Ein Beitrag zum Einsatz von echtzeitfähigen Linux-Varianten in der Automatisierungstechnik

Von der Fakultät Maschinenbau
der Universität Dortmund
genehmigte Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften

von
Dipl.-Ing. Dirk Düding
Dortmund

Tag der mündlichen Prüfung: 11. Juli 2003
Referent: Univ.-Prof. Dr.-Ing. Mathias Uhle
Korreferent: Univ.-Prof. Dr.-Ing. Klaus Heinz

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachgebiet Messtechnik der Universität Dortmund.

Mein erster Dank gilt Herrn Prof. Dr.-Ing. Mathias Uhle für die mit seinen Ratschlägen mögliche selbsständige Gestaltung der wissenschaftlichen Arbeit. Viele hilfreiche Anregungen, Ideen und Vorschläge trugen maßgeblich zum Gelingen der Arbeit bei.

Herrn Prof. Dr.-Ing. Klaus Heinz danke ich für das Interesse an der Arbeit und die Übernahme des Korreferats.

Allen Mitarbeitern des Fachgebietes Messtechnik sowie allen von mir betreuten studentischen Hilfskräften danke ich für ihre Leistungen, die mit zum Gelingen der Arbeit beitrugen.

Mein besonderer Dank gilt meinen Eltern, die mir mein Studium ermöglicht haben. Leider durften sie das Ergebnis nicht mehr miterleben.

Meinen Eltern

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

How tough is Tux ?

Angeregt wurde diese Dissertation durch ein Industrieprojekt des Fachgebietes Messtechnik der Fakultät Maschinenbau an der Universität Dortmund. Ziel dieses Projektes war die Entwicklung von automatisierten Pneumatikprüfständen (vergl. Anhang A). Begonnen hat diese Kooperation Anfang der 80er Jahre. Seitdem wurde der Prototyp des Prüfstandes kontinuierlich weiterentwickelt und verbessert.

Die Prüfstände werden mittels eines handelsüblichen PCs gesteuert. Die Logik dieser Steuerung ist vollständig in den PC verlagert. Dies steht im Gegensatz zum weitverbreiteten Einsatz einer SPS als Steuerungskomponente.

Als Betriebssystem wurde MU-DOS-5.1 [36] gewählt. Für dieses Betriebssystem sprachen zuerst einmal die geringen Anforderungen an die Hardware: Ein einfacher 386er PC mit 2MB-RAM und einer 40MB-Platte war für die gesamte Aufgabe bestens geeignet. Dieses Kriterium erfüllt aber auch das Konkurenzprodukt aus dem Hause Microsoft MS-DOS 3.x - 6.22 [122].

Viel wichtiger ist die Forderung nach Echtzeitfähigkeit, da die Steuerung innerhalb einer garantierten Zeitspanne auf Ereignisse reagieren muss, da ansonsten der Prüfling oder Prüfstand oder im Extremfall sogar Menschen zu Schaden kommen. Dies ist schließlich das KO-Kriterium für MS-DOS. Da bereits zu einem früheren Zeitpunkt am Fachgebiet mehrere Projekte unter dem echtzeitfähigen MU-DOS realisiert worden waren, war die Entscheidung für das Betriebssystem gefallen.

Ende der 90er Jahre befanden sich 15 Prüfstände im Einsatz. Durch Umstrukturierungsmaßnahmen im Betrieb wurde es nötig, die Prüfstände zu vernetzen. Erstens, um auf allen Prüfständen ständig die aktuellen Parametersätze für die

Prüfungen vorzuhalten, zweitens, um die Prüfaufträge aus dem eingesetzten PPS-System A+F TEAM 4.0 [2] zu übernehmen und drittens, um die Prüfergebnisse in diesem System zu archivieren.

Hierbei stellte sich sehr schnell heraus, dass das eingesetzte MU-DOS nicht die benötigte Netzwerkfunktionalität bereitstellt. Es ist zwar eine Anbindung auf TCP/IP-Basis mittels FTP möglich, aber weitergehende Funktionalität müsste komplett in Eigenregie entwickelt werden [44]. Eine aktuelle Nachfolgeversion des Betriebssystems zu kaufen, erwies sich als sehr problematisch, da diese inzwischen von einer Firma (MUDOS-GOLD) in den Vereinigten Staaten von Amerika entwickelt und vertrieben wird. Ein Kauf dieser Version scheiterte.

Weiterhin ist eine Fernwartbarkeit der Prüfstände sehr wünschenswert. Häufig wurde ein Techniker angefordert und es stellte sich heraus, dass die vermeintliche Hardware-Fehlermeldung ein ganz normaler Betriebszustand war, z.B. ein richtig erkannter defekter Regler. Im Zusammenhang mit der Fernwartung ergibt sich dann auch, neben der notwendigen Netzwerkfunktionalität, die Frage nach der Sicherheit des zugrundeliegenden Betriebsystems.

Alle hier aufgeführten Punkte erzwangen die Suche nach einem neuen Betriebssystem.

1.2 Zielsetzung

Lange Zeit teilten sich über 70 Real Time Systeme den Markt [31]. Zum einen handelt es sich oft um spezielle Systeme, die nur auf spezieller Hardware laufen und zum anderen um solche, die nur hochspezialisierte Funktionen erfüllen. Im Laufe der Zeit haben sich die Anforderungen geändert. Auf der einen Seite steht die Leistungsentwicklung der Hardware, auf der anderen Seite sollen Echtzeit-Systeme heute viel flexibler sein als früher.

Hierdurch ist der Markt der echtzeitfähigen Betriebssysteme stark in Bewegung geraten und die "Nischenprodukte" werden durch echtzeitfähige Standardbetriebssysteme, hier ist besonders Linux zu nennen, verdrängt. Diese Ausarbeitung hat deshalb das Ziel, zuerst Kriterien, die für den Einsatz eines Betriebssystems in der Automatisierungstechnik wichtig sind, herauszuarbeiten und diese anschließend den Möglichkeiten der angebotenen echtzeitfähigen Linux-Varianten gegenüberzustellen. Hierzu werden sowohl der unmodifizierte Linux-Kernel als auch speziell für Echtzeitfähigkeit ausgelegte Varianten analysiert.

Beim Einsatz von Linux kommt ein weiterer Gesichtspunkt für die Betrachtungen hinzu: Linux gehört in die Kategorie der *freien Software* und unterliegt speziellen Lizenzen. Den hieraus resultierenden Konsequenzen gilt zusätzliche Aufmerksamkeit. Es wird die Frage untersucht, in wieweit freie Software im Umfeld der Automatisierungstechnik wettbewerbsfähig ist.

1.3 Aufbau der Arbeit

Im folgenden Kapitel 2 werden Prozessbeispiele in der Automatisierungstechnik, bei denen echtzeitfähige Betriebssysteme zur Steuerung eingesetzt werden, dargestellt. Anhand dieser werden später die Anforderungen, die sich von technischer Seite an das Betriebssystem ergeben, formuliert. In diesem Kapitel erfolgt auch eine Definition des Begriffes *Echtzeit*.

Im Kapitel 3 werden die aktuellen Lizenzmodelle, die im Umfeld von Linux von Bedeutung sind, vorgestellt. Da die verschiedenen echtzeitfähigen Linux-Varianten unterschiedlichen Lizenzen unterliegen, wird in der späteren Analyse auch auf die sich hieraus ergebenden Konsequenzen eingegangen. Weiterhin wird ein kurzer Überblick über die Entwicklung von Linux gegeben.

Das Kapitel 4 beschreibt die Aufgaben, die ein Betriebssystem zu erfüllen hat und den Stand der Technik bei der Architektur von Betriebssystemen. Dies beinhaltet Erläuterungen, die sich durch die Echtzeit-Anforderungen ergeben.

Das Kapitel 5 widmet sich den Entwicklungswerkzeugen für die echtzeitfähigen Varianten. Es wird der GNU-Werkzeugkasten vorgestellt, da dieser die Entwicklungsbasis von Linux bildet. Weiterhin wird ein kurzer Überblick über den POSIX-Standard gegeben, der beim Vergleich von Betriebssystemen eine Hilfe ist.

Im Kapitel 6 werden die Anforderungen für den Einsatz eines echtzeitfähigen Betriebssystems in der Automatisierungstechnik herausgearbeitet. Die geschieht übergreifend sowohl aus technischer, als auch aus kaufmännischer Sicht.

Das Kapitel 7 untersucht sowohl den Standard Linux Kernel als auch die echtzeitfähigen Varianten bezüglich ihres Designs und erläutert weitere Details der Varianten, wie z.B. die verwendete Lizenz.

Das Kapitel 8 analysiert die verschiedenen echtzeitfähigen Linux-Varianten anhand der vorher formulierten Anforderungen und führt die sich ergebenden Schlussfolgerungen an. Hierzu zählen sowohl die technischen Aspekte, die sich aus der verwendeten Betriebssystem-Architektur ergeben als auch im besonderen die Konsequenzen, die aus den Lizenzmodellen resultieren. Im Abschnitt 8.3 werden die sich aus der Analyse ergebenden Konsequenzen für eine Beurteilung der Varianten anhand des formulierten Satzes an Kriterien benutzt. Somit ergibt sich für jede Variante ein charakteristisches Profil, welches in Abschnitt 8.4 dazu benutzt wird, für das bereits in der Einleitung erwähnte Prozessbeispiel ein neues Betriebssystem auszuwählen.

Im abschließenden Kapitel 9 werden die wichtigsten Ergebnisse dieser Arbeit noch einmal zusammengefasst und ein Ausblick auf die zukünftige Entwicklung gewagt.

Kapitel 2

Echtzeit in der Automatisierungstechnik

2.1 Motivation

Da der Begriff *Echtzeit* ein Kernthema dieser Arbeit ist, folgt eine genaue Definition und eine Differenzierung in zwei Kategorien der Echtzeitfähigkeit. Weiterhin werden einige Beispiele für Echtzeitsysteme gegeben. Anhand dieser werden die charakteristischen Eigenschaften dieser System herausgearbeitet.

2.2 Echtzeit und deren Klassifizierung

Es gibt viele Interpretationen des Begriffes Echtzeit. In allen taucht aber der Begriff der Antwortzeit oder auch Reaktionszeit auf. Dies ist die Zeit, welche zwischen dem Auftreten eines Ereignisses und der vom System generierten Antwort vergeht.

