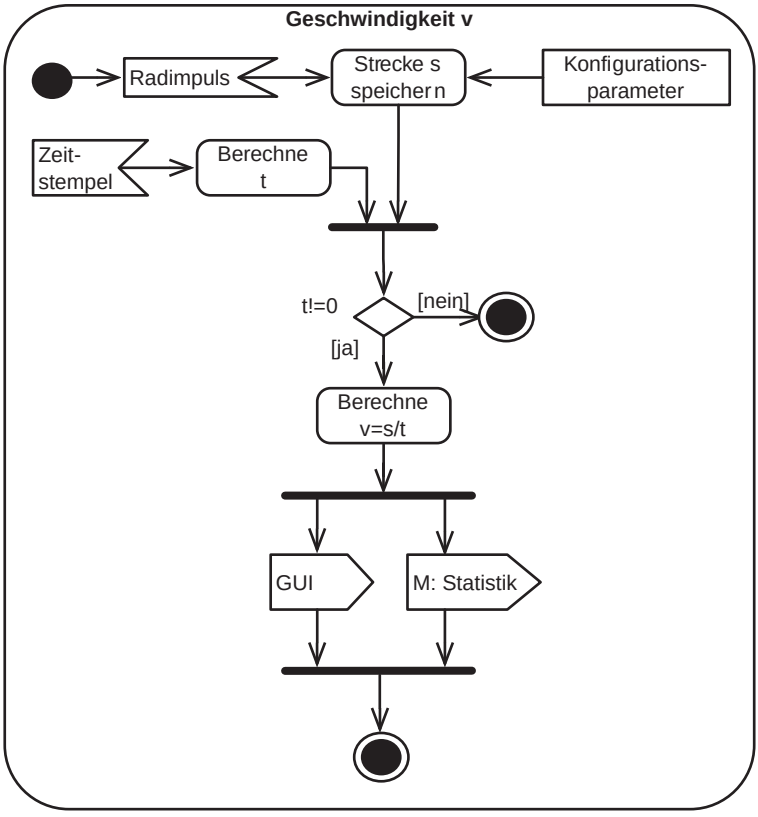


Abbildung 7-20
Aktivitätsdiagramm Fahrradcomputer

In dem Aktivitätsdiagramm (äußeres abgerundetes Rechteck) wird die Aktivität »Berechnung der Geschwindigkeit v« gezeigt. Die wichtigen Elemente des Aktivitätsdiagramms sind:

- Start- und Endknoten: Der Startknoten wird durch einen ausgefüllten schwarzen Punkt repräsentiert. Er repräsentiert den Startpunkt eines Ablaufs bei einer Aktivierung der Aktivität. Ein Endknoten (schwarzer Punkt mit Ring) beendet die gesamte Aktivität, sobald er von einem Token erreicht wird.
- Aktion: Eine Aktion wird durch ein Rechteck mit abgerundeten Ecken dargestellt. Sonderformen sind die Aktionen »Signal senden« (z.B. *M: Statistik*, hier also das Senden der Daten an die M-Task) und die Aktion »Signal empfangen« (z.B. *Zeitstempel* abrufen).
  Falls eine Aktion auf ein Ereignis wartet, das zeitabhängig versendet wird, benutzt man als eine weitere Sonderform eine stilisierte Eieruhr (nicht dargestellt).
- Objektknoten: Objektknoten werden durch ein Rechteck ohne Abrundungen dargestellt. Sie stehen für Daten oder Werte. Im Beispiel wird der Radumfang zur Bestimmung der Strecke benötigt, der wiederum Teil der Konfigurationsdaten ist.
- Kontrollelement: Token verweilen nicht in Kontrollelementen. Mit dem Start- und Endknoten sind bereits zwei Kontrollelemente beschrieben worden. Einige weitere, die wir im Diagramm verwendet haben, sind:
  - Verbindungs- und Verzweigungsknoten: In der Abbildung ist ein Verzweigungsknoten (if/then) dargestellt, der eine Division durch 0 abfängt.
  - Synchronisationsknoten: Die Berechnung von v muss auf die Bestimmung von s und t synchronisiert werden.
  - Parallelisierungsknoten: Die Ausgabe der Geschwindigkeit kann parallel an die Komponenten M und GUI weitergegeben werden.
- Kante: Kanten sind Übergänge zwischen Knoten und sind gerichtet. Knoten können z.B. Aktionen, Objektknoten oder Kontrollelemente sein.

8 Realzeitnachweis

Unter einem Realzeitnachweis versteht man den formalen Nachweis, dass alle Tasks eines Realzeitsystems die gegebenen minimalen und maximalen Deadlines unter allen Umständen einhalten. Es gibt eine Reihe verschiedener Möglichkeiten, dies auf grafische oder mathematische Weise zu bewerkstelligen. Die Beschreibung der Verfahren ist nicht trivial, insbesondere wenn versucht wird, alle Faktoren vollständig zu berücksichtigen. Dieses Kapitel versucht, den Realzeitnachweis anwendungsorientiert vorzustellen, sodass er in der Praxis genutzt werden kann.

Bevor nachfolgend der Realzeitnachweis für Realzeitsysteme vorgestellt wird, die ein prioritätengesteuertes Scheduling verwenden, und für Systeme, die auf einem Deadline-Scheduler beruhen, werden in Abschnitt 8.1 zunächst die Voraussetzungen geklärt.

Bei den von uns als Erstes vorgestellten Nachweisverfahren werden noch keine gemeinsamen und per Mutex oder Semaphor geschützten Ressourcen berücksichtigt. Das macht es leichter, den Überblick zu behalten. In Abschnitt 8.3 zeigen wir dann, wie Blockierzeiten berechnet und schließlich beim Nachweis berücksichtigt werden. Unabhängig davon, dass der Ansatz, den Realzeitnachweis nur mit den Userland-Tasks zu führen, praxistauglich ist, diskutieren wir in Abschnitt 8.4 noch den Einfluss des Betriebssystems.

8.1 Grundlagen

Um die Best- und die Worst-Case-Reaktionszeiten des Systems theoretisch nachzuweisen, überprüft der Realzeitnachweis prinzipiell die erste und zweite Realzeitbedingung (siehe Abschnitt 2.1):

1. Die Auslastung ρ_{ges} eines Rechensystems muss kleiner oder gleich der Anzahl Rechnerkerne *c* sein.

$$\rho_{ges} = \sum_{j=1}^{n} \frac{t_{Emax,j}}{t_{Pmin,j}} \leq c$$

2. Die Reaktionszeit jeder Task muss größer oder gleich der minimal zulässigen Reaktionszeit, aber kleiner oder gleich der maximal zulässigen Reaktionszeit sein.

$$t_{Dmin,i} \leq t_{Rmin,i} \leq t_{Rmax,i} \leq t_{Dmax,i}$$

Sind diese beiden Bedingungen erfüllt, handelt es sich um ein Realzeitsystem.

Um einen Realzeitnachweis durchführen zu können, müssen folgende zeitlichen und funktionalen Parameter des Gesamtsystems bekannt sein:

- minimale Prozesszeit t_{Pmin}, bei stochastischen Tasks die Auftrittshäufigkeit,
- minimal zulässige Reaktionszeit (minimale Deadline) t_{Dmin},
- maximal zulässige Reaktionszeit (maximale Deadline) t_{Dmax},
- minimale Ausführungszeit der Tasks t_{Emin},
- maximale Ausführungszeit der Tasks t_{Emax},
- Scheduling-Verfahren und im Fall eines prioritätengesteuerten Schedulings die Prioritäten der Tasks,
- Unterbrechbarkeit der Tasks und die Abhängigkeiten der Tasks voneinander,
- Ressourcenbenutzung der Tasks, insbesondere die Dauer der Ressourcenzugriffe.

Für jede Task muss der Realzeitnachweis die minimale und die maximale Reaktionszeit bestimmen. In einem Realzeitsystem befindet sich eine Task typischerweise in den Zuständen lauffähig (rechenbereit), aktiv oder schlafend (blockiert) (siehe Abbildung 3-5). Die Reaktionszeit einer Task ist somit die Summe aus der:

- Ausführungszeit der Task
- Summe der Verdrängungszeit (Preemption-Delay) aufgrund höherpriorer Tasks
- Summe der Blockierzeit aufgrund belegter Ressourcen durch niederpriore Tasks

Daher ist/sind für einen vollständigen Realzeitnachweis

1. die maximale Verzögerungszeit der Tasks aufgrund Verdrängungen (Preemption-Delay) durch höherpriore Tasks zu bestimmen. Der Fokus liegt auf dem korrekten Scheduling der rechenbereiten Tasks und der daraus entstehenden Verzögerungen.

2. die maximale Blockierzeit aufgrund belegter Ressourcen von niederprioren Tasks zu bestimmen. Berechnet wird die maximale Blockierzeit t_B. Eine höherpriore Task muss auf eine niederpriore Task warten, wenn letztere eine Ressource benutzt, die auch die höherpriore Task benutzen will.
3. die zeitlichen Auswirkungen des Betriebssystems zu betrachten und deren Einfluss auf die Reaktionszeiten der Tasks zu analysieren.

Geht man von periodischen Rechenzeitanforderungen aus – für den Worst Case ein legitimes Szenario – wiederholen sich die Vorgänge (Rechnerkernverteilung) nach einer bestimmten Zeit. Diese Zeit wird Hyperperiode genannt und ergibt sich als kleinstes gemeinsames Vielfaches (kgV) der minimalen Prozesszeiten der Tasks. Bei Tabelle 3-7 beträgt die Hyperperiode 120 ms, welche das kgV der minimalen Prozesszeiten 20 ms, 30 ms und 40 ms ist. Werden die Deadlines von den Tasks in der Hyperperiode eingehalten, handelt es sich um ein korrektes Realzeitsystem.

Im akademischen Umfeld wird häufig der Realzeitnachweis nur bezüglich der oberen Schranke $t_{Dmax,i}$ und nicht bezüglich der unteren Schranke durchgeführt. Letzteres wird häufig als trivial angesehen, insbesondere wenn $t_{Dmin,i} = 0$ ist. In diesem Fall werden die zeitlichen Parameter Phase, minimale Prozesszeit, maximale Ausführungszeit und maximale Deadline in folgender Form angegeben:

- $(t_{Ph}; t_{Pmin}; t_{Emax}; t_{Dmax})$

Ist die Phase gleich null, so erfolgt die Darstellung nur mit drei Parametern:

- $(t_{Pmin}; t_{Emax}; t_{Dmax})$

Sind die maximale Deadline und die minimale Periode gleich groß, werden nur zwei Parameter der Task dargestellt:

- $(t_{Pmin}; t_{Emax})$

Damit ergibt sich für unser Beispiel folgende Darstellung der temporalen Parameter der drei Tasks:

- V: (20;5)
- GUI: (40;15)
- MONITORING: (30;10)

Wenn Gruppen von Tasks zusammengefasst werden, sprechen wir auch von einem sogenannten Taskset oder Taskgebilde.

8.2 Nachweis ohne Berücksichtigung der Ressourcen

Im Folgenden wird zunächst der Realzeitnachweis für Systeme, die ein prioritätengesteuertes Scheduling einsetzen, vorgestellt, anschließend für Systeme mit Earliest Deadline First Scheduling. Ziel des Realzeitnachweises in diesem Kapitel ist es, die maximale Verdrängungszeit jeder Task zu bestimmen. Die Berechnung der zusätzlichen maximalen Blockierzeiten einer Task ist Inhalt von Abschnitt 8.3.

8.2.1 Prioritätengesteuertes Scheduling

Der mathematische Realzeitnachweis ist zweistufig. Zuerst wird mittels des *hinreichenden Schedulingtests* untersucht, ob ein Auslastungs-Grenzwert, abhängig vom jeweiligen Scheduling-Verfahren, überschritten wird (Stufe 1). Ergibt das Ergebnis des hinreichenden Schedulingtests, dass der Auslastungs-Grenzwert nicht überschritten wird, kann das System eine schritthaltende Verarbeitung garantieren. Wird der Auslastungs-Grenzwert überschritten, muss in der zweiten Stufe der *notwendige Schedulingtest* durchgeführt werden.

Hinreichender Schedulingtest

Für den hinreichenden Schedulingtest wird zunächst die Formel zur Berechnung der Gesamtauslastung aus Abschnitt 2.2.1 erweitert, indem man im Nenner das Minimum von minimaler Prozesszeit und maximaler Deadline einsetzt. Dadurch erhält man den *hinreichenden Schedulingtest* für ein Einprozessorsystem [Liu2000], der die Utilization u durch alle Tasks n berechnet. Es gilt Gleichung 8-1.

Gleichung 8-1
Hinreichender Schedulingtest

$$u = \sum_{j=1}^{n} \frac{t_{Emax,j}}{\min\left(t_{Dmax,j},\, t_{Pmin,j}\right)} \leq n(2^{\frac{1}{n}} - 1)$$

Hinreichend bedeutet, auch wenn die Gleichung 8-1 nicht erfüllt ist, kann dennoch ein Scheduling-Plan von einem Scheduling-Verfahren mit statischen Prioritäten gefunden werden. Die Wahl des Minimums zwischen maximaler Deadline und minimaler Prozesszeit einer Task im Nenner stellt einen sehr pessimistischen Ansatz dar und spiegelt nicht in jedem Fall die Auslastung des Systems wider. Daher verwenden wir auch die neue Variable u für das Ergebnis des Schedulingtests.