Im *Oxford Dictionary of Computing* wird die folgende Definition von Echtzeitsystemen gegeben:

> "Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timelines." [67]

Bei dieser Definition ist die Antwortzeit im Kontext des Gesamtsystems zu sehen. In einem Raketensteuerungssystem beträgt sie nur Millisekunden, in einer Montagestraße für Kraftfahrzeuge ist eine Reaktion innerhalb einer Sekunde

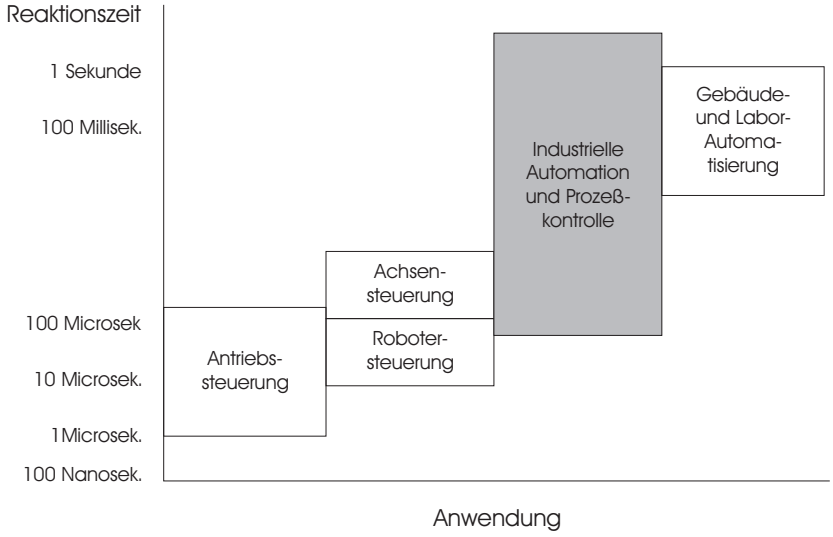


Abbildung 2.1: Reaktionszeiten unterschiedlicher Steuerungs- und Automatisierungsaufgaben[22].

unter Umständen völlig ausreichend. Abbildung 2.1 gibt einen Überblick über die Größenordnung der Reaktionszeit in der Automatisierungstechnik. Grau hinterlegt ist der im Rahmen dieser Arbeit betrachtete Bereich der industriellen Automation.

Durch diese Definition ist ein weitreichendes Feld an Computeranwendungen abgedeckt. So kann zum Beispiel auch Unix als echtzeitfähig betrachtet werden, da ein Benutzer auf einen eingegebenen Befehl in der Regel innerhalb einiger Sekunden eine Antwort erwartet. Erhält er diese nicht innerhalb dieser Zeitspanne, so bedeutet dies auch keinen großen Schaden. Diese Systeme können aber klar abgegrenzt werden von solchen Systemen, bei denen eine verspätete Antwort genau so bewertet werden muss wie eine falsche Antwort. Man denke an ein Regalbediengerät, bei welchem die Steuerung nicht innerhalb kürzester Zeit auf das Überfahren des Endschalters reagiert. Die Funktionsfähigkeit eines Echtzeit-Systems resultiert also nicht nur aus den logisch korrekten Ergebnissen, sondern auch aus der Zeit, in der diese Ergebnisse berechnet werden. In der Praxis wird deshalb in *harte* und *weiche* Echtzeit (*Hard/Soft Realtime*) unterschieden.

2.2.1 Hard Realtime

Ein Betriebssystem wird dann als *hart echtzeitfähig* bezeichnet, wenn alle Zeitschranken (*Deadlines*), bedingt durch Ereignisse der Umwelt, innerhalb einer vorhersagbaren Toleranz eingehalten werden können. Diese Ereignisse können planmäßig oder sporadisch auftreten. Die Eigenschaften des Betriebssystems sind in

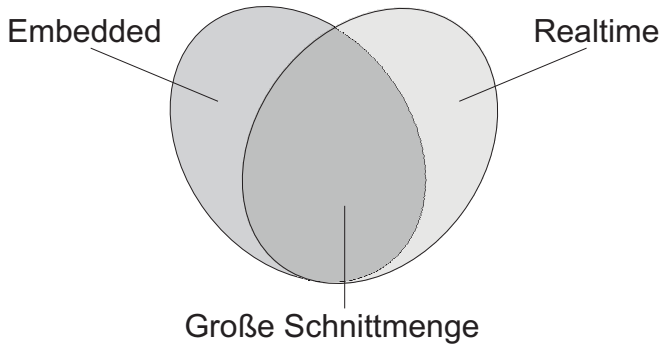


Abbildung 2.2: Allgemeine Einschätzung über Realtime und Embedded-Systeme

solchen Fällen wichtig, in denen das Nicht-Einhalten einer Deadline katastrophale Folgen hat, z.B. zur Zerstörung der Prüfeinrichtung oder zur Gefährdung von Menschen führt.

2.2.2 Soft Realtime

Ein Betriebssystem wird dann als *weich echtzeitfähig* bezeichnet, wenn alle Zeitschranken in einem statistischen Sinne erfüllt werden. Solche Betriebssysteme finden bei Multimedia-Anwendungen ihren Einsatz, in diesen Fällen hat das Verfehlen einer Deadline keine schlimmen Folgen, sondern ist in der Regel nur ein "Schönheitsfehler".

2.2.3 Embedded gleich Realtime?

Im Umfeld von Echtzeitsystemen fällt regelmäßig der Begriff *embedded* und es entsteht der Eindruck, alle Embedded-Systeme repräsentieren zugleich auch Echtzeit-Systeme. Zunächst zur Frage: Was sind Embedded-Systeme?

In *Software design for Realtime Systems* wird ein Embedded-System folgendermaßen charakterisiert:

> "Embedded computers are defined to be those where the computer is used as a component within a system: not as a computing engine in its own right." [28]

Im Allgemeinen gilt die Meinung: Embedded ist gleich Realtime und Realtime ist gleich embedded (Abb. 2.2). Dieser Sachverhalt stellt sich bei genauerer Untersuchung anders da: Nur 10% bis 15% aller Embedded-Systeme haben Echtzeitanforderungen und nur die Hälfte davon stellt harte Echtzeitanforderungen. Von diesen Systemen fällt weiterhin ein großer Teil nicht unter die Kategorie der

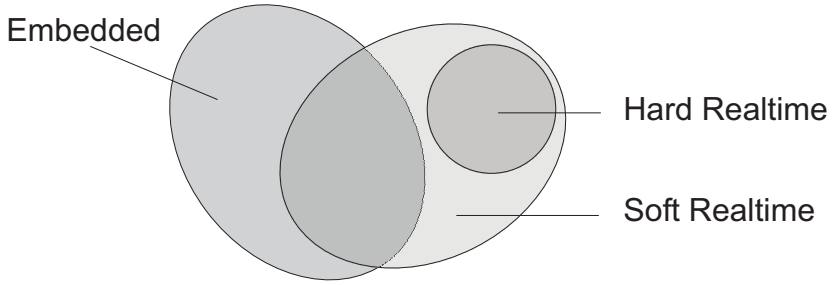


Abbildung 2.3: Marktsituation - Die meisten Embedded-Systeme stellen keine oder nur weiche Echtzeitanforderungen

traditionellen Embedded-Systeme. Der Rest der Embedded-Anwendungen profitiert zwar von kurzen Antwortzeiten, aber nur im Sinne des Durchsatzes: Es ist hinreichend für ein System, "wirklich schnell" oder "nur schnell genug" zu sein (Abb. 2.3) [84] .

In dieser Arbeit wird daher nur der Begriff Echtzeit-Systeme benutzt. Die in diesen Bereich fallenden Embedded-Systeme sind darin eingeschlossen, ohne gesondert erwähnt zu werden.

2.3 Prozessbeispiele

Nach dieser Definition nun einige Beispiele für Echtzeit-Systeme, wie sie in der Automatisierungstechnik vorkommen.

2.3.1 In der Prozesssteuerung

Eines der einfachsten Beispiele ist in Abbildung 2.4 dargestellt. Die Aufgabe des Systems ist es, den Flüssigkeitsstrom im Rohr konstant zu halten, hierzu wird die Stellung des Ventils geregelt.

Bei verändertem Durchfluss, muss der Rechner reagieren und die Stellung des Ventils anpassen. Diese Anpassung muss innerhalb einer festgelegten Zeitspanne erfolgen, damit die Änderungen des Durchflusses in den vorgegebenen Toleranzen bleiben. Die Regelung des Ventils kann dabei komplexe Berechnungen erfordern.

Dieses Beispiel kann wiederum Teil eines größeren Gesamtsystems sein. Der Computer steht über Sensoren und Aktoren mit der Umwelt in Verbindung und kontrolliert dabei die Funktion dieser Objekte.

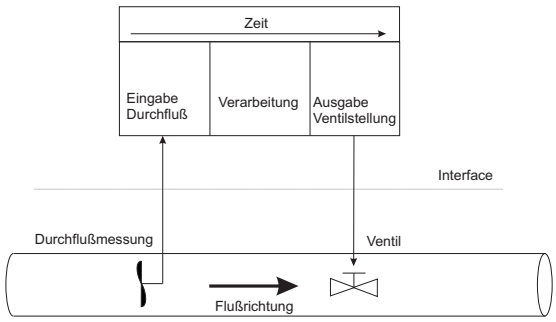


Abbildung 2.4: Ein einfaches Steuersystem

2.3.2 In der Fertigung

In der Fertigung hat der Einsatz von Computern die Kosten gesenkt und die Produktivität erhöht. Mit seiner Hilfe ist es möglich, den Informationsfluss von der Entwicklung bis zur Herstellung durchgängig zu halten. In Abbildung 2.5 ist ein System mit den für die Fertigung typischen Komponenten dargestellt. Alle Komponenten werden vom Rechner kontrolliert und koordiniert.

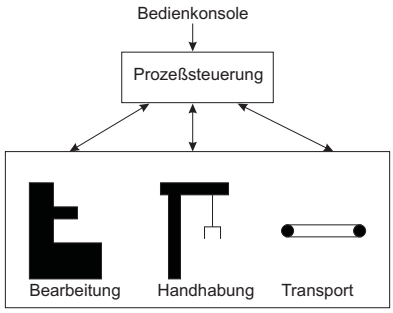


Abbildung 2.5: Ein einfaches Fertigungs-Steuerungssystem

2.3.3 Verallgemeinerung

In diesen Systeme ist der Computer mit Sensoren und Aktoren verbunden. Hierzu muss der Rechner in regelmäßigen Intervallen Messwerte aufnehmen, dies erfordert eine Echtzeituhr. Die Länge dieser Intervalle wird durch das Shannonsche Abtasttheorem [30] bestimmt.

Eine Konsole zur Bedienung des Systems zur Darstellung von Informationen und zur Administration durch den Menschen ist ein weiterer Bestandteil eines solchen Systems.

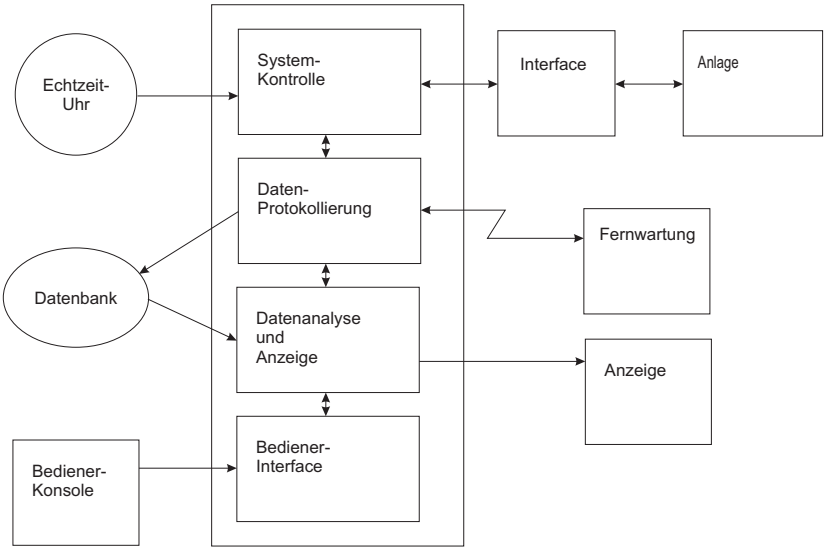


Abbildung 2.6: Ein verallgemeinertes Steuerungssystem

Für die Prozessverfolgung und die Qualitätssicherung ist es notwendig, die im System anfallenden und für den Prozess relevanten Daten in einer Datenbank abzulegen. Hierdurch ist eine genaue Analyse und eine Dokumentation des Prozesses möglich.

Ein verallgemeinertes System ist in Abbildung 2.6 dargestellt.

2.4 Charakteristika von Echtzeitsystemen

Echtzeitsysteme haben spezielle Eigenschaften, welche im folgenden indentifiziert werden. Nicht alle Systeme haben alle dieser Eigenschaften: jedes eingesetzte Bestriebssystem - und die eingesetzte Programmierumgebung, welche zur Programmierung eingesetzt wird - muss Einrichtungen bieten, die die Charakteristika unterstützen.

2.4.1 Größe und Komplexität

Im Zusammenhang mit der Entwicklung von Software stehen meist die Probleme Größe und Komplexität im Vordergrund. Der Entwurf, die Entwicklung und Wartung kleiner Programme kann ohne Probleme von einer Person durchgeführt werden. Bei Ausfall dieser Person ist es für einen Außenstehenden einfach, sich in das Programm einzuarbeiten und die weitere Wartung zu übernehmen. Es gilt: *Small is beautiful.*

Es erfüllen nicht alle Systeme diesen Anspruch. Oft ist es aber nicht nur die pure Anzahl an Codezeilen, die die Komplexität eines System beschreiben. Es hat sich gezeigt, dass diese Betrachtung zu einseitig ist. In *The characteristics of large systems* wird der Begriff *Vielfalt* als Hauptmerkmal benutzt:

> "The variety is that of needs and activities in the real world and their reflection in a program. But the real world is continuously changing. It is evolving. So too are, the needs and activities of society. Thus large programs, like all complex systems, must continuously evolve." [13]

Echtzeitfähige Systeme stehen in enger Beziehung mit ihrer Umwelt. Hieraus resultiert für das System die ständige Notwendigkeit der Veränderung und Anpassung. Es ist unmöglich, die Software auf Grund von geänderten Anforderungen kontinuierlich neu zu entwerfen und neu zu implementieren. Echtzeitsysteme bedürfen ständiger Pflege und Verbesserung. Die Systeme müssen erweiterbar sein.

Aufgabe der Programmiersprache (und der Entwicklungsumgebung) ist es, eine Möglichkeit zu schaffen, das komplexe Problem in kleinere, effektiv zu handhabende Module zerlegen zu können.

2.4.2 Behandlung von reellen Zahlen

Bei der Regelung einer Anlage wird ein Modell in Form von Differentialgleichungen benutzt. Diese verknüpfen die Ausgangsgrößen des Systems mit dem Zustand des Systems und den Eingangsgrößen. Will man die Ausgaben des Systems ändern, bedeutet dies die Lösung der Differentialgleichungen. Auf Grund dieser Schwierigkeit, der Komplexität des Modells und der unterschiedlichen (aber nicht unabhängigen) Eingaben und Ausgaben, wird die Steuerung oder Regelung in Form von Software - verbunden mit einem Computer - realisiert.

Die Lösung der Gleichungen wird von dieser Software übernommen. Die Algorithmen hierfür sind je nach Anwendungsfall kompliziert und erfordern ein hohes Maß an Rechengenauigkeit bei der Behandlung von reellen Zahlen. Die Unterstützung von Gleitkomma-Operationen und das Abfangen von Ausnahmefehlern (Exceptions) sind deshalb von großer Bedeutung. Ausnahmefehler können zum Beispiel durch eine Division durch Null hervorgerufen werden [24].

2.4.3 Sicherheit und Zuverlässigkeit

Je mehr Kontrolle dem Computer bei der Automatisierung gegeben wird, desto wichtiger ist es, dass der Computer bzw. die Software nicht versagt. Bei der Au-

tomatisierung einer chemischen Anlage kann das Versagen zu extremen Schäden und damit Kosten führen.

Es ist offensichtlich, dass Computerhardware und Software sicher und zuverlässig funktionieren müssen. Es muss möglich sein, ein System zu entwerfen, welches nur auf vorhersehbare Weise versagt. Weiterhin muss sichergestellt sein, dass auch über die eventuell vorhandenen Netzwerkschnittstellen der Software keine Gefahr für das System besteht.

2.4.4 Wechselwirkung mit Hardwareschnittstellen

In der Natur der Echtzeitsysteme in der Automatisierungstechnik liegt die Wechselwirkung mit extern angeschlossener Hardware. Es müssen Sensoren überwacht und Aktoren betätigt werden. Diese Geräte kommunizieren mit dem Computer mittels Eingabe und Ausgabe-Registern und generieren Interrupts, um dem Prozessor zu signalisieren, dass ein bestimmtes Ereignis eingetreten ist, welches eine Behandlung verlangt.

Es muss eine Unterstützung für diese Geräte geben. Diese Anforderung bezieht sich auf die Treiber für Schnittstellenkarten oder spezielle Daten Ein- und Ausgabekarten. Wird die benötigte Hardware nicht vom Betriebssystem unterstützt, so ist dies ein schwer wiegender Nachteil oder sogar das KO-Kriterium für das entsprechende Betriebssystem.

2.5 Zusammenfassung

In diesem Kapitel wurde zuerst ein Echtzeit-System definiert als ein System, welches auf extern generierte Ereignisse innerhalb einer vorher definierten endlichen Zeitspanne reagiert. Es wurden zwei Klassen unterschieden: hart und weich. Bei harter Echtzeitfähigkeit ist es unbedingt notwendig, die Zeitschranken einzuhalten, bei weicher Echtzeitfähigkeit führt ein gelegentliches Verpassen bzw. Auslassen einer Zeitschranke nicht zum Versagen des Systems.

Schließlich wurden die grundlegenden Charakteristika von Echtzeitsystemen dargestellt:

- Größe und Komplexität.
- Behandlung von reellen Zahlen.
- Zuverlässigkeit und Stabilität.
- Wechselwirkung mit Hardwareschnittstellen.

Diese Aspekte finden im Kapitel 6 ihre Beachtung.

Kapitel 3

Freie Software

3.1 Motivation

In diesem Kapitel soll nicht auf die technischen Eigenschaften des Betriebssystems und der Anwendungssoftware eingegangen werden, vielmehr soll eine Schnittstelle zwischen dem Spezialisten und dem kaufmännischen Entscheidungsträger gebildet werden. Die Besonderheiten der Philosophie der *freien Software* werden vorgestellt. Im Zusammenhang mit den Lizenzen unter denen die einzelnen Linux-Varianten, stehen ergeben sich hieraus Konsequenzen für den Einsatz der Systeme, die bei der Analyse der Varianten erläutert werden.

3.2 Entwicklung

Im folgenden Abschnitt werden die Grundlagen dargestellt, die ein Verständnis der freien Software-Bewegung ermöglichen. Hierbei werden eher herausragende Ereignisse beschrieben, die als Anfangspunkte der Entwicklung zu sehen sind, als dass der komplette Entwicklungsprozess beschrieben wird. Genau wie das Internet - welches im engen Zusammenhang mit der freien Software-Bewegung steht - verändern sich die freien Softwareprogramme bzw. Projekte ständig. Hieraus resultiert für Firmen, die Notwendigkeit, sich mit der freien Software auseinanderzusetzen. Hieraus resultiert ein ständiger Anpassungsprozess: Sie müssen sich an die aktuellen Anforderungen, die durch das wachsende Interesse der verschiedenen Zielgruppen an freier Software bedingt sind, anpassen.

3.2.1 Die Anfänge

Heutzutage fällt schon bei der Übersicht über das Angebot an (Echtzeit-)Betriebssystemen und Software die dominierende Stellung weniger großer Unternehmen

auf. In letzter Zeit berichten aber auch zunehmend Nicht-Computerzeitschriften über das Phänomen freier Software bzw. GNU/Linux, eine freie Unix-Variante (z.B. [140]).

Zuerst nur in Programmiererkreisen und an Universitäten etabliert, findet die freie Software-Bewegung zunehmend auch in der Industrie Aufmerksamkeit. Der Anteil an Ausstellern mit Produkten im Linuxumfeld auf der CeBIT in Hannover nimmt ständig zu.

Im Folgenden werden wichtige Entwicklungen, die in einem Zusammenhang zur Entwicklung der freien Software-Bewegung stehen, vorgestellt.

3.2.1.1 Unix

Die Entwicklung des Betriebssystems *Unix* - und auch der Programmiersprache "C" - in den Bell Labs von AT&T ist eng mit der Entwicklung des Internets verbunden. *Unix* entstand zur selben Zeit (Anfang der 70er Jahre) wie das Internet. Weil AT&T durch das amerikanische Monopolgesetz daran gehindert war, das Produkt kommerziell zu vermarkten, gab sie die Quellen für Lehr- und Forschungszwecke an Universitäten weiter (vgl. [61]). Dies führte zu einer sehr dynamischen Entwicklung von *Unix*. Komplexe Aufgaben werden in kleine Teilaufgaben zerlegt, die daraufhin von spezialisierten Programmen erledigt werden. Die Gesamtlösung setzt sich dann aus den einzelnen Bausteinen zusammen, die je nach Bedarf verschiedenartig kombiniert werden können.

3.2.1.2 Minix

Andrew Tanenbaum, Professor an der freien Universität Amsterdam, hat im Jahr 1987 ein Lehrbetriebssystem für den PC veröffentlicht, welches ohne jeden AT&T Code die Funktionalität von Unix Version 7 hat und als Quelltext preisgünstig zu kaufen ist.

Minix ist allerdings keine echte Basis für Anwendungsprogramme, sondern nur ein sehr lehrreiches Modell. Da *Minix* somit mit einigen Beschränkungen leben muss, es werden z.B. keine Änderungen am Kernel zugunsten von Anwendungen durchgeführt, blieb der Wunsch nach einem vollwertigen und freien Betriebssystem für die damals modernen 386-PCs unerfüllt.

3.2.1.3 Die FSF

In den Anfängen des Computerzeitalters war es für Programmierer üblich, ihre Programme untereinander auszutauschen. Diese Verfügungstellung bedeutet zugleich die Möglichkeit für jeden, den Quellcode der Programme entsprechend seinen Wünschen und Bedürfnissen anzupassen und zu modifizieren.

Das verwenden "fremden" geistigen Eigentums in eigenen "neuen" Programmen galt als durchaus legitim und normal. Erste Belege sind aus den frühen 70er Jahren dokumentiert. Zu diesem Zeitpunkt existierte allerdings noch nicht der Begriff *freie Software*.

Ein weiterer Meilenstein der freien Software-Bewegung war das Jahr 1985: In diesem Jahr wurde durch Richard M. Stallman die *Free Software Foundation (FSF)* gegründet. Ziel dieser Stiftung ist die Förderung und Verbreitung freier Software. Die FSF konzentriert sich hauptsächlich auf die verwaltungstechnischen und finanziellen Angelegenheiten des GNU-Projektes. GNU bedeutet *"GNU ist nicht gleich Unix"*. Dieses ambitionierte Projekt möchte ein komplett freies, Unix-kompatibles Software-System schaffen. Weitere Details werden im Abschnitt 3.3.0.9 behandelt. Die FSF erwirtschaftet einen Großteil ihrer Einnahmen durch die Distribution von CD-ROMs und Dokumentationen zu GNU Programmen [107].