Auf der anderen Seite bedeutet jedoch die Erfüllung der Ungleichung wiederum, dass ein Schedulingplan sicher gefunden wird. Die Grenze, bis zu der die prioritätengesteuerten Scheduling-Verfahren kor-

rekt funktionieren, ist abhängig von der Anzahl der Tasks n auf der CPU (Tabelle 8-1).

n	Auslastungsgrenze (in %)
1	100
2	82,8
3	78
4	75,7
5	74,3
10	71,8
$n \to \infty$	69,3 (ln 2)

Tabelle 8-1
Auslastungsgrenzen des hinreichenden Schedulingtests

Notwendiger Schedulingtest

Liefert der hinreichende Schedulingtest keine Aussage über die Realzeitfähigkeit des Tasksets, so kann mithilfe der Time Demand Analysis (TDA, siehe [Liu2000]) eine notwendige Scheduling- Bedingung aufgesetzt werden.

Das Verfahren basiert auf der Überprüfung der zweiten Realzeitbedingung. Hierzu ist es notwendig, für jede Task die minimale Reaktionszeit $t_{Rmin,i}$ und die maximale Reaktionszeit $t_{Rmax,i}$ zu kennen. Während die minimale Reaktionszeit $t_{Rmin,i}$ im günstigsten Fall identisch mit der minimalen Verarbeitungszeit $t_{Emin,i}$ ist, ist die Bestimmung der maximalen Reaktionszeit $t_{Rmax,i}$ komplizierter.

$$t_{C,i}(t) = \sum_{j \in J} \lceil \frac{t}{t_{Pmin,j}} \rceil * t_{Emax,j}$$
$$J = \{ j \,|\, Prio(T_j) \geq Prio(T_i) \}$$

Gleichung 8-2
Notwendiger Schedulingtest

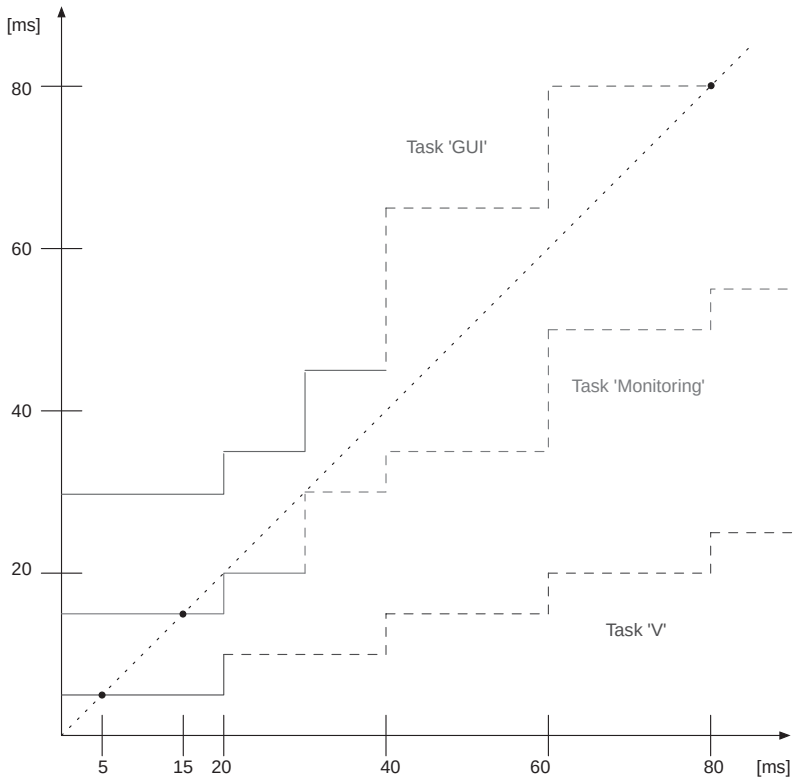
Die Idee bei der Bestimmung der maximalen Reaktionszeit $t_{Rmax,i}$ ist, die Anforderungen an Rechenzeit der gleich- oder höherprioren Tasks aufzusummieren (diese bilden die Menge J) und der zur Verfügung stehenden Rechenzeit gegenüberzustellen. Der Zeitpunkt, zu dem genauso viel Rechenzeit zur Verfügung gestellt worden ist, wie von den betrachteten Tasks benötigt wird, entspricht der maximalen Reaktionszeit $t_{Rmax,i}$ der Task.

Die Summe der Rechenzeitanforderungen von Tasks gleicher oder höherer Priorität kann mithilfe der Gleichung 8-2 für den Worst Case bestimmt werden. J stellt in Gleichung 8-2 die Menge der gleich- oder höherprioren Tasks dar. Die obere Gaußklammer (der mathematische Operator für die nächste Ganzzahl) berechnet die Häufigkeit des Auf-

tretens, die anschließend mit der jeweiligen maximalen Verarbeitungszeit multipliziert wird.

Die bis zum Zeitpunkt zur Verfügung stehende Rechenzeit bei einem Singlecore-Rechner wird durch die Funktion $t_{Available}(t) = t$ beschrieben. Grafisch ist dies die Winkelhalbierende in einem Koordinatensystem, bei dem die X- und Y-Achsen identisch normiert sind.

Abbildung 8-1
TDA Fahrradcomputer Taskset in ms: (20;5), (30;10), (40;15)



Die Zeit t, die die Gleichung $t_{C,i}(t) = t_{Available}(t)$ erfüllt, entspricht dem gesuchten $t_{Rmax,p}$ für Tasks der Menge J. In Abbildung 8-1 ist das Verfahren grafisch visualisiert. Task V (20;5) [ms] hat die höchste Priorität (Prio 1) und stellt alle 20 ms eine Rechenzeitanforderung mit der Höhe 5 ms. Die Höhe der Rechenzeitanforderung der Task erhöht sich periodisch, was in Abbildung 8-1 durch die gestrichelten Linien angedeutet wird. Die Rechenzeitanforderung in Höhe von 10 ms der Task Monitoring wird addiert. Task GUI mit der niedrigsten Priorität und einer Rechenzeitanforderung in Höhe von 15 ms wird zuletzt in der Grafik addiert.

Sobald $t_{C,i}()$ die Winkelhalbierende berührt, sind zu diesem Zeitpunkt alle anstehenden Rechenzeitanforderungen erfüllt, die Reaktion auch der niedrigpriorsten Task ist erfolgt. Daher entspricht dieser Zeitpunkt der maximalen Reaktionszeit der niedrigpriorsten Task. Task V hat eine maximale Reaktionszeit von 5 ms, für die Task Monitoring sind es 15 ms und für die Task GUI sind es 80 ms.

Da die Gleichung $t_{C,i}(t) = t_{Available}(t)$ aufgrund der oberen Gaußklammer nicht durch Umstellen gelöst werden kann, wird ein iteratives Verfahren nach Gleichung 8-3 eingesetzt.

$$t^{(l+1)} = \sum_{j \in M} \lceil \frac{t^{(l)}}{t_{Pmin,j}} \rceil \cdot t_{Emax,j}$$

Gleichung 8-3
TDA: Iterative Lösung

Als Startwert für die Iteration bietet sich die Summe der maximalen Ausführungszeiten $t_{Emax,k}$ der Tasks mit gleicher oder höherer Priorität an. Schließlich kommen im Worst Case höher- oder gleichpriore Tasks mindestens einmal mit ihrer maximalen Verarbeitungszeit zum Zuge. Die Iteration kann abgebrochen werden, wenn gilt:

- $t^{(l+1)} = t^{(l)}$

Das Verfahren gilt für Tasks mit folgender Bedingung: $t_{Dmax,k} \leq t_{Pmin,k}$. Für den Fall, dass die Deadline größer als die Periode ist, also Instanzen einer Task mehrmals bis zur Deadline rechenbereit werden können, muss die TDA zur allgemeinen TDA erweitert werden (vgl. [Liu2000]).

Die drei Tasks des Fahrradcomputers sind in Tabelle 8-2 inklusive ihrer Priorität, der minimalen Deadline und minimalen Ausführungszeit angegeben:

Beispiel 8-1
Bestimmung der maximalen Reaktionszeit

Tabelle 8-2
Beschreibungsgrößen des Fahrradcomputers

RZ-Anforderung	$Phase_i$	$Pmin_i$	$Dmin_i$	$Dmax_i$	$Emin_i$	$Emax_i$	Prio
V	0 ms	20 ms	0 ms	20 ms	1 ms	5 ms	1
*M*ONITORING	0 ms	30 ms	0 ms	30 ms	1 ms	10 ms	2
*G*UI	0 ms	40 ms	0 ms	40 ms	1 ms	15 ms	3

Der hinreichende Schedulingtest unseres Fahrradcomputers ergibt u = 0,9853 und ist somit größer als 78%, dem Grenzwert für drei Tasks gemäß Tabelle 8-1. Damit lässt sich keine Aussage über die Realzeitfähigkeit des Systems treffen.
Nach Gleichung 8-3 berechnen wir die maximale Reaktionszeit jeder Task, wobei wir bei der Task mit höchster Priorität beginnen:

$$V: \; t^{(l+1)} = \sum_{j=1}^{1} \lceil \frac{t^{(l)}}{t_{Pmin,j}} \rceil \cdot t_{Emax,j} = \lceil \frac{t^{(l)}}{t_{Pmin,1}} \rceil \cdot t_{Emax,1} = \lceil \frac{t^{(l)}}{20\,ms} \rceil \cdot 5\,ms$$

Die Menge J (vgl. Gleichung 8-2) der Tasks mit gleicher oder höherer Priorität besteht hier nur aus der Task V selbst. Als Startwert für die Iteration bietet sich die maximale Ausführungszeit an, denn diese fällt im Worst Case mindestens einmal an. Mit $t^{(0)}$ = 5 ms als Startwert ergibt sich $t^{(1)}$ ebenfalls zu 5 ms. Die maximale Reaktionszeit von Task V, die die höchste Priorität hat, beträgt somit 5 ms – was sich einfach nachvollziehen lässt, denn der Scheduler wählt aus der Menge der lauffähigen Tasks immer die Task mit höchster Priorität aus. Wird Task V rechenbereit, unterbricht sie jede andere gerade laufende Task und wird selbst, ohne unterbrochen zu werden, ausgeführt.
Für die Berechnung der maximalen Reaktionszeit von Task M, die die Priorität zwei hat, müssen als gleich- oder höherpriore Tasks, nämlich die Task V (Priorität eins) und M (Priorität zwei), berücksichtigt werden:

$$M: \; t^{(l+1)} = \sum_{j=1}^{2} \lceil \frac{t^{(l)}}{t_{Pmin,j}} \rceil \cdot t_{Emax,j} =$$

$$= \lceil \frac{t^{(l)}}{t_{Pmin,1}} \rceil \cdot t_{Emax,1} + \lceil \frac{t^{(l)}}{t_{Pmin,2}} \rceil \cdot t_{Emax,2} = \lceil \frac{t^{(l)}}{20\,ms} \rceil \cdot 5\,ms + \lceil \frac{t^{(l)}}{30\,ms} \rceil \cdot 10\,ms$$

Mit $t^{(0)}$ = 10 ms als Startwert ergibt sich folgender Verlauf von $t^{(1)}$ (in ms):

l	$t^{(l)}$	$t^{(l+1)}$
0	10	15
1	15	15

Der Abbruch erfolgt bereits nach der zweiten Berechnung. Die Iteration ergibt als maximale Reaktionszeit 15 ms für Task *M*.
Für die Berechnung der maximalen Reaktionszeit der niedrigstprioren Tasks werden schließlich alle Tasks (vgl. Gleichung 8-2) betrachtet:

$$G: \; t^{(l+1)} = \sum_{j=1}^{3} \lceil \frac{t^{(l)}}{t_{Pmin,j}} \rceil \cdot t_{Emax,j} = \lceil \frac{t^{(l)}}{t_{Pmin,1}} \rceil \cdot t_{Emax,1} + \lceil \frac{t^{(l)}}{t_{Pmin,2}} \rceil \cdot t_{Emax,2} + \lceil \frac{t^{(l)}}{t_{Pmin,3}} \rceil \cdot t_{Emax,3}$$

$$= \lceil \frac{t^{(l)}}{20\,ms} \rceil \cdot 5\,ms + \lceil \frac{t^{(l)}}{30\,ms} \rceil \cdot 10\,ms + \lceil \frac{t^{(l)}}{40\,ms} \rceil \cdot 15\,ms$$

Mit $t^{(0)}$ = 15 ms als Startwert ergibt sich folgender Verlauf von $t^{(1)}$ (in ms):

l	$t^{(l)}$	$t^{(l+1)}$
0	15	30
1	30	35
2	35	45
3	45	65
4	65	75
5	75	80
6	80	80

Die Iteration ergibt als maximale Reaktionszeit 80 ms für Task *G*.
Die mathematisch bestimmten maximalen Reaktionszeiten entsprechen den Schnittpunkten aus Abbildung 8-1.

Sind die maximalen Reaktionszeiten bestimmt, muss noch für alle Rechenzeitanforderungen die zweite Realzeitbedingung überprüft werden.

Beispiel 8-2
Fahrradcomputer: Anwendung der zweiten Realzeitbedingung

Mit Kenntnis der minimalen und der maximalen Reaktionszeiten lässt sich die zweite Realzeitbedingung überprüfen:
V: 0 ms ≤ 1 ms ≤ 5 ms ≤ 20 ms; Bedingung erfüllt
MONITORING: 0 ms ≤ 1 ms ≤ 15 ms ≤ 30 ms; Bedingung erfüllt
GUI: 0 ms ≤ 1 ms ≤ 80 ms ≤ 40 ms; Bedingung nicht erfüllt
Für Task GUI ist die zweite Realzeitbedingung nicht erfüllt, eine schritthaltende Verarbeitung ist bei Einsatz eines prioritätengesteuerten Schedulings also nicht möglich.

Zu beachten ist, dass eine Deadline-Verletzung bei jeder Task auftreten kann, nicht nur bei der Task mit der niedrigsten Priorität.