3.2.1.4 Linux

Mitte 1991 begann der Informatikstudent Linus Torvalds einen eigenen Unix-ähnlichen Multitasking-Kernel zu schreiben [33]. Hierbei versuchte er aber nicht eine kompletten Neuentwicklung, sondern orientierte sich am Design des Minix-Systems (unter diesem wurde auch die erste Version 0.01 entwickelt). Seine Entwicklung machte Torvalds sofort frei über das Internet verfügbar. Jeder kann den Quelltext bekommen und an der weiteren Entwicklung mitarbeiten.

Die erste lauffähige Version wurde im Januar 1992 mit der Versionsnummer 0.12 veröffentlicht. Eine passende grafische Benutzeroberfläche wurde und wird im freien *Xfree86 Projekt* [113] entwickelt und später dem Kernel hinzugefügt. Die grafische Oberfläche X macht Netzwerkfähigkeit notwendig, die die 0.12er Kernel-Version nicht hatte. Diese war schwieriger zu implementieren, als Torvalds dachte [33].

Torvalds stelle Linux unter die GNU General Public License (vgl. Absch. 3.4.3.1), eine Software-Lizenz für freie Software. Ein Betriebssystem-Kernel ist zwar das 'Herzstück' eines Betriebssystems, ohne ihn ist es nicht lauffähig, doch auch der Kernel allein ist nicht verwendbar. Das GNU Projekt war mit dem Zusammentragen programmierter, freier Software soweit fortgeschritten, dass dem gesamten System nur noch ein Kernel fehlte. Das erste integrierte GNU/Linux System mit der Kernel-Version 1.0 erschien im März 1994.

3.3 Kategorien freier Software

Um mit einem Computer sinnvoll arbeiten zu können, benötigt jeder Nutzer ein Betriebssystem. Auch Betriebssysteme lassen sich, wie jede andere Software auch, in zwei Varianten unterscheiden: freie und nicht-freie. In der vorliegenden Arbeit werden nur die freien Varianten und hier insbesondere Linux betrachtet. Kommerzielle Betriebssysteme werden nicht näher behandelt, es sei denn, sie berühren direkt das Thema freie Software oder Echtzeitfähigkeit.

Um über das Themengebiet *freie Software* diskutieren zu können, ist es notwendig zuerst kurz einige Begriffe zu den Kategorien zu definieren. Diese werden in späteren Teilen der vorliegenden Arbeit wieder aufgegriffen und eingehender betrachtet.

3.3.0.5 Freie Software

Freie Software beinhaltet für jeden die Erlaubnis, diese Software zu benutzen, zu kopieren und zu vertreiben. Jeder hat das Recht die Software zu modifizieren.

Diese Freiheit verliert die Software nie. Im GNU Projekt (siehe Abschnitt 3.3.0.9) wird diese Freiheit durch das so genannte *copyleft* geschützt [142]. Der Quelltext eines Programmes muss immer mit dem Ausführungsprogramm verfügbar sein.

Der Vertrieb kann sowohl gratis geschehen als auch gegen Entgelt:

> "Does the GPL allow me to sell copies of the program for money?
> Yes, the GPL allows everyone to do this."[117]

3.3.0.6 Open Source Software

Open Source bezeichnet die gleiche Software wie *freie Software*. Der Ausdruck *Open Source* ist von der Open Source Bewegung geprägt worden, um freie Software für den professionellen Gebrauch attraktiver zu machen.

3.3.0.7 Copylefted Software

Proprietäre Software Entwickler benutzen das Copyright, um den Benutzern die Freiheit bezüglich Modifikationen und Weitergabe der Software zu nehmen. Das GNU Projekt hat diesen Begriff umgedreht, um sicherzustellen, dass freie Software auch bei Modifikation des Quelltextes als freie Software erhalten bleibt. *Copyleft* bedeutet, dass alle, die Software (mit oder ohne Änderungen) weiter verteilen, auch die Freiheit zum Weitergeben und Verändern mitgeben müssen [143].

3.3.0.8 GPL-lizensierte Software

Die GNU GPL (General Public License) ist eine Zusammenstellung von Distributionsvorschriften, wie sie für copylefted Software verwendet wird. Software des GNU Projektes wird größtenteils unter der GPL lizensiert (siehe auch Abschnitt 3.4.3.1.

3.3.0.9 Das GNU-System

Das GNU Projekt wurde 1984 mit dem Ziel gegründet, ein freies komplettes Unix-ähnliches Betriebssystem zu entwickeln. Um dieses große Ziel zu erreichen, wird dieses System aus bestehenden Projekten mit freier Software zusammengefügt. Als Kernel für dieses System kommen zwei Alternativen in Frage: Linux und Hurd. Erstere verfolgt einen monolitischen Ansatz bei der Struktur des Kernels, die zweite einen Micro-Kernel (vgl. Absch. 4.5.1 und 4.5.2).

3.3.0.10 Shareware

Software, welche unter diese Kategorie fällt, ist in der Regel während einer anfänglichen Testphase kostenlos. Nach Ablauf dieser muss jedoch eine Lizenz erworben werden. Der Quelltext wird vom Entwickler nicht offen gelegt.

3.3.0.11 Freeware

Für diese Form der Software sind keine Lizenzgebühren zu entrichten. Der Quellcode der Software ist ebenso wie bei Shareware nicht freigegeben.

3.3.0.12 Proprietäre Software

Bei proprietärer Software bleiben die Rechte, die Software zu modifizieren und zu vertreiben beim Hersteller. Der Gebrauch der Software unterliegt einer Erlaubnis des Herausgebers und geht meist mit einer käuflich zu erwerbenden Lizenz einher. Der Quelltext der Software steht nicht zur Verfügung steht.

> "By definition, proprietary software means that it isn't yours to give - someone else makes their living by selling it."[98]

Diese Erklärung ist allerdings nicht exakt genug, genauer lässt sich proprietäre Software beschreiben als Software, welche durch den Urheberrechts- bzw. Copyright-Schutz Eigentumscharakter erlangt. Der rechtmäßige Nutzer (Lizenznehmer) hat nicht das Recht, diese zu seinen Zwecken zu bearbeiten oder Kopien zu erstellen und zu vertreiben.

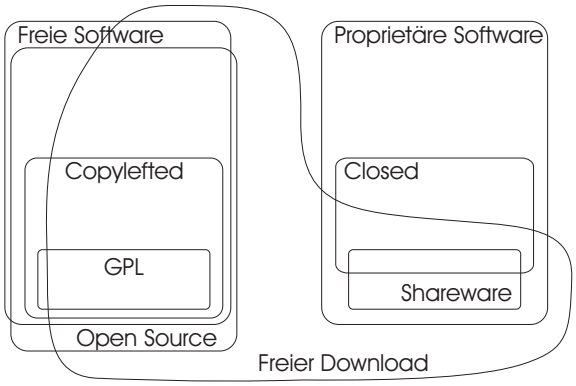


Abbildung 3.1: Klassifikation von Software [116]

3.3.0.13 Kommerzielle Software

Bei kommerzieller Software liegt das Hauptaugenmerk auf der Schaffung eines *Sale Values*, um über den Verkauf der Software einen Gewinn zu erzielen. Es kann zunächst nicht unterschieden werde, ob es sich um freie oder nicht-freie Software handelt. In der Regel ist kommerzielle Software zugleich auch proprietäre Software. Es gibt aber auch kommerzielle Ergänzungen zu freier Software.

In Abbildung 3.1 ist die Klassifizierung von Open-Source Software gegenüber anderer Software dargestellt.

3.4 Lizenzmodelle

Die aufgelisteten Software-Kategorien führen zu jeweils speziell abgestimmten Lizenzmodellen für freie Software. Um später besser die Unterschiede darzustellen, werden diese und in diesem Zusammenhang wichtige Aspekte näher betrachtet.

3.4.1 Urheberrecht

Im Urheberrecht sind alle Rechtsnormen zusammengefaßt, die sich mit dem Schutz des Schöpfers eines Werkes der Literatur, Wissenschaft und Kunst im Bezug auf seine geistige und persönliche Beziehung zum Werk und dessen Nutzung befassen. "Zu den geschützten Werken [...] gehören insbesondere Sprachwerke, wie Schriftwerke und Reden, sowie Programme für die Datenverarbeitung." [152]. Paragraph 12 Absatz 15 gewährt dem Urheber das Recht, sein Werk zu veröffentlichen, zu vervielfältigen, zu verbreiten und auszustellen. Bei Verletzung des Urheberrechts hat der Verletzte Anspruch auf Unterlassung und Schadensersatz. Im Gegensatz zum Copyright kann das Urheberrecht nicht abgegeben werden.

3.4.2 Allgemeine Software-Lizenzen

Im Allgemeinen versucht eine Software-Lizenz eine nicht autorisierte Nutzung fremden geistigen Eigentums zu verhindern. Hiermit soll eine Verletzung der Urheber- und Eigentumsrechte verhindert werden. Durch die Lizenz erhält der Benutzer die Befugnis, das Werk eines anderen zu nutzen. Hierbei ist sowohl ein Kauf als auch eine kostenlose Überlassung denkbar. Legal ist es dann nicht möglich, ein Werk weiterzugeben oder zu verändern.

3.4.3 Lizenzen freier Software

Im Bereich der freien Software werden die Lizenzen nach anderen Gesichtspunkten erstellt. Ein Weiterentwicklungsprozess ist ausdrücklich erwünscht. Ziel ist es, die freien Softwareprojekte so zu schützen, dass die Entwicklungen und Verdienste der Orginalprojekte nicht untergehen. Unter diesen Gesichtspunkten wurden spezielle Lizenzmodelle entwickelt, die diesen Ansprüchen gerecht werden. Es haben sich verschiedene Modelle herauskristallisiert, die jeweils versuchen, bestimmten Zielsetzungen gerecht zu werden. Die wichtigsten werden im folgenden vorgestellt.

3.4.3.1 GNU General Public License (GPL)

Das GNU-Projekt hat für freie Software die *GNU General Public License (GPL)* [41] entwickelt. An vielen Stellen findet sich für diese Lizenz auch die Bezeichnung *copyleft*. Durch die GPL wird festgelegt, dass durch sie lizenzierter Code nicht in proprietärer Software eingesetzt werden darf. So kann der GPL lizenzierte Linux-Kernel nicht dazu verwendet werden, einen proprietären Kernel zu erzeugen. Alle von einem GPL-lizenzierten Programm abgeleiteten Programme müssen von diesem die Freiheit des Quellcodes und Programms übernehmen.

Häufig wird diese Eigenschaft kritisiert, da die "Infektion" mit der GPL als Beschneidung der Freiheit, ein derivatives Programm unter eine andere (unter Umständen proprietäre) Lizenz zu stellen, verstanden wird.

3.4.3.2 GNU Lesser General Public License (LGPL)

Als modifizierte Variante der GPL wurde die *GNU Lesser Public License (LGPL)* [42] eingeführt. Sie erlaubt die Verwendung von freier Software auch in proprietären Programmen. Dies ist besonders für Befehlsbibliotheken interessant. Proprietäre Software darf eine solcherart lizenzierte Bibliothek benutzen, ohne selbst zur freien Software werden zu müssen. Die Bibliothek selbst kann aber nie zu proprietärer Software werden, da sie ihre Freiheit auf Grund ihrer Lizenz nie verliert.

3.4.3.3 Berkeley Software Development of Unix License (BSD)

Die am wenigsten restriktive Software-Lizenz ist die *Berkeley Software Development of Unix (BSD) License* [66]. Sie pflanzt sich nicht in die derivaten Programme fort. Es ist möglich, BSD-lizensierte Programme oder Programmteile zu verwenden, zu compilieren und dann zu verkaufen. Hierdurch ist es jedermann möglich, aus freier Software proprietäre Software zu entwickeln, um sie anschließend zu verkaufen. Hierdurch ist das Nebeneinander von freiem und proprietärem Ableger eines Programmes möglich.

Dies ist der eigentliche Unterschied zur GPL: mit GPL-lizensierte Software kann so etwas nicht geschehen.

3.4.4 Spezielle frei Software-Lizenzen

Hervorgerufen durch den immer größer werdenden Erfolg von freier Software, haben bereits mehrere Softwareunternehmen einen Teil ihrer Software freigegeben. Für diesen Anwendungsfall wurden spezielle Lizenzen entwickelt. Prominentestes Beispiel ist hier sicher Netscape mit der *Mozilla Public Licence (NPL und MozPL)* [112]. Diese Lizenzen basieren auf der GPL, lassen aber Netscape und den Entwicklungspartnern spezielle Rechte.

3.5 Die zwei Richtungen in der freien Software-Bewegung

Die Freie Software-Bewegung weist zwei unterschiedliche Strömungen auf. Auf der einen Seite die FSF und damit verbunden das GNU-Projekt, auf der anderen Seite die Open-Source Bewegung.

3.5.1 Die FSF und das GNU-Projekt

Die FSF fördert die Verbreitung freier Software und den Abbau von Restriktionen, die das Kopieren, Weiterverteilen und Verändern von Computerprogrammen betreffen. Das GNU-Projekt wird von der Free Software Foundation getragen. Hervorgegangen ist dieses Projekt aus der zu Beginn des Computerzeitalters kooperativen Grundhaltung in der Gemeinschaft der Computer-Nutzer und Programmierer. Die äußerte sich darin, dass Programme in der Regel samt Quellcode frei für jedermann zur Verfügung standen, um weiterentwickelt zu werden.

Mit dem Wachsen des Marktes für Computeranwendungen in den 80er Jahren wurde mehr und mehr Software proprietär. D.h. sie unterliegt der Erlaubnis des

Herstellers in Form einer käuflich zu erwerbenden Lizenz und alle Rechte, diese Software zu modifizieren und zu vertreiben, liegen beim Herausgeber.

Das GNU-Projekt wurde mit dem Ziel initiierte, wieder möglichst vielen Computernutzern freien Zugang zu Programmen und Quellcode zu ermöglichen. Das GNU/Linux-System ist das Vorzeigeprodukt der freien Software-Bewegung. Die Verdienste von Richard M. Stallman um die Fortentwicklung der freien Software, des GNU-Projektes und der FSF werden manchmal zugunsten der Publicity um Linux und Linus Torvalds vergessen.

Im Grunde ist Linux 'nur' der Kernel des Betriebssystems (vergl. Absch. 4.2). Sein Anteil am gesamten GNU/Linux System beträgt aber nur etwa 3% der Zeilen, wogegen ca. 28% aus dem GNU Projekt stammen (Editoren, Compiler, Kommandointerpreter usw.).

Im Gegensatz zu Stallman, der mit Leib und Seele Philosoph der freien Software Szene ist, ist Torvalds eher pragmatischer und 'business-friendly'.

Unabhängig von seinen Fähigkeiten hebt sich GNU/Linux von anderen Betriebssystemen durch seine Entwicklungsphilosophie ab: GNU/Linux ist frei. Hieraus resultiert auch der freie Vertrieb von Linux ohne Lizenzgebühren. Die gleichzeitige Freigabe des Quellcode hat den Effekt, dass es frei von firmenpolitschen Interessen weiterentwickelt wird und viel benutzerorientierter als viele kommerzielle Systeme ist. Jeder Computernutzer kann sich in die Entwicklung einbringen und z.B neue Funktionen, basierend auf den Quellen, in das System zu integrieren. Diese Mittel stehen im Gegensatz zu den Möglichkeiten, die bei der Nutzung proprietärer Software bestehen. Man kann nur Software von der Stange kaufen und hoffen, dass die angebotene Lösung auch halbwegs "passt".

3.5.2 Open Source Software

Open Source ist das Markenzeichen der Open Source Initiative. Die Idee, die hinter der Open Source Bewegung steht, ist einfach zu beschreiben: Programmierer lesen und schreiben Quellcodes, daneben ist aber das parallele Fehlersuchen und Rückmelden von Programmierern und Anwendern das Hauptmerkmal dieser Entwicklungsmethode. Hierdurch wird ein ständiger Verbesserungsprozess, eine weite Verbreitung der Software und eine Verminderung von Programmierfehlern erreicht. Die Entwicklung geschieht dezentral an den verschiedensten Orten der Welt mit einer großen Zahl von Freiwilligen. Der technische Fortschritt erfolgt in rasantem Tempo, das von Unternehmen, die Software nach klassischen Methoden entwickeln, nicht gehalten werden kann. Die so entwickelten Programme werden durch ständige Kontrolle und Gegenkontrolle laufend verbessert. Um dieses Potential auch wirtschaftlich zu nutzen, wurde die Open Source Initiative im Februar 1998 Bewegung ins Leben gerufen [124].

Dahinter steckt die Motivation, dass der schnelle, evolutionäre Prozess freier

Softwareentwicklung bessere Software hervorbringt, als die traditionellen Modelle, in dem der Quellcode dem Anwender verborgen bleibt. Die Umbennenung von *freier Software* in *Open Source Software (OSS)* geschah unter dem Gesichtspunkt, die Verwechselung zwischen "frei" und "kostenlos" zu vermeiden. Die Prinzipien dieser Organisation haben sich eindeutig in Richtung kommerzieller Nutzung freier Software verschoben. Es gilt freie Software branchenunabhängig für betriebliche Anforderungen effizient und wertschöpfungsrelevant nutzbar zu machen und damit Alternativen zu proprietärer Software aufzuzeigen.

3.5.3 Der feine Unterschied

Beide Bewegungen - "freie Software" und "Open Source" - beschreiben mehr oder weniger die gleiche Software, sagen aber verschiedene Dinge über die zugrundeliegenden Werte aus. Auf der einen Seite kämpfen die Fundamentalisten, auf der anderen die Realos. Vom GNU Projekt hört man oft die Bedenken gegen die Open Source Bewegung, dass der Profit künftig über die geistige Freiheit gestellt wird. Die Gesetze des Marktes sollen nicht die Ideen der freien Software Gemeinschaft und deren freiheitlichen Prinzipien vergessen lassen [77].

3.6 Zusammenfassung

Nach einer kurzen historischen Betrachtung über die Entwicklung der freien Software wurde Philosophie der freien Software und der damit verbundenen Lizenzmodelle beschrieben. Für die weitere Betrachtung sind hier besonders die GPL, LGPL und die Auswirkungen von Software-Patenten wichtig. Aber auch die Zugänglichkeit vom Quelltext bei freier bzw. proprietärer Software wird später von Bedeutung sein.

Kapitel 4

Die Theorie

4.1 Motivation

Bevor in Kapitel 7 eine Unterscheidung und Auflistung der verschiedenen Systeme stattfinden kann, gilt es zunächst herauszustellen, was ein Betriebssystem überhaupt ist und welche Aufgaben es erfüllen muss.

4.2 Das Betriebssystem

Ein modernes Computersystem besteht aus einem oder mehreren Prozessoren, dem Hauptspeicher, Timern, Festplatten, Netzwerkkarten und anderen Ein- und Ausgabegeräten, ein sogenannter "von Neumann"-Rechner. Um eine einfache Entwicklung von Anwendungen für dieses komplexe System zu ermöglichen, ist es notwendig, die Hardware vor den Softwareentwicklern zu verbergen. Es wird eine Softwareschicht über die Hardware gelegt, welche dem Benutzer eine einfach zu verstehende und zu programmierende Schnittstelle bietet. Diese Softwareschicht ist der Kernel. Somit ergibt sich folgender Aufbau eines Computersystems (Abb. 4.1).

Die unterste Schicht besteht aus den physikalischen Bausteinen. Diese umfassen den oder die Prozessoren des Systems, den Hauptspeicher sowie die Ein- und Ausgabe-Geräte. Die Ein- und Ausgabe-Einheiten werden durch Zugriff auf die entsprechenden Kontrollregister angesteuert. Der Kernel hat die Aufgabe, die Komplexität der Hardware zu verbergen und dem Programmierer einen komfortablen Zugriff auf die Ressourcen zu ermöglichen.

Als nächste Schicht kommt die restliche Systemsoftware. Hier sind zu nennen: der Kommandointerpreter, Compiler, Editoren usw.

Im engeren Sinne umfasst ein Betriebsystem nur den Kernel. Die restlichen

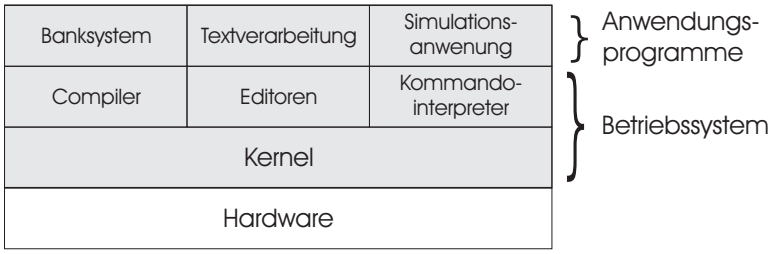


Abbildung 4.1: Struktur eines Computersystems

Programme gehören nicht zum eigentlichen Betriebssystem, sondern sind eine Erweiterung dessen. Sie werden in der Regel mit dem Kernel ausgeliefert. Soweit nicht anders vermerkt, ist in dieser Arbeit Betriebssystem identisch mit dem Kernel.

Um eine klare Abgrenzung zu liefern: der Kernel ist der Teil der Software, welcher im *Kernel-* oder *Supervisormode* des Prozessors ausgeführt wird. Compiler und Editoren laufen im *Usermode*.

Als letzte Schicht kommen die Anwendungsprogramme. Diese Programme dienen dazu, bestimmte Aufgaben für den Benutzer zu erledigen.

Was zeichnet den Supervisormode gegenüber dem Usermode aus?

Privilegienebenen

Ein wesentlicher Aspekt für die Betriebssicherheit eines Rechners ist der Schutz der für den Betrieb erforderlichen Systemsoftware (der Kernel) gegenüber unerlaubten Zugriffen durch Anwendersoftware (Benutzerprogramme). Grundlage hierfür sind die in der Hardware verankerten Betriebsarten zur Vergabe von Privilegienebenen. In den allgemein üblichen Zwei-Ebenen-Systemen laufen die elementaren Funktionen im priviligierten Supervisormode und die Benutzeraktivitäten (hierzu zählen auch die Erweiterungen des Betriebsystems) laufen im nichtpriviligierten Usermode. Festgelegt wird die jeweilige Betriebsart durch ein Modusbit im Statusregister des Prozessors.