8.2.2 EDF-Scheduling

Earliest Deadline First, was manchmal auch kürzer als Deadline Scheduling bezeichnet wird, arbeitet optimal für *unterbrechbare Tasks* auf einem *Singlecore-System*, wovon wir im Weiteren ausgehen werden.

Hinreichender Schedulingtest

Der hinreichende Schedulingtest ergibt sich zu (vgl. [Liu2000]):

Gleichung 8-4
Hinreichender Schedulingtest

$$u = \sum_{j=1}^{n} \frac{t_{Emax,j}}{\min\left(t_{Dmax,j},\, t_{Pmin,j}\right)} \leq 1$$

Wenn die maximale Deadline aller Tasks größer oder gleich der Periode ist ($t_{Dmax,j} \geq t_{Pmin,j}$), reduziert sich Gleichung 8-4 auf die erste Realzeitbedingung, bei der die Auslastung in einem Einprozessorsystem kleiner oder gleich eins sein muss. In diesem Fall stellt die Auslastungsbedingung eine hinreichende und notwendige Bedingung dar, da EDF in einem Einprozessorsystem optimal arbeitet. Anders verhält es sich im Fall $t_{Dmax,j} < t_{Pmin,j}$. Gleichung 8-4 stellt nun eine hinreichende Bedingung dar. Wir können also lediglich sagen, dass bei Nichterfüllung der Ungleichung eventuell kein schritthaltendes Scheduling mehr möglich ist. Für eine detaillierte Analyse müssen wir auch hier den notwendigen Schedulingtest durchführen.

Notwendiger Schedulingtest

Ein exakter Schedulingtest in einem System mit periodischen und sporadischen Tasks lässt sich mit dem in [Gresser93] vorgestellten Ereignisstrom-Modell durchführen. Beim Ereignisstrom-Modell handelt es sich um eine formale Methode zur Beschreibung des Auftretens von Ereignissen wie Zeitpunkt, Häufigkeit und Abhängigkeiten zwischen Ereignissen. Das Modell besteht aus mehreren Komponenten, von denen zwei für periodische Tasks von besonderem Interesse sind:

1. Ereignisdichtefunktion und
2. Rechenzeitanforderungsfunktion.

Das Ereignisstrommodell bildet die Zeitparameter auf Intervall I ab. Dazu werden die temporalen Parameter einer Task in eine entsprechende Ereignisdichtefunktion transformiert, woraus sich die Rechenzeitanforderungsfunktion $t_C(I)$ ergibt.

Um die Ereignisdichtefunktion (Ereignisfunktion) zu bestimmen, muss die minimale Prozesszeit bekannt sein. Für jede Intervallgröße I von 0 bis ∞ bestimmt man die im Intervall I maximal auftretende Anzahl von Rechenzeitanforderungen (Ereignissen). Für periodische Prozesse, die für Realzeitsysteme typisch sind, ist das sehr einfach. Innerhalb der Periode der Task tritt genau eine Rechenzeitanforderung auf, nämlich die Rechenzeitanforderung zum Starten der Task.

Mathematisch wird die Ereignisdichtefunktion durch Gleichung 8-5 beschrieben. Um die Anzahl der Rechenzeitanforderungen in einem Intervall I zu berechnen, wird zur Intervallbreite die minimale Prozesszeit addiert und die Phase subtrahiert. Die erhaltene Summe wird durch die minimale Prozesszeit dividiert. Auf dieses Ergebnis wird die untere

Gaußklammer, also nur der ganzzahlige Anteil (Funktion `floor()`), angesetzt. Die grafische Darstellung findet sich in Abbildung 8-2.

$$E_i(I) = \lfloor \frac{I + t_{Pmin,i} - t_{Ph,i}}{t_{Pmin,i}} \rfloor$$

Gleichung 8-5 *Ereignisdichtefunktion*

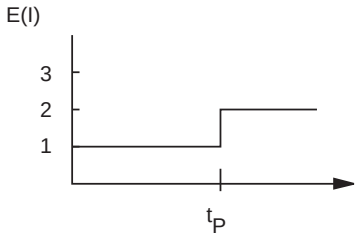


Abbildung 8-2 *Ereignisdichtefunktion einer periodischen Task*

Für einmalig (also nicht periodisch) auftretende Rechenzeitanforderungen findet der Wert unendlich (∞) als Periode Anwendung, wodurch sich als Ereignisdichtefunktion E(I) = 1 ergibt.

Die Rechenzeitanforderungsfunktion in Gleichung 8-6 wird berechnet, indem die Ereignisdichte mit der Ausführungszeit multipliziert und um die maximale Deadline nach rechts verschoben wird.

$$t_{C,i}(I) = E(I - t_{Dmax,i}) = \lfloor \frac{I + t_{Pmin,i} - t_{Dmax,i} - t_{Ph,i}}{t_{Pmin,i}} \rfloor \cdot t_{Emax,i}$$

Gleichung 8-6 *Rechenzeitanforderungsfunktion in Abhängigkeit von I*

Die Rechenzeitanforderungsfunktion in Abhängigkeit von I muss für alle Rechenzeitanforderungen aufsummiert werden. Zur Überprüfung, ob die maximal zulässige Deadline für alle Rechenzeitanforderungen eingehalten wird, muss die Bedingung $t_{C,gesamt}(I) \leq I$ erfüllt sein (siehe Gleichung 8-7). Grafisch veranschaulicht bedeutet dies, dass die Rechenzeitanforderungsfunktion die Winkelhalbierende nicht überschreiten darf. Umgekehrt gibt der Abstand von der Winkelhalbierenden an, welche Auslastung der Rechner durch die Bearbeitung der Tasks erfährt.

$$t_{C,ges}(I) = \sum_{j=1}^{n} \lfloor \frac{I + t_{Pmin,j} - t_{Dmax,j} - t_{Ph,j}}{t_{Pmin,j}} \rfloor \cdot t_{Emax,j} \leq I$$

Gleichung 8-7 *Notwendiger Schedulingtest*

Die Hyperperiode, die sich aus dem kleinsten gemeinsamen Vielfachen der $t_{Pmin,i}$ ergibt, bestimmt zunächst den maximalen Bereich, in dem die Bedingung $t_{C,gesamt}(I) \leq I$ überprüft werden muss. Kommen jedoch nicht alle Rechenzeitanforderungen gleichzeitig zum Zeitpunkt t = 0, existiert

also eine oder auch mehrere Phasen mit $t_{Ph} \neq 0$, ist der zu untersuchende Bereich gemäß Gleichung 8-8 auszuweiten:

Gleichung 8-7 ist zu untersuchen für alle I im Bereich von $0 < I < kgV(t_{Pmin,i}) + \max(t_{Ph,i})$.

Allerdings muss glücklicherweise nicht jedes der unendlich vielen Intervallbreiten I untersucht werden. Da es sich bei $t_{C,gesamt}(I)$ um eine Treppenfunktion handelt, sind nur die einzelnen Sprungstellen relevant. Diese wiederum ergeben sich durch Untersuchung der einzelnen Gaußklammern der aufgestellten Funktion $t_{C,gesamt}(I)$. Für all diejenigen I, für die eine solche Gaußklammer den Wert ändert, ist eine Berechnung durchzuführen.

Gleichung 8-8
Modifizierte Hyperperiode

$$H_{mod} = H + \max_{0 \leq j \leq n} (t_{Ph,j})$$

Realzeitnachweis beim EDF-Scheduling

Im Folgenden soll der Realzeitnachweis für den bereits mehrfach eingesetzten Fahrradcomputer im Fall eines EDF-Schedulings durchgeführt werden. Die zugehörigen Beschreibungsgrößen finden Sie in Tabelle 8-2.

Der hinreichende Schedulingtest des Fahrradcomputers für dynamische Prioritäten ergibt u = 0,9853 und ist somit kleiner als 1 = 100%. Da für alle Tasks gilt: $t_{Pmin,i} \geq t_{Dmax,i}$, kann jetzt bereits die Aussage getroffen werden, dass es sich bei den drei periodischen Tasks um ein Realzeitsystem handelt.

Wir überprüfen dies mit dem Schedulingtest. Zur Veranschaulichung sind in Abbildung 8-3 die Rechenzeitanforderungsfunktionen der drei Tasks und deren Summe dargestellt. Die Hyperperiode beträgt 120 ms, entsprechend muss das Intervall [0,120] ms untersucht werden. Da die Summe aller Rechenzeitanforderungen (oberste Linie) im zu untersuchenden Intervall unterhalb der Winkelhalbierenden bleibt, kann das System die Realzeitbedingungen einhalten. Würde innerhalb des zu untersuchenden Intervalls die Rechenzeitanforderung oberhalb der vom System zur Verfügung gestellten Rechenleistung liegen, wäre das System überlastet und somit könnte keine schritthaltende Verarbeitung garantiert werden.

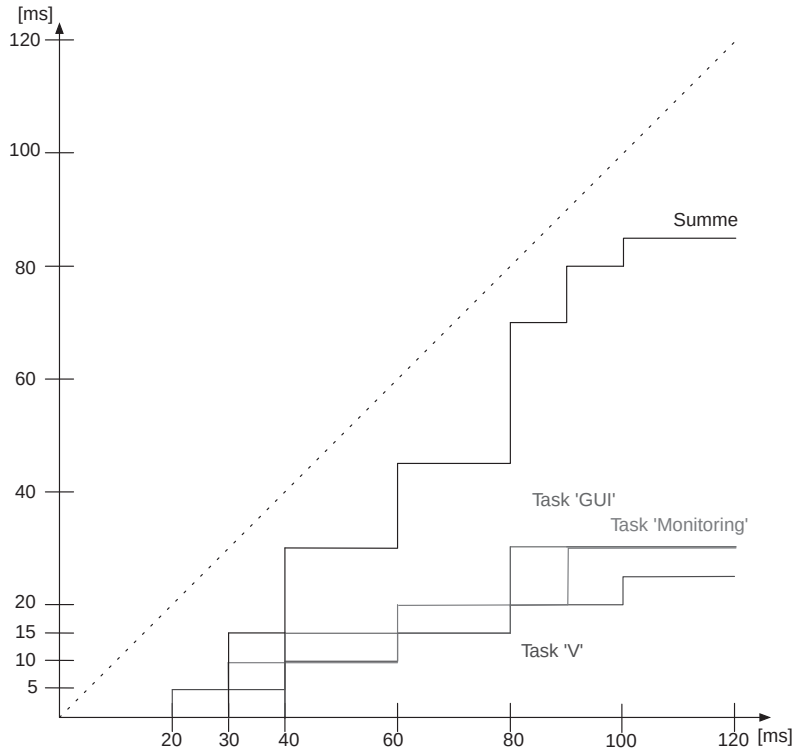


Abbildung 8-3
Fahrradcomputer bei EDF-Scheduling

Rechnerisch muss dafür Gleichung 8-7 für die Intervalle I = 20 ms, 30 ms, 40 ms, 60 ms, 80 ms, 90 ms, 100 ms, 120 ms bestimmt werden. Die Sprünge stellen Vielfache der einzelnen Perioden der Tasks dar.

$$t_{C,ges}(I) = \sum \lfloor \frac{I + t_{Pmin,i} - t_{Dmax,i} - t_{Ph,i}}{t_{Pmin,i}} \rfloor \cdot t_{Emax,i}$$

$$t_{C,ges}(I) = \lfloor \frac{I + 20ms - 20ms - 0ms}{20ms} \rfloor \cdot 5ms + \lfloor \frac{I + 30ms - 30ms - 0ms}{30ms} \rfloor \cdot 10ms$$

$$+ \lfloor \frac{I + 40ms - 40ms - 0ms}{40ms} \rfloor \cdot 15ms$$

$$t_{C,ges}(I) = \lfloor \frac{I}{20ms} \rfloor \cdot 5ms + \lfloor \frac{I}{30ms} \rfloor \cdot 10ms + \lfloor \frac{I}{40ms} \rfloor \cdot 15ms$$

I ist zu untersuchen im Bereich von:

$0 < I < kgV(20\ ms, 30\ ms, 40\ ms) + max(0\ ms, 0\ ms, 0\ ms) = 120\ ms$

Die einzelnen I, die zu untersuchen sind, ergeben sich aus:

I_V: {20 ms, 40 ms, 60 ms, 80 ms, 100 ms}
I_M: {30 ms, 60 ms, 90 ms}
I_G: {40 ms, 80 ms}

zu:

I : {20 ms, 30 ms, 40 ms, 60 ms, 80 ms, 90 ms, 100 ms}

Damit ergibt sich:

$$t_{C,ges}(20ms) = \lfloor\frac{20ms}{20ms}\rfloor \cdot 5ms + \lfloor\frac{20ms}{30ms}\rfloor \cdot 10ms + \lfloor\frac{20ms}{40ms}\rfloor \cdot 15ms = 5ms$$

$$t_{C,ges}(30ms) = \lfloor\frac{30ms}{20ms}\rfloor \cdot 5ms + \lfloor\frac{30ms}{30ms}\rfloor \cdot 10ms + \lfloor\frac{30ms}{40ms}\rfloor \cdot 15ms = 15ms$$

$$t_{C,ges}(40ms) = \lfloor\frac{40ms}{20ms}\rfloor \cdot 5ms + \lfloor\frac{40ms}{30ms}\rfloor \cdot 10ms + \lfloor\frac{40ms}{40ms}\rfloor \cdot 15ms = 35ms$$

$$t_{C,ges}(60ms) = \lfloor\frac{60ms}{20ms}\rfloor \cdot 5ms + \lfloor\frac{60ms}{30ms}\rfloor \cdot 10ms + \lfloor\frac{60ms}{40ms}\rfloor \cdot 15ms = 50ms$$

$$t_{C,ges}(80ms) = \lfloor\frac{80ms}{20ms}\rfloor \cdot 5ms + \lfloor\frac{80ms}{30ms}\rfloor \cdot 10ms + \lfloor\frac{80ms}{40ms}\rfloor \cdot 15ms = 70ms$$

$$t_{C,ges}(90ms) = \lfloor\frac{90ms}{20ms}\rfloor \cdot 5ms + \lfloor\frac{90ms}{30ms}\rfloor \cdot 10ms + \lfloor\frac{90ms}{40ms}\rfloor \cdot 15ms = 80ms$$

$$t_{C,ges}(100ms) = \lfloor\frac{100ms}{20ms}\rfloor \cdot 5ms + \lfloor\frac{100ms}{30ms}\rfloor \cdot 10ms + \lfloor\frac{100ms}{40ms}\rfloor \cdot 15ms = 85ms$$

Die Bedingung $t_{C,gesamt}(I) \leq t$ ist für alle zu untersuchenden I erfüllt, die maximal zulässige Deadline wird von allen Rechenzeitanforderungen eingehalten.

Jetzt muss noch überprüft werden, ob auch die minimal zulässige Deadline $t_{Dmin,i}$ für alle Rechenzeitanforderungen eingehalten wird.

$$t_{Dmin,i} \leq t_{Rmin,i}$$

$$V: \quad 0ms \leq 1ms$$

$$M: \quad 0ms \leq 1ms$$

$$G: \quad 0ms \leq 1ms$$

Da alle Bedingungen eingehalten werden, ist das System realzeitfähig.

Realzeitnachweis bei Berücksichtigung der Phase

Haben die Tasks eine Phase zueinander, hängt also eine Task von einer anderen ab, entschärft das die Situation, da nicht mehr alle Rechenzeitanforderungen gleichzeitig kommen. Um das Verfahren zu zeigen, wollen wir dennoch den Realzeitnachweis durchführen. Grafisch bedeutet die Phase, dass die Tasks mit der Phase eine zusätzliche Rechtsverschiebung erfahren (siehe Gleichung 8-6). Wenn wir annehmen, dass die GUI-Task gegenüber der V-Task eine Phase von 10 ms hat, ergibt sich das Taskset zu:

- V(20;5)
- M: (30;10)
- GUI (10;40;15;40)

Durch die Phase muss die Hyperperiode modifiziert werden zu:

$H_{Mod} = H + t_{Ph,3} = 120 + 10 = 130$

Die Rechenzeitanforderungsfunktion ergibt sich dann für die Hyperperiode = 130 ms zu:

$$t_{C,ges}(I) = \lfloor \frac{I + 20ms - 20ms - 0ms}{20ms} \rfloor \cdot 5ms + \lfloor \frac{I + 30ms - 30ms - 0ms}{30ms} \rfloor \cdot 10ms$$

$$+ \lfloor \frac{I + 40ms - 40ms - 10ms}{40ms} \rfloor \cdot 15ms$$

$$t_{C,ges}(I) = \lfloor \frac{I}{20ms} \rfloor \cdot 5ms + \lfloor \frac{I}{30ms} \rfloor \cdot 10ms + \lfloor \frac{I - 10ms}{40ms} \rfloor \cdot 15ms$$

I ist zu untersuchen im Bereich von $0 < I < kgV(20ms, 30ms, 40ms) + max(0ms, 0ms, 0ms) = 120ms$

I_V: {20ms, 40ms, 60ms, 80ms, 100ms}

I_M: {30ms, 60ms, 90ms}

I_G: {50ms, 90ms}

I: {20ms, 30ms, 40ms, 50ms, 60ms, 80ms, 90ms, 100ms}

Die Berechnung ergibt dann folgende Werte für die zu untersuchenden Intervalle I:

I	$t_{C,gesamt}(I)$
20 ms	5 ms
30 ms	15 ms
40 ms	20 ms
50 ms	35 ms
60 ms	50 ms
80 ms	55 ms
90 ms	80 ms

Schritthaltender Betrieb ist damit wie erwartet möglich.

8.3 Nachweis unter Berücksichtigung der Ressourcen

Der Realzeitnachweis unter Berücksichtigung von Ressourcen läuft zweistufig ab. Zunächst müssen die Blockierzeiten für die einzelnen Tasks bestimmt werden, die sich durch den Schutz kritischer Abschnitte ergeben. Mit diesen Daten kann schließlich der eigentliche Schedulingtest durchgeführt werden.

8.3.1 Berechnung der Blockierzeit

Vollständig unabhängige Tasks kommen in der Realität sehr selten vor, meist teilen sie untereinander Ressourcen, deren notwendiger Schutz wiederum zu einem geänderten Zeitverhalten führt. Um in diesem Fall einen Realzeitnachweis führen zu können, sind in einem ersten Schritt die sich ergebenden Blockierzeiten zu bestimmen. Im zweiten Schritt wird dann mithilfe der Blockierzeiten und der übrigen Parameter der eigentliche Realzeitnachweis geführt.

Die Bestimmung der Blockierzeiten wird durch die sich ergebende Prioritätsinversion erschwert. So muss der jeweils zum Einsatz kommende Algorithmus zur Prioritätsvererbung (beispielsweise die Unterbrechungssperre, PIP oder PCP) berücksichtigt werden.

Zur mathematischen Bestimmung der maximalen Blockierzeiten einer Task erweitern wir unseren Fahrradcomputer um eine weitere Task und portieren das System auf eine schnellere Hardware. Außerdem benutzen die Tasks nun gemeinsame Ressourcen. Das zu untersuchende System besteht insgesamt aus vier Tasks, die drei unterschiedliche Ressourcen verwenden. Die Tasks benutzen die drei Ressourcen in der in

Abbildung 8-4 dargestellten Weise; die sich daraus ergebende Task-Ressourcen-Zuordnung ist in Tabelle 8-3 in Matrizenform dargestellt. Diese Matrix enthält in der letzten Spalte bereits die Priorität einer Ressource. Diese entspricht der Task, die die Ressource verwendet und die jeweils höchste Priorität hat.

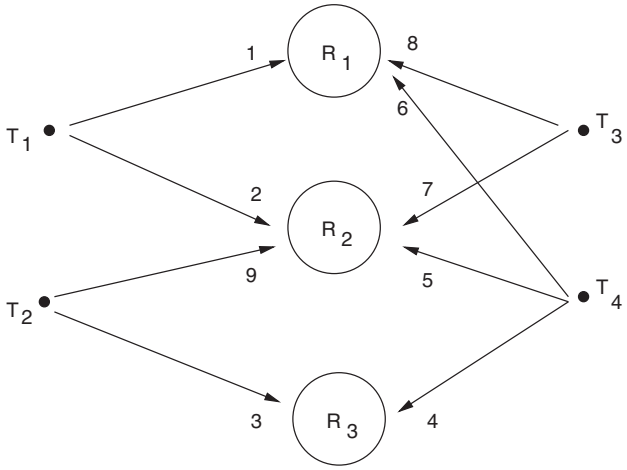


Abbildung 8-4
Ressourcengraph am Beispiel des Fahrradcomputers

Tasks	R_1	R_2	R_3
T_1	1	2	0
T_2	0	9	3
T_3	8	7	0
T_4	6	5	4
$\prod(R)$	T_1	T_1	T_2

Tabelle 8-3
Zuordnung von Tasks und Ressourcen am Beispiel des Fahrradcomputers

Für die Matrix wird in jeder Zeile die Länge der kritischen Bereiche (t_{CS}) der verschiedenen Ressourcen für jede Task notiert, für ein prioritätengesteuertes Scheduling abfallend nach Prioritäten der Tasks. T_1 hat die höchste Priorität, T_4 die niedrigste.

In der letzten Zeile werden die sich daraus ergebenden Prioritäten für die Ressourcen eingetragen. Und zwar bekommt eine Ressource die Priorität der höchsten zugreifenden Task zugeteilt. So ergibt sich beispielsweise für R_3 aufgrund der Zuordnung die Priorität von T_2, da diese Task die mit der höchsten Priorität ist, die auf R_3 zugreift. Für die Berechnung der maximalen Blockierzeiten brauchen wir nur die Länge der jeweiligen kritischen Bereiche. Die restlichen temporalen Parameter der Tasks werden beim Schedulingtest in Abschnitt 8.3.2 vorgestellt, da sie auch dort erst benötigt werden.

8.3.1.1 Blockierzeiten bei Unterbrechungssperren

Beim Schutz der kritischen Bereiche durch eine Unterbrechungssperre (Seite 28) (*N*on *P*reemptible *C*ritical *S*ection, NPCS) lässt sich mit Gleichung 8-9 [Buttazzo1997] die Blockierzeit einfach bestimmen.

Gleichung 8-9
NPCS-Blockierzeit mit statischen Prioritäten

$$t_{B,i} = \max_{i+1 \leq j \leq n} (t_{CS,j})$$

Dabei ist n die Anzahl der periodischen Tasks im System, wobei die Tasks mit fallender Priorität aufsteigend indiziert sind.

Eine Task kann nur durch eine Task mit niedrigerer Priorität aufgrund einer Ressource für die Länge t_{CS} blockiert werden. Daher kommen für eine Task i nur die niederprioren Tasks zwischen `i + 1` und `n` infrage.

Für ein System mit konstanten Prioritäten lässt sich nun die Blockierzeit t_B für jede Task aus Tabelle 8-3 berechnen.

T_1

Es kommen als mögliche Verursacher von Blockierzeiten nur die niederprioren Tasks T_2 bis T_4 in die Auswahl. Das Maximum der kritischen Bereiche ergibt sich zu:

$t_{B,1}$= max (0,9,3,8,7,0,6,5,4) = 9

T_2

Als Verursacher kommen nur die Tasks T_3 und T_4 infrage:

$t_{B,2}$= max (8,7,0,6,5,4) = 8

T_3

Als Verursacher einer Blockierzeit kann nur noch T_4 in Frage kommen:

$t_{B,3}$= max (6,5,4) = 6

T_4

Es gibt keine niederprioren Tasks mehr, somit kann für T_4 auch keine Blockierzeit aufgrund von Ressourcenzugriffen auftreten:

$t_{B,4}$= 0

Bei einem EDF-Scheduling sind kritische Abschnitte sämtlicher Tasks zu berücksichtigen. Aus der infrage kommenden Menge der kritischen Bereiche bestimmt der längste die maximale Blockierzeit der Task.

$$t_{B,i} = \max_{i \le j \le n \mid j \neq i} (t_{CS,j})$$

Gleichung 8-10
NPCS-Blockierzeit mit EDF-Scheduling

T_1

Es kommen als mögliche Verursacher die Tasks T_2 bis T_4 in die Auswahl. Das Maximum der kritischen Bereiche ergibt sich zu:

$t_{B,1}$= max (0,9,3,8,7,0,6,5,4) = 9

T_2

Als Verursacher kommen die Tasks T_1, T_3 und T_4 infrage:

$t_{B,2}$= max (1,2,0,8,7,0,6,5,4) = 8

T_3

Als Verursacher kommen die Tasks T_1, T_2 und T_4 infrage:

$t_{B,3}$= max (1,2,0,0,9,3,6,5,4) = 9

T_4

Als Verursacher kommen die Tasks T_1 bis T_3 infrage:

$t_{B,4}$= max (1,2,0,0,9,3,8,7,0) = 9

8.3.1.2 PIP

Bei der klassischen Prioritätsvererbung (PIP) werden zunächst zwei Blockierzeiten unterschieden [Buttazzo1997]: die direkte (Task Blocking) und die indirekte (Ressource Blocking) Blockierzeit.

Direktes Blockieren. Das direkte Blockieren ergibt sich, wenn die untersuchte Task i auf eine Ressource R zugreifen möchte, die bereits durch eine andere, niederpriore Task gehalten wird. Folglich müssen alle Tasks mit niedrigerer Priorität ($i+1 \le j \le n$) betrachtet werden, die eine Ressource mit der Task i teilen. Jede dieser Tasks kann Task i über die gemeinsame Ressource blockieren, und das im ungünstigsten Fall mit der jeweils längsten Blockierzeit. Daher ist für jede niederpriore Task T_j ($i+1 \le j \le n$) die maximale Blockierzeit der gemeinsamen Ressource ($\prod(R_k) \geq \pi_i$) aufzusummieren.

$$t_{BT,i} = \sum_{j=i+1}^{n} \max_{k} \{t_{CS,jk} \mid \Pi(R_k) \geq \pi_i\}$$

Gleichung 8-11
PIP: Direkte Blockierzeit

Gleichung 8-11 gibt die Berechnung mathematisch wieder. Der Index i und der Index j stehen für die Tasks gemäß ihrer Priorität, wobei eins eine hohe Priorität bedeutet. Der Index k steht für die Ressourcen. Es wird für jede einzelne Task, die niedrigere Priorität hat (dafür steht in der Formel das Summenzeichen), jeweils das Maximum der Blockierzei-

ten genommen (Funktion `max()`). Allerdings müssen nur die Ressourcen berücksichtigt werden, die von der eigenen Task oder von höherprioren Tasks verwendet werden. Hierfür wird in der Formel die Priorität der Ressource eingesetzt (Ressourcenpriorität: $\prod(R_k)$).

An dieser Stelle ist die Gleichung auch missverständlich. Damit die Blockierzeit einer Ressource für die max-Funktion berücksichtigt wird, muss die Priorität der Ressource größer oder gleich der Priorität der Task i sein. Die Priorität ist allerdings dann größer, wenn die Zahlen kleiner sind. Auf Zahlenebene müsste also an dieser Stelle ein Kleinergleich-Zeichen stehen.

In der Zuordnungsmatrix werden für die Ermittlung der direkten Blockierzeit somit alle Zeilen untersucht, die aufgrund der Priorität infrage kommen, und es wird das Maximum jeder dieser Zeilen bestimmt.