Der Schutzmechanismus besteht zum einen in der prozessorexternen Anzeige der jeweiligen Betriebsart durch die Statussignale des Prozessors. Diese kann durch eine Speicherverwaltungseinheit dazu genutzt werden, Zugriffe von im Usermode laufenden Programmen für bestimmte Adressbereiche einzuschränken oder sie ganz zu unterbinden. Demgegenüber werden der Supervisorebene die vollen Zugriffsrechte eingeräumt.

Ein weiterer Schutz bieten die sogenannten *priviligierten Befehle*, die nur im Supervisormode ausführbar sind. Zu ihnen zählen alle Befehle, mit denen die

Modusbits im Statusregister verändert werden können, so auch die Befehle für die Umschaltung in den Usermode. Programmen, die im Usermode laufen, ist über die Einschränkung der Zugriffe auf die priviligierten Adressbereiche hinaus die Ausführung priviligierter Befehle verwehrt.

Der Übergang vom Usermode in den Supervisormode geschieht durch einen priviligierten Befehl, der Übergang vom User- in den Supervisormode durch einen sog. Betriebssystemaufruf (vergl. Absch. 4.4.5.

4.3 Was ist ein Betriebssystem

Mit Betriebssystem wird im Allgemeinen die Software bezeichnet, welche für den Betrieb eines Computers erforderlich ist. Diese Definition zeigt, dass es nicht sehr einfach ist, präzise zu formulieren, was ein Betriebssystem ist.

Ein Betriebssystem kann man unter zwei Sichtweisen betrachten:

- Das Betriebssystem als erweiterte Maschine.
- Das Betriebssystem als Ressourcenmanager.

Im Folgenden werden beide beschrieben.

4.3.1 Das Betriebssystem als erweiterte Maschine

Die erste Aufgabe des Betriebssystem erschließt sich aus einer Top-down-Sicht. Hieraus ergibt sich die Aufgabe des Betriebssystems, dem Benutzer das Äquivalent einer *erweiterten Maschine* bzw. einer *virtuellen Maschine* zu präsentieren, die leichter zu programmieren ist, als die darunterliegende Hardware. Das Betriebsystem vereinfacht das Programmiermodell der Maschine durch Bereitstellung von Funktionen auf höherem Abstraktionsniveau.

4.3.2 Das Betriebssystem als Ressourcenmanager

Eine Alternative zur oben angeführten Top-down-Sicht ist die Bottom-up-Sicht. Bei dieser Betrachtungsweise ergibt sich als Aufgabe des Betriebssystems die Verwaltung aller Bestandteile eines komplexen Systems. Für das Betriebssystem heißt es, die Prozessoren, den Speicher und die I/O-Geräte den konkurrierenden Programmen geordnet zuzuteilen.

Das Betriebssystem sorgt z.B. dafür, dass mehrere gleichzeitig anstehende File-Input-/Output-Operationen in einer geordneten Art und Weise nacheinander ausgeführt werden. Ein anderer interessanter Fall ergibt sich, wenn mehrere

Benutzer gleichzeitig einen Computer benutzen. Hieraus folgt noch klarer die Notwendigkeit, Speicher, I/O-Geräte und andere Ressourcen zu verwalten und zu schützen.

4.4 Betriebssystem-Konzepte

Die Schnittstelle zwischen dem Betriebssystem und den Benutzerprogrammen ist definiert durch einen Satz von "erweiterten Befehlen", die das Betriebssystem zur Verfügung stellt. Diese *Systemaufrufe* erzeugen, zerstören und benutzen verschieden Software-Objekte, die durch das Betriebssystem verwaltet werden.

Die wichtigsten dieser Objekte sind die *Prozesse*. Diesen kommt auch bei der späteren Betrachtung unter dem Aspekt der Echtzeitverarbeitung eine besondere Bedeutung zu.

4.4.1 Prozesse

Ein Prozess (bzw. Task) ist ein im Speicher laufendes Programm, also die Ausführung des Programmcodes und nicht der Code selber. Ein Programm ist, wie eine Datei, passiv, während ein Prozess mit Programmzeiger und Ressourcen aktiv ist.

Die Idee bei der effizienten Nutzung der CPU durch Multiprogramming ist, dass mehrere Prozesse gleichzeitig im Speicher sind und andere Prozesse davon profitieren, wenn einer auf Eingabe oder Ausgabe warten muss. Diese Zeit würde sinnlos verstreichen, würde nur ein Prozess ausgeführt. Der wartende Prozess wird also unterbrochen und ein anderer Prozess bekommt CPU-Zeit zur Verfügung. Die Idee, die hinter diesem Konzept steht besteht darin, jeden Prozess als eine Einheit aus Programm, Eingaben, Ausgaben und einem Zustand zu sehen. Ein einzelner Prozessor kann mit einem Scheduling-Algorithmus, welcher bestimmt, wann von einem Prozess zu einem anderen gewechselt wird, zwischen Prozessen aufgeteilt werden.

Im Kontext der Echtzeitverarbeitung kommt dem Konzept der Prozesse ein besondere Bedeutung zu: Eine Anwendung in der "Realität"/"Real World" muss z.B. mit Robotorarmen, Motorreglungen, Temperatursensoren, Endschaltern kommunizieren und auf Ereignisse reagieren. Das Besondere hieran ist, dass dies alles "gleichzeitig" passieren muss.

Ein Steuerungssoftware für ein autonomes Fahrzeug (wie z.B. den Mars-Explorer) habe die folgenden Aufgaben zu erfüllen:

- Navigation im Gelände.
- Kontrollieren, ob sich noch alle Räder auf dem Boden befinden.

- Steuerung der verschiedenen Antriebsmotoren.
- Erfassung der unterschiedlichsten Umgebungsmesswerte, wie Temperatur oder Helligkeit.
- Sammeln von Proben.

Die Aufgabe der Software ist es , schnell genug auf die verschiedenen Eingaben zu reagieren und Ausgaben zu erzeugen (siehe Absch. 2.2). Die Eingaben und Ausgaben können hierbei völlig unabhängig voneinander sein, oder in engen Zusammenhang stehen. Wie sehen die möglichen Strukturen des Programms hierfür aus? Es gibt prinzipiell fünf Alternativen [48]:

Verwendung eines Prozesses Alle verschiedenen Eingaben und Ausgaben werden in einem einzelnen Prozess verarbeitet. Der Verwaltungsaufwand für mehrere Prozesse wird hierdurch vermieden, führt aber zu einem komplexen Code. Die Wartung und Erweiterung dieses Systems wird hierdurch erheblich erschwert.

Verwendung mehrerer Prozesse Für jede Aufgabe die die Software erfüllen muss, wird ein eigener Prozess erzeugt. Dieser Ansatz vermeidet die Komplexität eines einzelnen Prozesses, erzeugt aber das Problem, wie die einzelnen Prozesse ihre Aktivitäten koordinieren. Weiterhin hat diese Lösung ein Problem mit der Performance und Skalierbarkeit.

Verwendung nicht ganz so vieler Prozesse Wenn ein Prozess pro Aktivität zu viel ist, werden mehrere Aktivitäten in einem Prozess zusammengefasst. Hierbei besteht das Problem, geeignete Aktivitäten zu finden, welche sich problemlos kombinieren lassen, um eine klare Struktur aufrecht zu erhalten. Für kritische Anwendungen stellt sich oft heraus, dass Komplexität Fehler versteckt.

Verwendung von Signalen Signale ahmen mehrere Prozesse nach. Wenn jede Ein- und Ausgabe ein Signal erzeugen kann, ist es möglich, diesen Ansatz zu verfolgen. Ein Signal-Handler kann als unabhängiger, asynchroner Kontrollfluß in einem Prozess angesehen werden, der nur auf bestimmte Signale reagiert. Das Problem bei der Verwendung von Signalen ist, dass es für das Hauptprogramm so aussieht, als wären es eingenständige Kontrollströme, sie es aber gar nicht sind.

Die Verwendung von Threads Das Konzept der Threads bietet mehrere Tasks innerhalb eines einzelnen Prozessadressraums (siehe Abb.4.2). Rechts ist ein Prozess mit mehreren Kontrollflüssen, die als Threads bezeichnet werden, dargestellt.

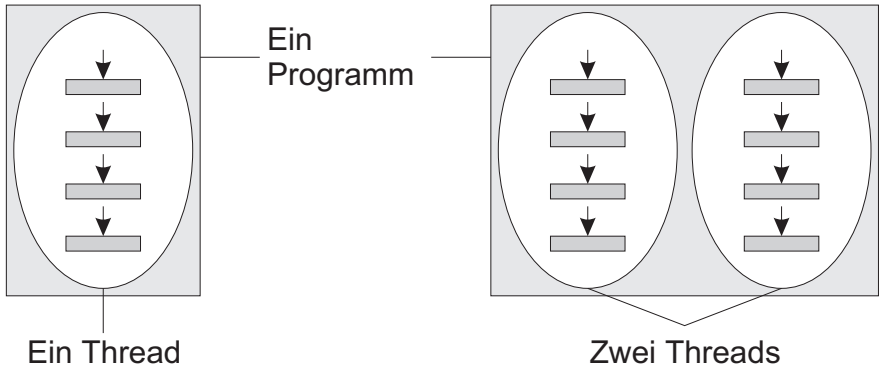


Abbildung 4.2: Das Konzept der Threads

Die aufgeführten Alternativen finden alle ihre Anwendung. Allerdings haben die erste und letzte schwerwiegende Einschränkungen, die bestimmen, wie groß und portabel die Problemlösung wird.

Wie man am obigen Beispiel erkennen kann, ist es notwendig, dass die an sich unabhängigen Prozesse miteinander interagieren. Ein Prozess erzeugt zum Beispiel eine Ausgabe, die von einem weiteren als Eingabe benutzt wird. Im Folgenden Kommando

```
cat datei1 datei2 datei3 | grep Fehler
```

liest der erste Prozess der das Kommando `cat` ausführt drei Dateien und gibt sie aus. Der zweite Prozess - mit dem Programm `grep` durchsucht die Ausgabe nach dem Wort "Fehler". Je nach Ausführungsgeschwindigkeit der Prozesse kann es dazu kommen, dass der Prozess der `grep` ausführt, im Prinzip weiter ausgeführt werden könnte, aber keine weiteren Eingaben vorliegen. Er ist so lange *blockiert*, bis die Eingabe zur Verfügung steht. Ein weiterer Grund für die Suspensierung eines Prozesses liegt nicht in einem Problem begründet, sondern darin, das das Betriebssystem u.U. die CPU einem anderen Prozess zugeteilt hat.

Hieraus ergeben sich drei Zustände, in denen sich ein Prozess befinden kann (Abb. 4.3):

1. *rechnend* - Der Prozess ist dem Prozessor zugeteilt.

2. *rechenbereit* - Der Prozess ist ausführbar, aber der Prozessor ist einem anderen Prozess zugeteilt.

3. *blockiert* - Der Prozess kann nicht ausgeführt werden, bis ein externes Ereignis eintritt.