Indirektes Blockieren. Das indirekte Blockieren ergibt sich durch höherpriore Tasks, die die untersuchte Task i zusätzlich (nicht nur aufgrund der im Realzeitnachweis bereits betrachteten Verarbeitungszeit) verzögern, indem sie selbst auf einer Ressource schlafen müssen, die eine andere Task hält. Somit müssen auch die Blockierzeiten der von den höherprioren Tasks benutzten Ressourcen untersucht werden, allerdings hiervon nur diejenigen, die auch von den niederprioren Tasks verwendet werden.

Gleichung 8-12
PIP: Indirekte Blockierzeit

$$t_{BR,i} = \sum_{k=1}^{m} \max_{j>i} \{t_{CS,jk} \mid \Pi(R_k) \geq \pi_i\}$$

Der Index im Summenzeichen deutet bei Gleichung 8-12 bereits an, dass hier jeweils eine Ressource, also die Spalte, einen Summanden für die Summenbildung liefert. Der Summand ist wieder das Maximum der Länge eines kritischen Abschnittes, wobei nur die Ressourcen berücksichtigt werden, deren Ressourcenpriorität höher ist als die Priorität der Task i.

Es wird also das Maximum der infrage kommenden Spalten gebildet.

Eine Task kann im Worst Case aufgrund der Verwendung von Ressourcen von niederprioren Tasks sowohl direkt als auch indirekt blockiert werden. Da die beiden Formeln die Arten der Blockierung nicht vollständig unabhängig voneinander berechnen, ist nur die kleinere der beiden Zeiten relevant. Es gilt somit Gleichung 8-13.

$$t_{B,i} = \min\{t_{BT,i}\,;\, t_{BR,i}\}$$

Gleichung 8-13
PIP: Blockierzeit bei prioritätengesteuertem Scheduling

Nach Gleichung 8-11 und Gleichung 8-12 ergibt sich mithilfe von Tabelle 8-3 für die einzelnen Tasks:

T_1

Entsprechend der Formel für die *Task-Blocking*-Zeit werden alle signifikanten Stellen niederpriorer Tasks (also die Reihen von T_2 bis T_4) untersucht und das Maximum aus diesen Stellen pro Reihe ermittelt. Interessant sind für T_1 die 1. und 2. Spalte jeder Reihe, da nur hier die Bedingung $\prod(R_k) \geq \pi_i$ erfüllt ist. Damit ergibt sich $t_{BT,1}$ zu:

```
max (0,9) + max (8,7) + max (6,5) = 9 + 8 + 6 = 23
```

Für die Blockierzeit aufgrund der Ressourcen (*Ressource Blocking*) werden nun die relevanten Spalten (also wieder die 1. und 2.) der niederprioren Tasks (siehe `j>i` in Formel) untersucht. Damit ergibt sich $t_{BR,1}$ zu:

```
max(0,8,6) + max(9,7,5) = 8 + 9 = 17
```

T_2

Nun werden für $t_{BT,2}$ die gesamten Reihen von T_3 und T_4 interessant. Alle Ressourcen (also alle Spalten) müssen nun betrachtet werden, da $\prod(R_k) \geq \pi_i$ erfüllt ist. Damit ergibt sich $t_{BT,2}$ zu:

```
max(8,7,0) + max(6,5,4) = 8 + 6 = 14
```

Auch für die Resource Blocking Time werden nun alle drei Spalten betrachtet. Damit ergibt sich $t_{BR,2}$ zu:

```
max(8,6) + max(7,5) + max(0,4) = 8 + 7 + 4 = 19
```

T_3

Nach den Überlegungen über T_1 und T_2 ergibt sich für $t_{BT,3}$:

```
max(6,5,4) = 6
```

und für $t_{BR,3}$:

```
6 + 5 + 4 = 15
```

T_4

Für T_4 steht keine Zeile/Spalte zur Untersuchung an. Somit ergibt sich

$t_{BT,4} = t_{BR,4} = 0$

Entsprechend Gleichung 8-13 ergeben sich die Blockierzeiten für die Tasks zu:

- $t_{B,1} = 17$
- $t_{B,2} = 14$

- $t_{B,3} = 6$
- $t_{B,4} = 0$

Die Berechnung der Blockierzeiten bei Scheduling-Verfahren, die nicht auf konstanten Prioritäten beruhen, wie beispielsweise beim EDF-Scheduling, lässt sich auf ähnliche Weise bestimmen. Es wird wieder die Matrix mit den Tasks als Zeilen und den Ressourcen als Spalten aufgestellt. Die Spalte bezüglich der Ressourcenpriorität wird ausgelassen.

Da jede Task damit rechnen muss, auf jede andere Task im System bezüglich der betrachteten Ressource warten zu müssen, ergibt sich für die direkte Blockierzeit $t_{BT,i}$ und für die indirekte Blockierzeit $t_{BR,i}$Gleichung 8-14.

Für die direkte Blockierzeit wird von jeder Zeile ($1 \leq j \leq n$), mit Ausnahme der Zeile der Task i ($j \leq i$), das Maximum über alle Blockierzeiten (k), sofern die betrachtete Task i überhaupt die Ressource k nutzt ($t_{CS,ik} \neq 0$), aufsummiert.

Für die indirekte Blockierzeit wird über alle Ressourcen (Spalten), die von der Task i verwendet werden ($t_{CS,ik} \neq 0$), das Maximum der Blockierzeiten $t_{CS,jk}$ aller Tasks j aufsummiert, wobei die Blockierzeit der betrachteten Task i ausgenommen ist ($j \leq i$).

Gleichung 8-14
Blockierzeiten beim EDF-Scheduling

$$t_{BT,i} = \sum_{1 \leq j \leq n \mid j \neq i} \max_{k \mid t_{CS,i,k} \neq 0} (t_{CS,j,k})$$

$$t_{BR,i} = \sum_{k \mid t_{CS,i,k} \neq 0} \max_{j \mid j \neq i} (t_{CS,j,k})$$

$$t_{B,i} = \min(t_{BT,i}, t_{BR,i})$$

Da in unserem Beispiel jede Ressource von den Tasks einzeln und nicht verschränkt benutzt wird, können die Blockierzeiten $t_{B,i}$ auch pragmatisch in Analogie zu der Berechnung für ein prioritätengesteuertes Scheduling bestimmt werden (vgl. Gleichung 8-14).

Werden mehrere Ressourcen von einer Task gleichzeitig benutzt (verschränkter Zugriff), so ist darauf zu achten, dass die Ressource mit der höchsten Priorität die Ressourcenpriorität für alle von dieser Task gleichzeitig benutzten Ressourcen bestimmt. In diesem Fall ist eine Bestimmung der Priorität der Ressourcen wie beim prioritätengesteuerten Scheduling nötig.

In unserem Beispiel werden nun für die jeweilige Task die Blockierzeiten berechnet, indem die zuvor aufgestellte Tabelle so umsortiert

wird, dass jeweils die zu betrachtende Task i oben steht, quasi als hätte sie die höchste Priorität.

T_1 hat damit die gleiche Blockierzeit wie schon beim prioritätengesteuerten Scheduling (statisches Scheduling).

Für T_2 bis T_4 ergeben sich die folgenden Matrizen, die wir durch Umsortieren erhalten. Wir sortieren sie so, dass die Task i, deren Blockierzeiten bestimmt werden sollen, an oberster Stelle steht.

T_2

Tasks	R_1	R_2	R_3
T_2	0	9	3
T_1	1	2	0
T_3	8	7	0
T_4	6	5	4

$t_{BT,2}$ wird durch Betrachtung aller anderen Tasks (T_1, T_3 und T_4) bestimmt: Die 2. und 3. Spalte wird nur berücksichtigt, da T2 nicht auf R_1 zugreift. Damit ergibt sich $t_{BT,2}$ zu:

```
max(2,0) + max(7,0) + max(5,4) = 2 + 7 + 5 = 14
```

Für die Resource Blocking Time werden die 2. und 3. Spalte betrachtet. Damit ergibt sich $t_{BR,2}$ zu:

```
max(2,7,5) + max(0,0,4) = 7 + 4 = 11
```

T_3

Tasks	R_1	R_2	R_3
T_3	8	7	0
T_1	1	2	0
T_2	0	9	3
T_4	6	5	4

Für $t_{BT,3}$ ergibt sich unter Beachtung der Tasks T_1, T_2 und T_4 sowie der 1. und 2. Spalte:

```
max(1,2) + max(0,9) + max(6,5) = 2 + 9 + 6 = 17
```

Für die Resource Blocking Time werden die 1. und 2. Spalte betrachtet. Damit ergibt sich $t_{BR,2}$ zu:

```
max(1,0,6) + max(2,9,5) = 6+9 = 15
```

T_4

Im Gegensatz zur statischen Prioritätenverteilung kann bei einem Scheduling, das nicht auf konstanten Prioritäten beruht, auch T_4

vorrangig ausgewählt werden. Analog zur Berechnung von den Blockierzeiten von T_2 und T_3 ergibt sich für T_4:

Tasks	R_1	R_2	R_3
T_4	6	5	4
T_1	1	2	0
T_2	0	9	3
T_3	8	7	0

Für $t_{BT,4}$ ergibt sich:

```
max(1,2,0) + max(0,9,3) + max(8,7,0) = 2 + 9 + 8 = 19
```

Für die Resource Blocking Time ergibt sich $t_{BR,2}$ zu:

```
max(1,0,8) + max(2,9,7) + max(0,3,0) = 8 + 9 + 3 = 20
```

Entsprechend Gleichung 8-13 ergeben sich die Blockierzeiten für die Tasks bei einem EDF-Scheduling zu:

- $t_{B,1} = 17$
- $t_{B,2} = 11$
- $t_{B,3} = 15$
- $t_{B,4} = 19$

Gegenüber den Blockierzeiten bei statischen Prioritäten haben die Tasks T_3 und T_4 nun sehr hohe maximale Blockierzeiten.

8.3.1.3 PCP

Prioritätengesteuertes Scheduling

Beim PCP muss der Entwickler die Benutzung der Ressourcen im System anmelden. Dadurch kann das System die Priority-Ceiling der Ressourcen berechnen und Tasks blockieren, deren Priorität nicht hoch genug ist. Daher kann auch die maximale Blockierzeit weitaus einfacher berechnet werden. Denn wenn eine Task eine Ressource R benötigt, so kann sie maximal für die Länge eines kritischen Bereiches einer anderen Task blockiert werden, die ebenfalls diese Ressource benötigt. T_i kann also maximal für die Länge des kritischen Bereichs für R von T_j blockiert werden, wenn gilt $T_i < T_j$ und $\pi(R) \geq \pi_i$.

Für die Blockierzeit einer Task i gilt Gleichung 8-15 [Buttazzo1997]. Die Gleichung bildet das Maximum über alle Tabelleneinträge der niederprioren Tasks (i<j), berücksichtigt allerdings nur die Ressourcen, die die Task selbst (direktes Blockieren, Task Blocking) oder eine höherpriore Task (indirektes Blockieren, Resource Blocking) verwendet (die Prio-

rität der Ressource muss höher sein als die Priorität der betrachteten Task i).

$$t_{B,i} = \max \{t_{CS,jk} \mid j > i \; und \; \Pi(R_k) \geq \pi_i\}$$

Gleichung 8-15
PCP-Blockierzeit bei konstanten Prioritäten

Ein mehrmaliges Blockieren während der Ausführung einer Task kann bei PCP gegenüber PIP nicht auftreten. Daher ist auch die Blocking Time geringer.

Betrachten wir wieder das Beispiel aus Abbildung 8-4.

Tasks	R_1	R_2	R_3
T_1	1	2	0
T_2	0	9	3
T_3	8	7	0
T_4	6	5	4
$\prod$(R)	T_1	T_1	T_2

Für die Tasks ergeben sich mit dem PCP-Protokoll nachfolgende Blockierzeiten:

T_1

Alle signifikanten Stellen ($\prod(R_k) \geq \pi_i$) der niederprioren Tasks (T_2 bis T_4) führen zu:

```
t = max (0,9,8,7,6,5) = 9
```

T_2

Nun sind alle Ressourcen zu betrachten, da die Bedingung $\prod(R_k) \geq \pi_i$ nun auch für R_3 gilt:

```
t = max (8,7,0,6,5,4 ) = 8
```

T_3

Entsprechend der obigen Überlegungen ergibt sich für T_3:

```
tB,3= max (6,5,4) = 6
```

T_4

Wie schon bei PIP ergibt sich:

```
t = 0
```

Es lässt sich unschwer erkennen, dass die Blockierzeiten von T_1 und T_2 nach dem PCP-Protokoll deutlich geringer sind als bei Verwendung von PIP.

EDF-Scheduling

Beim Earliest Deadline First muss eine Task damit rechnen, von jeder anderen Task unterbrochen zu werden. Damit müssen aber auch alle Ressourcen bei der Berechnung der Blockierzeit betrachtet werden, was einem Worst Case (bei einer Priority Ceiling sämtlicher Ressourcen) von eins entsprechen würde. Man erkennt bereits hier, dass die Blockierzeiten bei PCP höher sind als bei einem prioritätengesteuerten Scheduling.

Gleichung 8-16
PCP-Blockierzeit beim EDF

$$t_{B,i} = \max\{t_{CS,jk} \mid j \neq i\}$$

Die Blockierzeit für eine Task T_i aus einem Taskset ist also der längste kritische Bereich aus der Menge der Tasks ohne T_i.

Damit ergibt sich aus dem vorherigen Beispiel:

Tasks	R_1	R_2	R_3
T_1	1	2	0
T_2	0	9	3
T_3	8	7	0
T_4	6	5	4

T_1

$t_{B,1}$= max (0,9,3,8,7,0,6,5,4) = 9

T_2

$t_{B,2}$= max (1,2,0,8,7,0,6,5,4) = 8

T_3

$t_{B,3}$= max (1,2,0,0,9,3,6,5,4) = 9

T_4

$t_{B,4}$= max (1,2,0,0,9,3,8,7,0) = 9

8.3.2 Schedulingtest

Die maximalen Blockierzeiten jeder Task haben wir entsprechend den verwendeten Protokollen bestimmt. Um nun mithilfe der bereits bekannten Schedulingtests zu überprüfen, ob die vier Tasks aus vorherigem Beispiel alle ihre maximalen Deadlines einhalten, benötigen wir noch die temporalen Parameter der Tasks:

Tasks	$t_{Pmin,i}$	$t_{Emax,i}$	$t_{Dmax,i}$
T_1	30	3	30
T_2	40	12	40
T_3	70	15	70
T_4	100	15	100

Prioritätengesteuertes Scheduling

Der hinreichende Schedulingtest für Tasks mit statischen Prioritäten (Gleichung 8-1), erweitert um die Blockierzeit, ergibt Gleichung 8-17.

$$u_i = \frac{t_{B,i}}{\min\left(t_{Dmax,i},\, t_{Pmin,i}\right)} + \sum_{j=1}^{n} \frac{t_{Emax,j}}{\min\left(t_{Dmax,j},\, t_{Pmin,j}\right)} \leq n(2^{\frac{1}{n}} - 1)$$

Gleichung 8-17 *Hinreichender Schedulingtest mit Blockierzeit und statischen Prioritäten*

Die Blockierzeiten bei PCP sind:

Tasks	t_B
T_1	9
T_2	8
T_3	6
T_4	0

Damit ergeben sich bei PCP für die vier Tasks folgende Reaktionszeiten:

T_1

$9/30 + 3/30 = 0{,}4 \leq 1$

T_2

$8/40 + (3/30 + 12/40) = 0{,}6 \leq 0{,}828$

T_3

$6/70 + (3/30 + 12/40 + 15/70) = 0{,}7 \leq 0{,}78$

T_4

$0 + (3/30 + 12/40 + 15/70 + 15/100) = 0{,}7642 \not\leq 0{,}757$

Für T_4 müssen wir mit der TDA-Analyse prüfen, ob die Task ihre Deadline einhält.

Wird die Time-Demand-Analyse (TDA) um die Blocking Time erweitert, ergibt sich Gleichung 8-18.

Gleichung 8-18 *Notwendiger Schedulingtest mit Blockierzeit und statischen Prioritäten*

$$t_{C,i}(t) = \lceil \frac{t}{t_{Pmin,i}} \rceil * t_{B,i} + \sum_{j \in J} \lceil \frac{t}{t_{Pmin,j}} \rceil * t_{Emax,j}$$

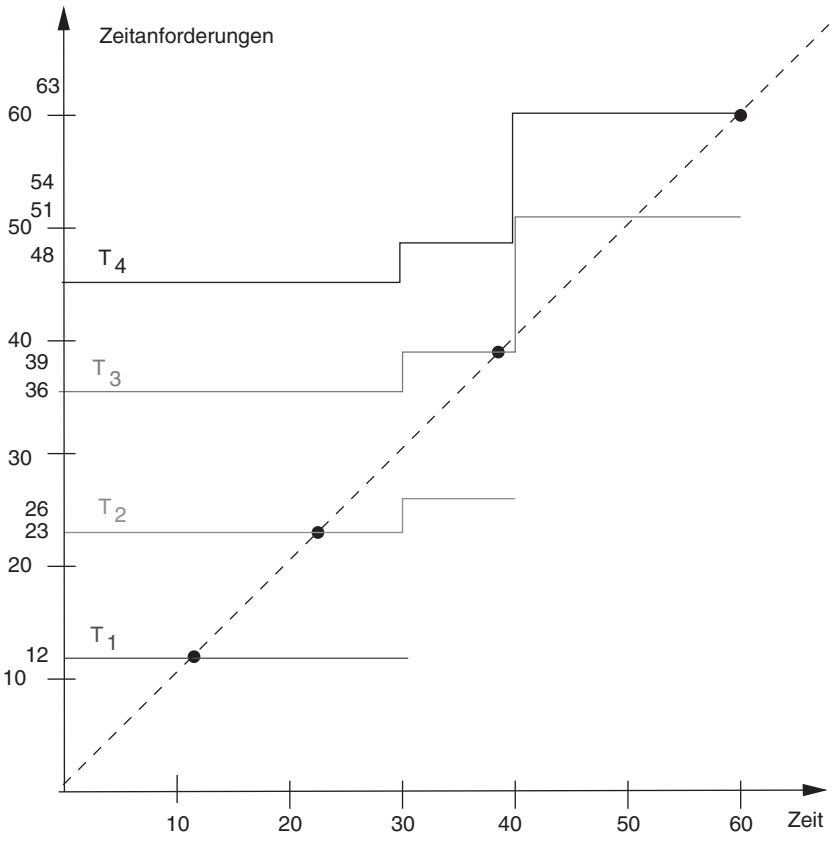
Die iterative Lösung von $t_C(t) = t$ für T_4 mit $t^0 = t_{Emax,4} = 15$ ergibt:

$t^1 = 15 + 0 + \lceil 15/30 \rceil * 3 + \lceil 15/40 \rceil * 12 + \lceil 15/70 \rceil * 15 = 45$
$t^2 = 15 + 0 + \lceil 45/30 \rceil * 3 + \lceil 45/40 \rceil * 12 + \lceil 45/70 \rceil * 15 = 60$
$t^3 = 15 + 0 + \lceil 60/30 \rceil * 3 + \lceil 60/40 \rceil * 12 + \lceil 60/70 \rceil * 15 = 60$

Damit ergibt sich eine maximale Reaktionszeit für T_4 von 60 ms. Dies ist niedriger als die max. Deadline von 100 ms, somit kann schritthaltende Verarbeitung garantiert werden.

Für eine grafische Lösung der TDA kann man die Rechenzeitanforderungen nicht – wie bisher – als Summe auftragen, sondern muss jede Task einzeln nach Abbildung 8-5 konstruieren:

Abbildung 8-5 *Zeitanforderungen (inklusive Blockierzeiten)*



EDF-Scheduling

Für ein System ohne konstante Prioritäten gilt nun Gleichung 8-19.

$$u_i = \frac{t_{B,i}}{\min\left(t_{Dmax,i},\, t_{Pmin,i}\right)} + \sum_{j=1}^{n} \frac{t_{Emax,j}}{\min\left(t_{Dmax,j},\, t_{Pmin,j}\right)} \leq 1$$

Gleichung 8-19 *Hinreichender Schedulingtest mit Blockierzeit (EDF)*

Die vier Tasks aus dem vorherigen Beispiel können als periodisches Taskset auch folgendermaßen beschrieben werden: $T_1(30;3)$, $T_2(40;12)$, $T_3(70;15)$ und $T_4(100;15)$. Es handelt sich hierbei um eine notwendige und hinreichende Bedingung, da $t_{Pmin,i} = t_{Dmax,i}$ ist. Mit den zuvor berechneten Blockierzeiten bei Einsatz eines PCP im Fall des EDF

Tasks	t_B
T_1	9
T_2	8
T_3	6
T_4	0

ergeben sich folgende Reaktionszeiten:

T_1

9/30 + (3/30 + 12/40 + 15/70 + 15/100) = 1,06 ≰ 1 (Bedingung verletzt!)

T_2

8/40 + (3/30 + 12/40 + 15/70 + 15/100) = 0,96 ≤ 1

T_3

9/70 + (3/30 + 12/40 + 15/70 + 15/100) = 0,89 ≤ 1

T_4

9/100 + (3/30 + 12/40 + 15/70 + 15/100) = 0,85 ≤ 1

Bei Verwendung eines EDF-Schedulers und Einsatz des PCP können mit dem Taskset des Fahrradcomputers die Realzeitanforderungen nicht eingehalten werden.

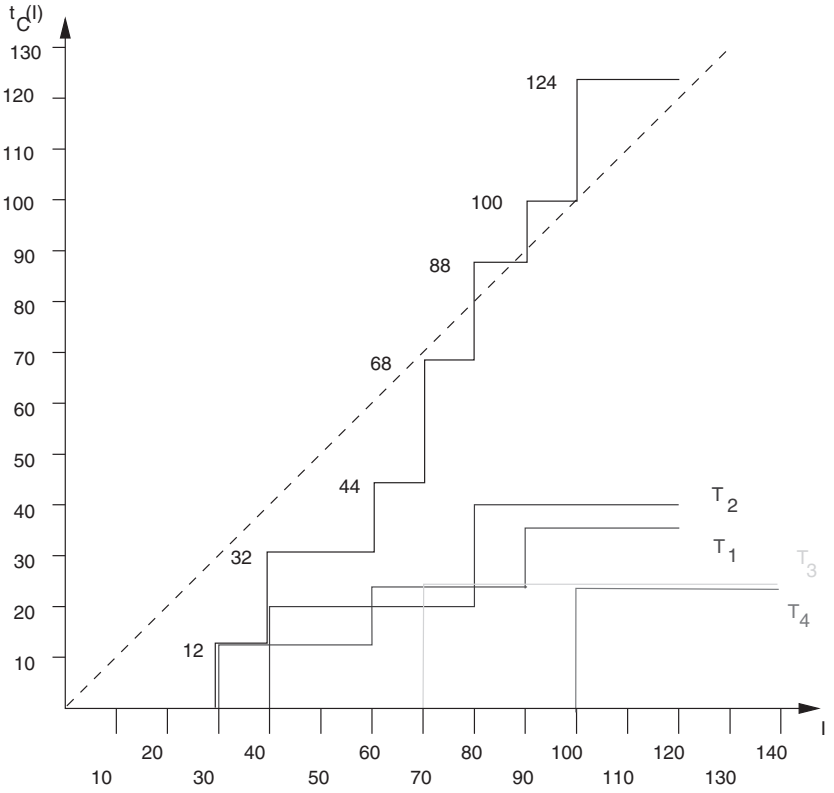
Unabhängig davon zeigt Gleichung 8-20 die Berechnungsgrundlage für den mathematischen Nachweis der Einhaltung von Realzeitanforderungen im Fall eines EDF-Schedulings unter Berücksichtigung von Blockierzeiten.

Gleichung 8-20 *Notwendiger Schedulingtest mit Blockierzeiten (EDF)*

$$t_{C,ges}(I) = \sum_{j=1}^{n} \lfloor \frac{I + t_{Pmin,j} - t_{Dmax,j} - t_{Ph,j}}{t_{Pmin,j}} \rfloor \cdot (t_{Emax,j} + t_{B,j}) \leq I$$

In der Anwendung auf das Beispiel des Fahrradcomputers ergibt sich grafisch betrachtet Abbildung 8-6. Wie erwartet ist die Überlast deutlich zu erkennen.

Abbildung 8-6 *Ereignisstrom-modell: 4-Task-Set*



Abschließende Ergebnisse

Der Realzeitnachweis ohne Berücksichtigung der Blockierzeiten hat gezeigt, dass in einem System mit EDF ein Taskset eher rechtzeitig ausgeführt werden kann, während Systeme mit konstanten Prioritäten bei entsprechend hoher Auslastung dies nicht mehr garantieren können. Bei EDF handelt es sich um ein optimales Scheduling-Verfahren für Einprozessorsysteme.

Wird die Blockierzeit jedoch in den Realzeitnachweis mit einbezogen, wendet sich das Blatt. Denn in Systemen ohne konstante Prioritäten treten Blockierzeiten naturgemäß für jede Task auf. Somit entstehen für einige Tasks höhere Blockierzeiten als in Systemen mit konstanten

Prioritäten. Dies muss beim Realzeitnachweis unbedingt beachtet werden.

Natürlich hängt die Blockierzeit selbst vom gewählten Protokoll ab, wobei hier PCP und SPCP generell kürzere Blockierzeiten als PIP garantieren. Ähnlich gute Blockierzeiten bieten Systeme mit Unterbrechungssperren (NPCS). In unserem Beispiel sind diese genauso hoch wie bei PCP/SPCP, was jedoch am gewählten Beispiel liegt. Da bei NPCS die Unterbrechungssperren global sind, bietet sich diese Art der Implementierung nur für Tasksets an, bei denen möglichst viele Tasks auf eine große Menge der Ressourcen zugreift und die kritischen Bereiche sehr kurz sind.

Task	NPCS (stat. Prioritäten)	NPCS (EDF)	PIP (stat. Prioritäten)$_i$	PIP (EDF)	PCP (stat. Prioritäten)$_i$	PCP (EDF)
T_1	9	9	17	17	9	9
T_2	8	8	14	11	8	8
T_3	6	9	6	15	6	9
T_4	0	9	0	19	0	9

Tabelle 8-4 *Blockierzeiten beispielhaft im Vergleich*

8.4 Bewertung und weitere Einflussfaktoren

In den vorherigen Kapiteln haben wir den Realzeitnachweis in der Anwendungsebene (Userland) kennengelernt. Dabei sind wir von einer idealen Hardware und einem idealen Betriebssystem ausgegangen, welches das Zeitverhalten der Task nicht weitergehend beeinflusst. In der Realität ist das natürlich nicht so.

Das Betriebssystem, also die Funktionalität, die das Betriebssystem in Form von Systemfunktionen zur Verfügung stellt, hat sehr wohl einen entscheidenden Einfluss auf die Reaktionszeiten der Tasks und soll nun genauer betrachtet werden.

Die Reaktion auf Rechenzeitanforderungen beginnt immer zuerst innerhalb des Betriebssystems (ISR), welches über den Scheduler die zugehörige Task aktiviert. Dabei sind in der Regel die folgenden Randbedingungen gegeben:

- Der technische Prozess löst einen Interrupt aus.
- Die benutzerspezifische Reaktion ist in einer Userland-Task formuliert.
- Bei der Aktivierung der Interrupt-Service-Routine besitzt eine Userland-Task den Rechnerkern, die nicht die Reaktion auf dieses Ereignis enthält.