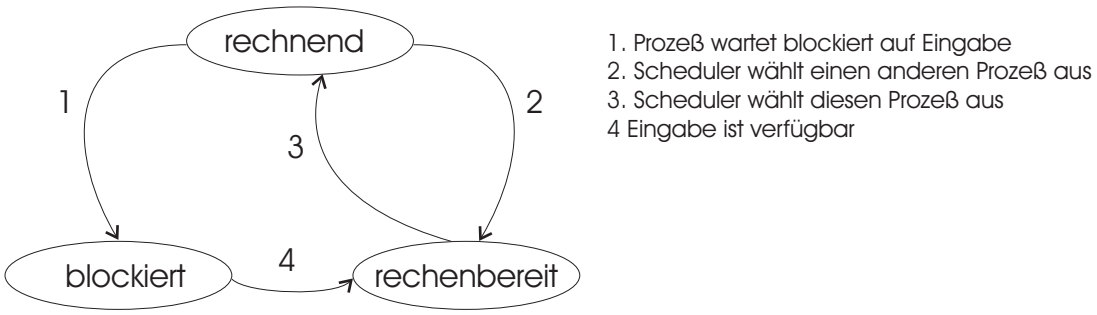


Abbildung 4.3: Die Prozesszustände und ihre Übergänge [147].

Wie in Abbildung 4.3 dargestellt ist, sind vier Übergänge zwischen den Zuständen möglich. Übergang 1 wird ausgeführt, wenn der Prozess nicht weiter ausgeführt werden kann, er z.B. eine Eingabe erwartet, aber noch keine vorliegt.

Die Übergänge 2 und 3 werden durch den Prozess-Scheduler (der ein Teil des Betriebssystems ist) veranlasst, ohne dass die Prozesse selbst etwas davon merken. Übergang 2 wird veranlasst, wenn der Scheduler die CPU einem anderen Prozess zuteilt. Bei Übergang 3 haben alle anderen Prozesse ihre Rechenzeit verbraucht und der erste Prozess kommt wieder an die Reihe. Nach welchen Kriterien der Scheduler entscheidet, wann ein Prozess Rechenzeit zugeteilt bekommt, wird im Abschnitt 4.4.3 besprochenerläutert.

Der Übergang 4 geschieht, wenn das externe Ereignis, auf das der Prozess wartet, eintritt. Wenn kein anderer Prozess ausgeführt wird, wird Übergang 3 ausgeführt und der Prozess kann seine Ausführung fortsetzen. Andernfalls verbleibt er im Zustand *rechenbereit*.

Mit Hilfe des Prozessmodells ist es einfach, zu beschreiben, was innerhalb eines Systems geschieht. Ein Teil der Prozesse ist Teil des Betriebssystems und führt Aufgaben wie die Verwaltung von Plattenspeicher aus. Ein anderer Teil sind vom Benutzer gestartete Programme.

Das Betriebssystem besteht dann in der untersten Schicht aus dem Scheduler, der für die Unterbrechungsbehandlung und die Erzeugung und das Abbrechen von Prozessen zuständig ist. Der Rest besteht aus einer Vielzahl von Prozessen in der Schicht darüber (Abb. 4.4).

4.4.2 Kommunikation und Synchronisation

Die verschiedenen Prozesse müssen untereinander kommunizieren. Dies kann im Beispiel der Shell-Pipeline die Übergabe der Ausgabe an den Prozess sein. Diese Kommunikation sollte in einer wohl definierten Art und Weise und ohne die Benutzung von Unterbrechungen geschehen. Es lassen sich zwei Bereiche unter-

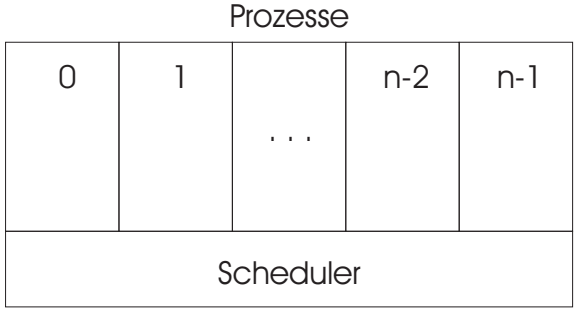


Abbildung 4.4: Schichten des Betriebssystems [147].

scheiden, die von Bedeutung sind: Synchronisation im Kernel und Synchronisation zwischen Benutzer-Prozessen

4.4.2.1 Kernel Synchronisation

Die in dieser Arbeit betrachteten Linux-Varianten benutzen alle Kernel, welche *reentrant* sind. das heißt, mehrere Prozesse befinden sich im Kern-Modus (vgl. hierzu Absch. 4.5.1). Auf einem Einprozessor-System befindet sich natürlich nur ein Prozess in der Ausführung, es können aber weitere Prozesse, welche auf die CPU warten, im Kernelmodus blockiert sein.

Es muss gewährleistet sein, dass ein Auführungspfad im Kernel, welcher unterbrochen wird, während er auf Datenstrukturen im Kernel zugreift, kein anderer Ausführungspfad diese Daten modifiziert. Wird dies nicht sichergestellt, sind korrupte Daten das Resultat.

Ein Beispiel: Die globale Variable V enthalte die Anzahl an freien Zugriffen auf eine Systemresource. Der erste Ausführungspfad A liest diese Variable V und stellt fest, dass genau ein Zugriff möglich ist. An diesem Punkt erhält der zweite Ausführungspfad B die Kontrolle und liest ebenfalls V aus, erniedrigt den Wert um eins und beginnt die Resource zu benutzen. Jetzt gelangt Pfad em A wieder zur Ausführung, nimmt an, dass er V erniedrigen kann und benutzt ebenfalls die Resource, welche B bereits benutzt. Das Ergebnis ist, dass V den Wert -1 besitzt und zwei Ausführungspfade die Resource zur gleichen Zeit mit zerstörerischen Folgen benutzen.

Wenn das Ergebnis einer Berechnung davon abhängt, in welcher Reihenfolge zwei oder mehr Prozesse zur Ausführung kommen, dann ist der Programmcode inkorrekt: Dies wird als *race condition* bezeichnet.

In der Regel wird der sichere Zugriff auf globale Variablen durch die Benutzung nicht unterbrechbarer Operationen (*atomic operations*)sichergestellt. Auf viele Datenstrukturen kann aber nicht mit einer einzelnen Operation zugegriffen

werden. Das Löschen aus einer doppelt-verlinkten Liste z.B. erfordert den Zugriff auf zwei Zeiger (Verweis auf eine Stelle im Speicher) zur gleichen Zeit. Jeder Bereich eines Programmscodes, welcher abgearbeitet sein muss, bevor ein zweiter Prozess mit der Ausführung beginnt wird als *kritischer Bereich* bezeichnet.

4.4.2.1.1 Nicht unterbrechbarer Kernel Um die Probleme bei der Synchronisation zu vereinfachen, besteht eine Möglichkeit darin, den Kernel nicht unterbrechbar zu machen (nicht *präemptiv*). Wenn ein Prozess sich im Kernmodus befindet, kann er nicht unterbrochen und durch einen anderen Prozess ausgetauscht werden. Auf einem Einprozessor-System ist somit sichergestellt, dass auf alle Kernel-Datenstrukturen, welche nicht durch Interrupts verändert werden, gefahrlos zugegriffen werden kann.

Auf Systemen mit mehreren Prozessoren ist ein führt ein nicht unterbrechbarer Kernel nicht zur Lösung, da dort ein anderer Ausführungspfad auf einer weiteren CPU gleichzeitig auf die Kernel-Datenstrukturen zugreifen kann.

4.4.2.1.2 Abschalten der Interrupts Das Abschalten der Interrupts ist auf Ein-Prozessor-Systemen ein weiterer Ansatz, um Synchronität zu erhalten. Bevor ein Prozess auf eine kritische Datenstruktur zugreift, werden alle Interrupts gesperrt und erst nachdem der Prozess die Datenstruktur freigibt wieder aktiviert. Hierdurch wird verhinder, dass der Scheduler durch das Auftreten eines Interrupts zur Ausführung kommt und eventuell die Kontrolle an einen anderen Prozess übergibt.

Obwohl dieser Mechanismus sehr einfach ist, ist er alles andere als optimal: Es ist durchaus wahrscheinlich, dass ein Prozess sich relativ lange in einem kritischen Bereich befindet und somit auch die Interrupts für eine lange Zeit gesperrt sind. Hieraus resultiert ein Einfrieren aller Hardwareaktivitäten.

Auch den bereits erwähnten Systemen mit mehr als einem Prozessor funktioniert dieser Mechanismus nicht, da ein zweiter Prozess auch ohne Scheduling in den kritischen Bereich eintreten kann - er läuft auf einer eigene CPU und erzeugt somit eine *race condition*.

4.4.2.1.3 Kernel-Semaphoren Eine gebräuchliche Methode, Datenstrukturen zu schützen, sind Semaphoren. Diese schützen einen kritischen Bereich vor Zugriffen durch weitere konkurierende Prozesse. Eine durch eine Semaphore geschützte Ressource ist vergleichbar mit einem Raum, dessen Tür verschlossen ist. Will ein Prozess diese Ressource nutzen, versucht er, den Raum zu betreten, was ihm aber nicht gelingt, da die Tür verschlossen ist.

Kernel-Semaphoren sind zwei Datenfelder zugeordnet:

Zähler Dieser dient dazu, festzuhalten, ob die Ressource frei ist bzw. die Anzahl der wartenden Prozesse zu speichern

Warteschlangenliste Verweist auf eine Liste mit den wartenden Prozessen, ist die Ressource frei, so ist die Liste folglich leer.

4.4.2.1.4 Spin Locks *Spin locks* sind ein Mechanismus, welcher für den Einsatz in Multi-Prozessor-System gedacht ist. Diese sind ähnlich den bereits beschriebenen Semaphoren. Sobald ein Prozess auf eine durch eine Semaphore geschützten Bereich trifft, tritt er in eine Warteschleife ein (*"Spin"*). Auf Systemen mit nur einem Prozessor macht dieser Mechanismus so keinen Sinn, da der wartende Prozess ununterbrochen die Warteschleife durchläuft und so der Prozess, welcher die Ressource blockiert nicht mehr zu Ausführung kommen kann und somit die Ressource auch nicht mehr freigeben kann.

Auf Systemen mit mehreren Prozessoren haben Spin Locks einige Vorteile: zum einen ist der Overhead sehr klein und zum anderen ist es für einen Prozess günstig, "seine" CPU zu behalten und zu warten, bis die Ressource, die er benötigt, frei wird.

4.4.2.2 Prozess Kommunikation

Die bisher beschriebenen Mechanismen dienen dazu, Daten im Kernel zu schützen und synchron zu halten. Aber auch für Programme im Benutzer-Modus (vgl. hierzu Absch. 4.5.1) besteht die Notwendigkeit für Kommunikationsmöglichkeiten. Hierzu gibt es verschiedene Mechanismen.

4.4.2.2.1 Pipes Eine *Pipe* ist ein Kommunikationskanal zwischen zwei Prozessen. Alle Daten die von dem einen Prozess in die Pipe geschrieben werden, werden zum zweiten geleitet, welcher diese dann lesen und verarbeiten kann. Die Kommunikation durch eine Pipe findet nur in einer Richtung statt.

Pipes eignen sich für die Kommunikation zwischen Erzeuger- und Verbraucherprozessen. Ein Prozess erfasst z.B. Messdaten während der zweite diese weiterverarbeitet.

4.4.2.2.2 FIFOs(named Pipes) Obwohl Pipes ein einfaches und flexibles Kommunikationsmittel sind, haben sie einen großen Nachteil: es ist nicht möglich, eine bereits existierende Pipe zu öffnen. D.h. es ist nicht ohne weiteres möglich, eine Pipe von zwei Prozessen gleichzeitig zur Kommunikation mit einem dritten zu nutzen.