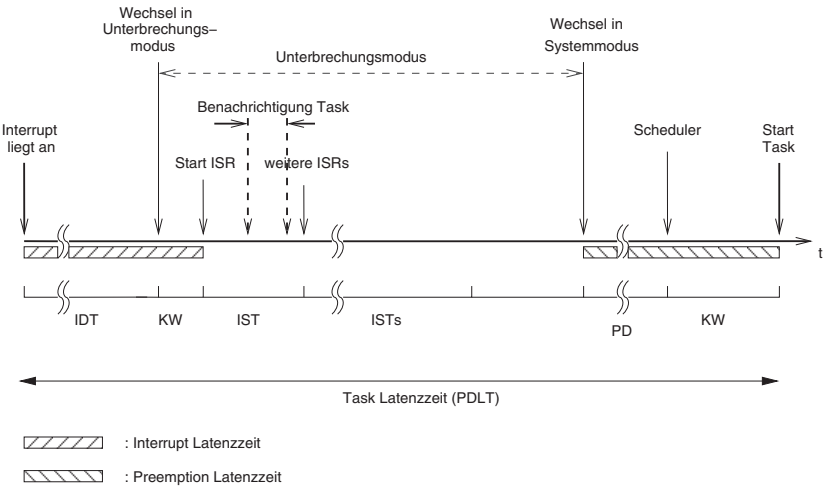
In vielen Systemen ist die Interrupt-Service-Routine zur Realisierung der Primärreaktion Bestandteil des Gerätetreibers. Zur Einleitung der Se-

kundärreaktion (Aktivieren der Task) kann das Betriebssystem spezielle Funktionen anbieten. In Unix- beziehungsweise POSIX- Systemen kann die Sekundärreaktion durch ein Unix- bzw. POSIX-Signal ausgelöst werden. Sie muss dann in der Form einer Signalreaktion in einer geeigneten Task beschrieben werden. Standard-Unix-Signale sind hierfür ungeeignet, da sie verloren gehen können und keine Zuordnung zu Prioritäten kennen. Die Realzeiterweiterungen zu POSIX enthalten daher auch Erweiterungen des Signalkonzeptes.

Eine andere effiziente Möglichkeit zur Einleitung der Sekundärreaktion ist die Benutzung eines Semaphors oder einer Condition-Variablen. Eine Task wird vor Ausführen der Sekundärreaktion durch ein gesperrtes Semaphor angehalten. In der Primärreaktion wird dieses Semaphor freigegeben und damit die Sekundärreaktion ausgeführt.

Während der Ausführung der Primär- und der Sekundärreaktion kann es aus unterschiedlichen Gründen zu Verzögerungen kommen, die zu einer Verlängerung der Reaktionszeit führen. Unter anderem sind dies die in Abbildung 8-7 dargestellten Latenzzeiten, die für einen Realzeitnachweis nicht vernachlässigt werden dürfen.

Abbildung 8-7
Latenzzeiten bis zum Taskstart



Tritt ein Ereignis auf, so kann je nach Implementierung eine Interruptsperre (Interrupt Disable Time: IDT) die Ausführung der Primärreaktion verzögern. Erst nach einem Kontextwechsel (KW) vom User-Modus oder Kernel-Modus in den Unterbrechungsmodus kann die ISR ausgeführt werden. Die Dauer der ISR ist die Interrupt Service Time (IST). Die ISR benachrichtigt auch die User Task (Sekundärreaktion).

Falls während oder nach der ISR weitere ISRs anderer Rechenzeitanforderungen auftreten, so verlängert die Ausführung dieser ISRs das Zurückschalten vom Unterbrechungsmodus in den Systemmodus (ISTs).

Hinzu kommen noch sogenannte Preemption Delays (PD), die per Lockmechanismus geschützte, kritische Abschnitte im Betriebssystemkern darstellen. Um im Benutzermodus die Task auszuführen, ist ebenfalls ein Kontextwechsel (KW) nötig.

$$t_{PDLT} = t_{IDT,max} + \sum t_{IST} + t_{PD,max} + 2 * t_{KW}$$

Gleichung 8-21
Berechnung der Task-Latenzzeit (PDLT)

Gleichung 8-21 zeigt eine Annäherung an die Process Dispatch Latency Time (PDLT). Es fehlt die Betrachtung der Auftrittsfrequenz der Interrupts, als Annäherung gehen wir davon aus, dass für den Worst Case alle Interrupts während der Task Latenzzeit maximal einmal auftreten können.

Typisch für diese Arten von Latenzzeiten sind das nicht deterministische Auftreten sowie die Schwierigkeit, Maximalwerte angeben zu können. Der Code des Betriebssystemkerns steht für eine Analyse oft nicht zur Verfügung. Diese Latenzzeiten treten unerwartet auf, da der interne Systemzustand des Betriebssystems beim Auftreten einer Rechenzeitanforderung (zum Beispiel per Interrupt) sowie während der Ausführung einer Task nicht bekannt ist.

Aufgrund des zufälligen Charakters der Latenzzeiten ist es aber für ein zeitlich deterministisches System wichtig, die Maximalwerte zu kennen, da sonst kein umfassender Realzeitnachweis möglich ist. Im Betrieb machen sich diese Verzögerungen eher als Ausreißer bemerkbar. Mit gewöhnlichen Messmethoden (zum Beispiel Benchmarks) werden Latenzzeiten nicht sichtbar, Maximalwerte können also nicht nach herkömmlicher Art bestimmt werden.

Darum wird in der Praxis häufig mithilfe von Laufzeitmessungen eine Abschätzung vorgenommen, wie leistungsfähig ein Rechensystem sein muss, um die geforderten Zeitbedingungen zu erfüllen. Zur Erlangung einer ausreichenden Sicherheit wird das System um den sogenannten Sicherheitsfaktor gegenüber dem geschätzten Wert überdimensioniert. Die Hoffnung ist, dass alle eventuell auftretenden Latenzzeiten durch diese Sicherheit abgefangen werden können. Es bleibt die Unsicherheit, ob der gewählte Sicherheitsfaktor wirklich ausreichend groß genug gewählt wurde.

Eine Ursache für die auftretenden Latenzzeiten ist der Schutz kritischer Bereiche. Bei der Betrachtung der Protokolle haben wir mit dem NPCS-Protokoll bereits einen Vertreter für Unterbrechungssperren bzw. Verdrängungssperren kennengelernt. Hersteller können die maximalen Latenzzeiten, die innerhalb der PDLT auftreten können, angeben.

Doch es kann auch nach der PDLT zu weiteren Verzögerungen durch das Betriebssystem kommen. Diese treten eher selten auf, sind dann aber sehr viel länger. Daher ist auch der Nachweis schwierig.

Die Reaktionszeit beinhaltet die eigentliche Reaktion selbst und somit auch eventuellen Betriebssystemcode, der als Teil der Taskreaktion ausgeführt wird. Beim Zugriff auf gemeinsame Daten, welche wir in den vorherigen Abschnitten auf User-Ebene betrachtet haben, treten Blockierzeiten auch bei der Ausführung des Betriebssystemcodes auf. Denn die Systemfunktionen, die sich gegenseitig unterbrechen können, greifen schreibend und lesend auf interne, globale Daten des Betriebssystems zu. Wichtig bei der Realisierung der Systemfunktionen in einem Realzeitbetriebssystem sind somit die Ziele:

- Unterbrechbarkeit, um kurze Reaktionszeiten zu gewährleisten,
- Wahrung der Datenkonsistenz.

Die einfachste Möglichkeit, die Datenkonsistenz zu garantieren, ist eine globale Unterbrechungssperre. Diese wird immer dann gesetzt, wenn ein Systemcall aufgerufen wird. In modernen Systemen wird diese Art des Schutzmechanismus allerdings nicht mehr eingesetzt, da sie zu langen Verzögerungen, insbesondere auch im Kontext von Multicore-Architekturen, führt.

Daneben existieren andere Ansätze, von denen im Folgenden drei genauer betrachtet werden sollen:

1. Unterbrechungssperre
2. Unterbrechungspunkte (Preemption Points)
3. Kritische Bereiche

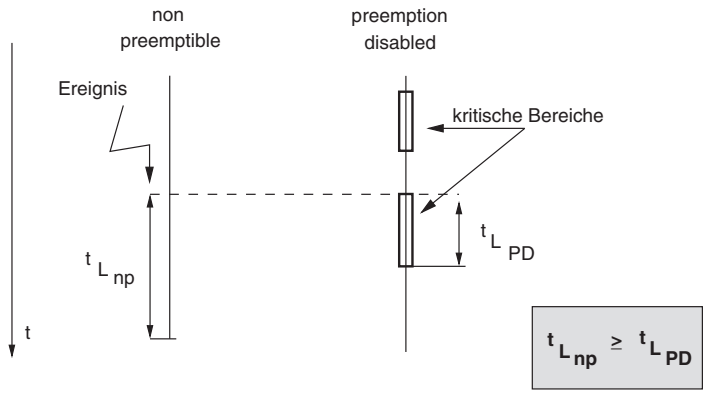


Abbildung 8-8
Latenzzeiten durch Unterbrechungssperre

Anstatt den kompletten Systemcall zu schützen, werden bei der Unterbrechungssperre (NPCS) nur die kritischen Abschnitte geschützt. In Abbildung 8-8 ist die dadurch entstehende Latenzzeit rechts dargestellt. Sie

tritt immer vor der Sekundärreaktion auf. Links ist im Vergleich dazu die komplette Unterbrechungssperre dargestellt. Um die maximale Länge der Latenzzeit für diesen Fall zu berechnen, ist die Kenntnis der Länge der kritischen Bereiche der Systemfunktionen notwendig. Vor einigen Jahren war diese Methode in Betriebssystemen noch sehr verbreitet anzutreffen, da es sich um einen sehr einfachen Schutz handelt. Mit dem Einsatz von Multicore-Systemen und dem Ziel, die Unterbrechbarkeit in einem System zu maximieren, findet man diesen Schutz in modernen Realzeitsystemen nur noch im Ausnahmefall für extrem kurze kritische Bereiche.

Unterbrechungspunkte (Wiedereintrittspunkte) können eingerichtet werden, wenn alle systeminternen Daten beim Ausführen einer Systemfunktion an bestimmten Stellen in einem konsistenten Zustand sind. Die Unterbrechung an dieser Stelle und ein erneuter Aufruf einer Systemfunktion sind daher möglich.

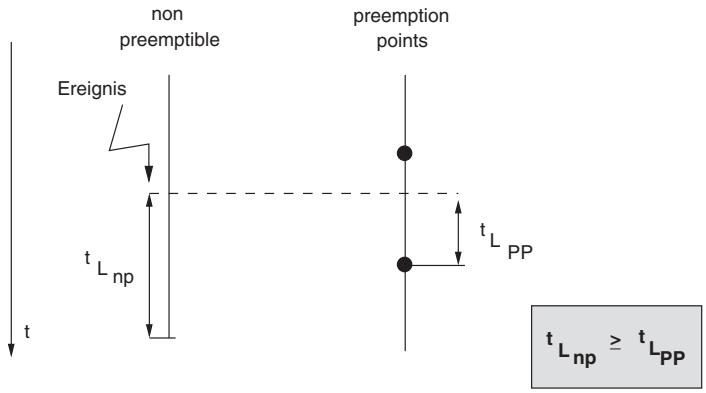


Abbildung 8-9
Latenzzeiten durch Unterbrechungspunkte

In Abbildung 8-9 sind eine unterbrechbare Systemfunktion (links) und eine Systemfunktion mit Unterbrechungspunkten (rechts) dargestellt. Während der Ausführung der Systemfunktion ist das Auftreten eines Ereignisses dargestellt, auf welches so schnell wie möglich reagiert werden soll (höchste Priorität). Die Reaktion auf das Ereignis (zum Beispiel Taskwechsel) kann bei dem nicht unterbrechbaren Systemdienst erst nach dessen Beenden geschehen ($t_{L,np}$), wenn in den User-Modus zurück gewechselt wird.

Bei der rechts dargestellten Systemfunktion kann bei Erreichen eines Unterbrechungspunktes bereits die Reaktion auf das Ereignis eingeleitet werden. Die Systemdaten befinden sich an diesem Punkt wieder in einem konsistenten Zustand. Das Ausführen der gleichen oder einer anderen Systemfunktion und die damit verbundene mögliche Manipulation

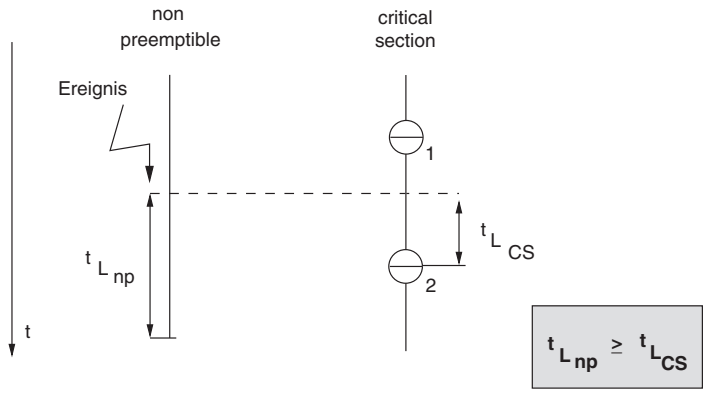
weiterer oder gleicher Objekte kann nicht zu einem inkonsistenten Datenzustand führen.

Der Maximalwert der hierdurch bedingten Latenzzeit $t_{L,PP}$ ist durch den maximalen zeitlichen Abstand zwischen aufeinanderfolgenden Unterbrechungspunkten bestimmt.

Der dritte Ansatz, die Bildung von kritischen Abschnitten, entspricht der Standardlösung des Konsistenzproblems. Da hier insbesondere der zeitliche Ablauf interessiert, wird das Konzept ausführlich dargestellt. Zur Realisierung sind folgende Bedingungen zu erfüllen:

1. Eine Folge von Anweisungen, die Daten von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt (kritischer Abschnitt), darf so lange nicht erneut aufgerufen werden, bis die letzte Anweisung für den vorhergehenden Aufruf ausgeführt wurde.
2. Wenn Daten, die in je zwei kritischen Abschnitten benutzt und in mindestens einem davon verändert werden, einen nicht leeren Durchschnitt bilden, so stehen diese zueinander in Konflikt und dürfen nur unter wechselseitigem Ausschluss ausgeführt werden.

Abbildung 8-10
Latenzzeiten durch kritische Bereiche



Gegenüber einem Systemdienst mit kompletter Unterbrechungssperre ist in Abbildung 8-10 rechts ein Systemdienst mit zwei Bereichen dargestellt, in dem die Unterbrechung des Dienstes gesperrt ist. Das Semaphor wird gesetzt (belegt), während der kritische Bereich in Ausführung ist. Trifft nun ein Ereignis während der Ausführung eines kritischen Bereiches ein, so kommt es zu der Verzögerung $t_{L,CS}$.

Wir gehen davon aus, dass die Systemfunktionen aus kritischen Abschnitten konstruiert sind und dass das Betreten eines Abschnitts durch mehrere Tasks mit einem Semaphor verhindert wird. Überholungsvorgänge sind zwischen diesen Abschnitten möglich. Die Systemfunktion hat beispielsweise, wie in Abbildung 8-10 dargestellt, zwei kritische Bereiche (1,2). Bei Betreten eines kritischen Bereiches wird das Semaphor

belegt. Da die Daten, auf welche während des kritischen Bereichs zugegriffen wird, vor einem erneuten Zugriff durch das Semaphor geschützt sind, lässt sich der Systemdienst jederzeit unterbrechen.

Zum Zeitpunkt t, also wenn der Interrupt eintrifft, kann der Systemdienst sofort unterbrochen werden, da ein erneuter Zugriff durch das gesetzte Semaphor zwei verhindert wird. Eine Verzögerung bei der Bearbeitung einer Rechenzeitanforderung tritt nur dann auf, wenn die Bearbeitungsfunktion auf ein durch ein Semaphor gesperrtes Betriebsmittel warten muss, was erst während der Ausführung einer Sekundärreaktion passiert.

Auf den ersten Blick sieht die Lösung mit Semaphoren gut aus, da eine Task jederzeit unterbrochen werden kann, wenn eine Rechenzeitanforderung auftritt. Der Schutz kritischer Bereiche durch Semaphore kann bei der Ausführung des unterbrochenen Systemdienstes zu Verzögerungen führen. Dieser Fall ist in Abbildung 8-11 dargestellt.

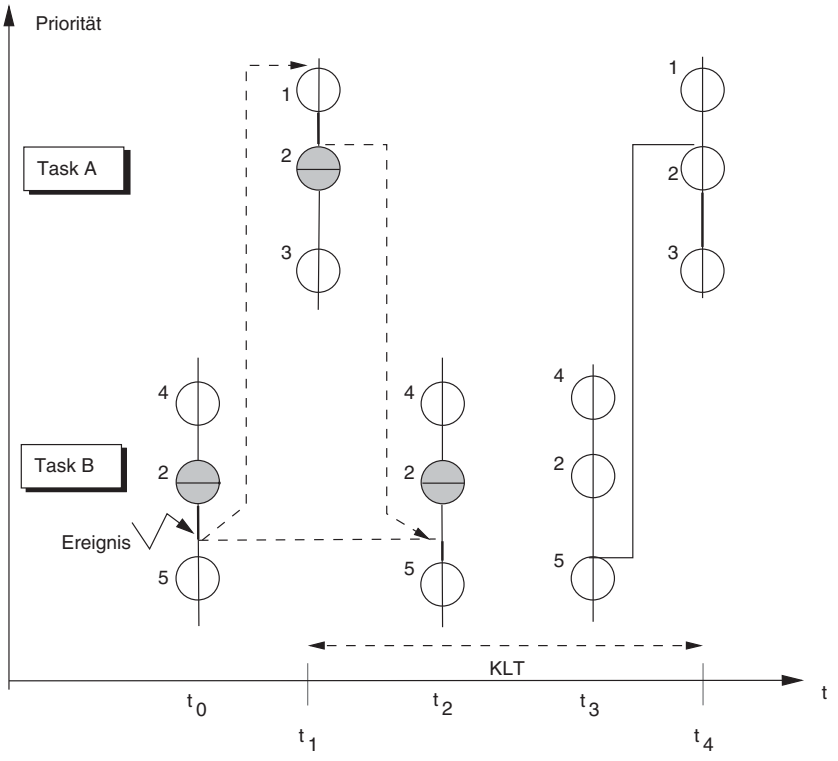


Abbildung 8-11
Kernel Latency Time (KLT)

Diese Art von Verzögerungszeit, die dabei auftritt, wird auch als Kernel-Latenzzeit (KLT) bezeichnet. Die Systemdienste bestehen in diesem Beispiel unter anderem aus drei geschützten Bereichen. Jeder dieser drei Bereiche ist durch ein eigenes Semaphor geschützt (1-2-3 und 4-2-5). Die

beiden dargestellten Systemfunktionen greifen auf gemeinsame Daten zu, daher haben beide Systemfunktionen einen gemeinsamen kritischen Bereich (2).

Task A hat eine höhere Priorität als Task B. Tritt während der Ausführung eines kritischen Bereiches (Task B) ein Ereignis auf (t_0), sind der Bereich und somit das Objekt, auf welches in diesem Bereich zugegriffen wird, geschützt. Ein möglicher Kontextwechsel zur rechenbereiten Task A, die in diesem Beispiel die Reaktion auf das aufgetretene Ereignis darstellt, ist sofort möglich. Zum Zeitpunkt t_1 ist Task A aktiv. Wird nun jedoch der gleiche Systemdienst oder ein Systemdienst benutzt, der auf das bereits gesperrte Objekt zugreift, kommt es zur Kernel-Latenzzeit. Diese Verzögerungszeit besteht aus einem Kontextwechsel zu Task B (t_2 - t_1), dem Beenden des kritischen Bereiches (t_3 - t_2), welcher geschützt ist, und einem Kontextwechsel zurück zu Task A (t_4 - t_3). Damit lässt sich die KLT nach Gleichung 8-22 berechnen.

Gleichung 8-22
Berechnung der Kernel Latency Time

$$t_{KLT} = 2 * t_{KW} + t_{CS}$$

In Gleichung 8-22 steht t_{KW} für die Dauer eines Kontextwechsels und $t_{CS,max}$ für die Ausführungsdauer des längsten kritischen Bereiches der Systemfunktion. Die Ursache der Verzögerungen bei der Ausführung einer Task und somit auch bei der Ausführung eines Systemdienstes ist der Einfluss von anderen Tasks, die in einem Multitasking-Betriebssystem aktiv sind.

Bei der Berechnung der KLT oben ist die Zeit für die Ereignisbehandlung und Latenzzeiten bis zum Taskstart nicht enthalten. Für den Worst Case der Verzögerung einer Task gilt Gleichung 8-23.

Gleichung 8-23
Berechnung der maximalen Verzögerung einer Task

$$t_{TDelay} = t_{PDLT,max} + t_{KLT,max}$$

Die KLT tritt beim Ausführen der Sekundärreaktion auf, wird also bei Angaben der Hersteller von Realzeitbetriebssystemen zur Interrupt-Latenzzeit und Task-Latenzzeit (PDLT) nicht beachtet. Im Gegenteil: Durch den Schutz kritischer Bereiche durch z.B. Semaphore können die Hersteller sehr gute ILT- und PDLT-Zeiten angeben. Das Problem der Blockierzeit wird auf einen späteren Zeitpunkt verschoben: beim Ausführen der tatsächlichen Reaktion, d.h. beim Ausführen der Task selbst.

Wenn das System aus mehreren Tasks besteht, die mehrere Systemfunktionen verschränkt ausführen, und eine Systemfunktion aus verschiedenen kritischen Bereichen besteht, so können je nach Länge der einzelnen kritischen Bereiche durch die Kaskadierung sehr hohe Latenzzeiten im Worst Case auftreten. Die Wahrscheinlichkeit für die maxima-

le Latenzzeit aufgrund verschachtelter Systemfunktionen ist zwar gering, aber sie besteht. Maximalwerte treten bei Tests als Ausreißer auf und können nur mit großem Aufwand reproduziert werden.

Die Hersteller der Betriebssysteme können dazu beitragen, auch diese Latenzzeiten zu bestimmen, indem Anzahl und Länge der kritischen Bereiche der Systemfunktionen bekannt gegeben würden. Für den Programmierer besteht die Möglichkeit, maximale Latenzzeiten zu bestimmen, wie dies in [Mächtel2000] beschrieben ist, auch ohne Kenntnis des Source-Codes des Betriebssystems. Jedoch ist ein nicht zu unterschätzender Aufwand dafür nötig, weswegen in der Praxis weiterhin mit dem Sicherheitsfaktor bei der Auslastung gearbeitet wird.

Bibliographie

[Abel1990]
Dirk Abel: *Petri-Netze für Ingenieure: Modellbildung und Analyse diskret gesteuerter Systeme*. Springer Verlag, Berlin, 1990.

[Bertogna2009]
M. Bertogna, M. Cirinei, G. Lipari: *Schedulability analysis of global scheduling algorithms on multiprocessor platforms*. IEEE Transactions on Parallel and Distributed Systems, Volume 20, Issue 4, 2009.

[Buttazzo1997]
Giorgio C. Buttazzo: *Hard Real-Time Computing Systems*. Kluwer, Portland, 1997.

[Gresser93]
Klaus Gresser: *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme*. VDI-Verlag GmbH, Düsseldorf, 1993.

[KuQu08]
Eva-Katharina Kunst, Jürgen Quade: *Kern-Technik:* Serie. 5/2008, München, 2008, Linux-Magazin.

[Liu2000]
Jane W.S. Liu: *Real-Time Systems*. Prentice-Hall, Inc, New Jersey, 2000.

[LoMa11]
Paul Lokuciejewski, Peter Marwedel: *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer Verlag, Berlin, 2011.

[Lopez2004]
J.M. Lopez, J.L. Díaz, D.F. García: *Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling*.

IEEE Transactions on parallel and distributed systems, Vol.15, No.7., 2004.

[Mächtel2000]
Michael Mächtel: *Entstehung von Latenzzeiten in Betriebssystemen und Methoden zur meßtechnischen Erfassung*. VDI-Verlag GmbH, Düsseldorf, 2000.

[POSIX 1003.1]
The IEEE and The Open Group: *IEEE Std 1003.1-2008*. 2008. [http://pubs.opengroup.org/onlinepubs/9699919799/]

[QuKu2011]
Jürgen Quade, Eva-Katharina Kunst: *Linux-Treiber entwickeln:* Eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung. 3. Auflage, Heidelberg, 2011, dpunkt.verlag GmbH.

[QuKu2011-59]
Jürgen Quade, Eva-Katharina Kunst: *Kern-Technik 59:* Serie. 11/2011, München, 2011, Linux-Magazin.

[QuKu2012-62]
Jürgen Quade, Eva-Katharina Kunst: *Kern-Technik 62:* Serie. 5/2012, München, 2012, Linux-Magazin.

[QuKu2012-63]
Jürgen Quade, Eva-Katharina Kunst: *Kern-Technik 63:* Serie. 7/2012, München, 2012, Linux-Magazin.

[Rupp2012]
Chris Rupp, Stefan Queins, die SOPHISTen: *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. 4. Auflage, Carl Hanser Verlag, 2012.

[Winne2009]
Olaf C. Winne: *Leitfaden für die Norm IEC 61508*. Elektronik Praxis, 2009.

[Yod99]
Victor Yodaiken: *The RTLinux Manifesto*. Department of Computer Science New Mexico Institute of Technology, 1999.

Stichwortverzeichnis

C

D

E

F

L

M

N

O

P

Q

R

S

T

U

3., aktualisierte und erweiterte Auflage
2011, 598 Seiten, gebunden
€ 49,90 (D)
ISBN 978-3-89864-696-3

Jürgen Quade · Eva-Katharina Kunst

Linux-Treiber entwickeln

Eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung

3., aktualisierte und erweiterte Auflage

Dieses Buch ermöglicht einen fundierten Einstieg in den Linux-Kernel mit einem Schwerpunkt auf der Entwicklung von Gerätetreibern. Für den ersten eigenen Kernelcode bietet es theoretische Grundlagen und wiederverwertbare Softwarebausteine. Dank der detaillierten Beschreibung von über 600 internen Systemfunktionen eignet sich das Buch als Nachschlagewerk.

Die 3. Auflage wurde durchgehend auf den Kernel 2.6.37 aktualisiert. Das betrifft besonders die seit der 2. Auflage hinzugekommenen Umbauten im Linux-Kernel. Neue Themen: Echtzeit-Eigenschaften, Stromsparmodi (»Green Computing«) u.v.m.

»... kein Buch für den Nachttisch ... Wer sich aber ernsthaft mit der Integration neuer Geräte in den Kernel auseinandersetzen muss, kommt an diesem Werk nicht vorbei.«

c't 2011, Heft 19

dpunkt.verlag

Ringstraße 19 B · 69115 Heidelberg
fon 0 62 21/14 83 40
fax 0 62 21/14 83 99
e-mail hallo@dpunkt.de
http://www.dpunkt.de