Um diese Beschränkung aufzuheben wurde eine sogenannte *named Pipe* bzw. ein *FIFO* eingeführt. FIFO steht hierbei für "first in, first out" - das erste Byte

welches in eine named Pipe geschrieben wird, wird auch als erstes wieder gelesen. Dadurch dass FIFOs einen Namen besitzen, ist es möglich, diese aus unterschiedlichen Prozessen heraus anzusprechen.

4.4.2.2.3 IPC-Semaphoren IPC-Semaphoren zur Kommunikation zwischen Prozessen (**I**nter **P**rozess **C**ommunication) sind ähnlich den bereits beschriebenen Kernel-Semaphoren. Sie sind Zähler, welche kontrollierten Zugriff auf gemeinsam benutzte Datenstrukturen erlauben durch mehrere Prozesse erlauben.

Im Gegensatz zu Kernel-Semaphoren können IPC-Semaphoren aber mehrere unabhängige Ressourcen schützen. Ausserdem existiert für IPC-Semaphoren ein Mechanismus, um Änderungen die unter dem Schutz einer Semaphore ausgeführt wurden, rückgängig zu machen. Terminiert ein Prozess auf nicht geordnete Weise (er "stirbt"), werden alle Werte in den Zustand zurück gesetzt, als ob der Prozess nie gelaufen wäre.

4.4.2.2.4 Shared Memory Der mächtigste Mechanismus für die Prozesskommunikation ist das *shared Memory*. Zwei Prozesse können auf die gleichen Datenstrukturen zugreifen, indem sie diese in einem gemeinsam genutzten Speicherbereich ablegen.

Mittels der beschriebenen Mechanismen ist es möglich, sowohl im Kern- als auch im Benutzer-Modus Daten synchron zu halten und zwischen Prozessen zu kommunizieren.

Das Vorhandensein mehrerer Prozesse macht eine Verwaltung dieser nötig:

4.4.3 Scheduling

Wie bereits in Abschnitt 4.4.1 gezeigt, ist es nicht nur bei Echtzeitsystemen üblich, dass mehrere Prozesse ausführbar warten. In diesen Fällen muss das Betriebssystem entscheiden, welcher Prozess als erster ausgeführt wird. Der Teil des Betriebssystems, welcher diese Aufgabe übernimmt, wird *Scheduler* genannt. Der Algorithmus, nach dem er seine Entscheidung trifft, wird als *Scheduling-Algorithmus* bezeichnet.

Auf Basis einer vorgegebenen Strategie muss der Scheduler Entscheidungen treffen und nicht einen Mechanismus zur Verfügung zu stellen. Kriterien für mögliche Scheduling-Algorithmen sind z.B.:

Fairness Jeder Prozess enthält einen gerechten Anteil der Prozessorzeit.

Effizienz Der Prozessor ist immer voll ausgelastet.

Antwortzeit Die Antwortzeit für interaktiv arbeitende Benutzer wird minimiert.

Verweilzeit Die Wartezeit für die Ausgabe von Stapelaufträgen wird minimiert.

Durchsatz Die Anzahl der Aufträge, die in einem bestimmten Zeitintervall ausgeführt werden, wird maximiert.

Hierbei ist zu beachten, dass sich diese Kriterien teilweise widersprechen. Jeder Scheduling-Algorithmus, der eine Klasse von Aufträgen bevorzugt, z.B. die Stapelaufträge, benachteiligt die anderen [71]. Die Schwierigkeit hierbei ist, dass der Scheduler keine Vorhersagen über den Verlauf der Ausführung von Prozessen machen kann. Wenn ein Prozess zur Ausführung gebracht wird, weiß der Scheduler nicht, wie lange es dauern wird, bis der Prozess blockiert, weil er auf I/O-, auf ein Semaphor oder aus einem anderen Grund warten muss. Um sicherzustellen, dass kein Prozess zu lange ausgeführt wird, besitzen die meisten Computer einen Timer, der in regelmäßigen Abständen einen Interrupt auslöst. Bei jeder Unterbrechung erhält das Betriebssystem die Kontrolle und entscheidet dann, ob der Prozess weiter ausgeführt wird oder ob ein anderer Prozess zur Ausführung kommt. Dieses Verfahren wird *preemptive-scheduling* genannt. Frühere Stapelsysteme arbeiteten nach dem *Run-To-Completion*-Verfahren, dies wird auch als *nonpreemptive-scheduling*-Verfahren bezeichnet. Für Echtzeit-Systeme ist dieses nicht geeignet, da hier keine Antwortzeiten garantiert werden können. Im Folgenden werden einige gebräuchliche Algorithmen vorgestellt und im Bezug auf ihre Vor- und Nachteile betrachtet.

4.4.3.1 Round-Robin-Scheduling

Der älteste und vielleicht einfachste Algorithmus ist Round-Robin. Bei diesem wird der nächste zur Ausführung kommende Prozess aus einer Warteschlange nach der FIFO-Reihenfolge ausgewählt. Wenn der Prozess für eine bestimmtes Zeitintervall - das *Quantum* genannt wird - aktiv war, wird er unterbrochen und als letzter in die Warteschlange eingereiht. Jeder Prozess ist also für eine bestimmte Zeitscheibe aktiv (siehe Abb. 4.5).

Der entscheidende Faktor beim Round-Robin-Verfahren ist die Länge des Quantums. Jeder Prozesswechsel benötigt eine gewisse Zeit für die Verwaltung. Angenommen diese Zeit sei 5 Millisekunden, so werden bei einem Quantum von 20 Millisekunden 20 Prozent der Prozessorzeit für Verwaltungsaufgaben verschwendet. Es gilt also, einen guten Kompromiss für die Länge eines Quantums zu finden, um bei möglichst guter Prozessorauslastung noch gute Antwortzeiten des Systems zu erhalten.

4.4.3.2 Prioritäts-Scheduling

Beim Round-Robin-Scheduling wird die Annahme getroffen, alle Prozesse seien gleich wichtig. Dies trifft aber gerade bei Echtzeitverarbeitung nicht zu. Hier sind

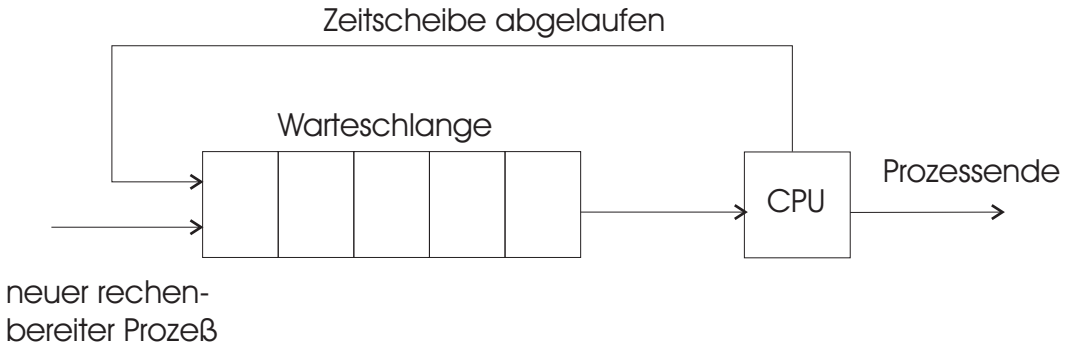


Abbildung 4.5: Round-Robin-Scheduling.

Prozesse die nur eine Visualisierung des Systems machen nicht so wichtig, wie Steuerungs- und Regelungs-Prozesse. Diese Betrachtung führt zum *Prioritäts-Scheduling*: jedem Prozess wird eine Priorität zugewiesen und der ausführbare Prozess mit der höchsten Priorität wird ausgeführt. Um zu verhindern, dass Prozesse mit hoher Priorität zu lange ausgeführt werde, erniedrigt der Scheduler bei jedem Timerinterrupt die Priorität. Wenn hierdurch seine Priorität unter die eines anderen Prozesses fällt, wird ein Prozesswechsel durchgeführt.

Die für die Vergabe der Prioritäten denkbaren Verfahren - statische und dynamische Zuweisung - werden im folgenden dargestellt.

4.4.3.2.1 Statische Zuweisung Hier erfolgt die Zuweisung der Priorität nach festen Kriterien. Prozesse in einem kommerziellen Rechenzentrum bekommen z.B. nach dem Preis pro Stunde eine Priorität zugewiesen.

4.4.3.2.2 Dynamische Zuweisung Hier erfolgt die Zuweisung dynamisch durch das System, um bestimmte Zielsetzungen zu erreichen. I/O-intensive Prozesse sollten sofort den Prozessor zugeteilt bekommen, damit sofort die nächste I/O-Operation gestartet werden kann, die dann parallel zu einem Rechenprozess ausgeführt werden kann.

Prozesse werden häufig in Prioritätsklassen eingeteilt, wobei zwischen den Klassen ein Prioritäts-Scheduling und innerhalb ein Round-Robin-Scheduling durchgeführt wird (siehe Abb. 4.6). Hierbei ist es wichtig, die Prioritäten von Zeit zu Zeit anzupassen, da es sonst vorkommen kann, dass Prozesse mit niedriger Priorität nie ausgeführt werden.

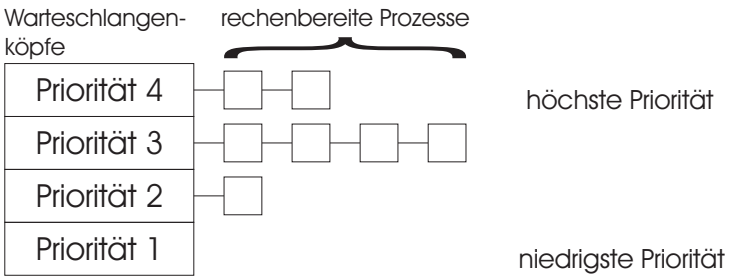


Abbildung 4.6: Scheduling mit vier Prioritätsklassen [147].

4.4.3.3 Shortest-Job-First oder Rate Monotonic

Bei Anwendungen, bei denen die Ausführungszeiten im voraus bekannt sind, bietet sich der Algorithmus *Shortest-Job-First* an, dieser wird auch als *Rate Monotonic* bezeichnet. Dies ist im wesentlichen ein Scheduling mit Prioritäten, wobei die Priorität bestimmt wird durch den Kehrwert der erwarteten Ausführungszeit.

Dieser Algorithmus scheint sehr gerecht zu sein. Er sorgt für ein kleines Verhältnis aus Wartezeit zu Ausführungszeit.

Ein Problem ergibt sich für interaktive Prozesse. Diese besitzen im Allgemeinen keine feste Ausführungsmuster. Man kann versuchen die ungefähre Ausführungszeit aus der Vergangenheit zu schätzen und aufgrund dieser Schätzung den nächsten auszuführenden Job zu bestimmen. Mit T_0 für die geschätzte Ausführungszeit eines Kommandos und T_1 als gemessene Zeit für die erste Ausführung kann durch die Berechnung der gewichteten Summe $aT_0+(1-a)T_1$ eine neue Schätzung berechnet werden. Der Wert für a beeinflusst hierbei, wie lange die Dauer der vergangenen Ausführungen behalten werden soll. Für einen Wert von $a = 1/2$ erhalten wir folgende Reihe von Schätzungen:

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Nach drei neuen Ausführungen ist das Gewicht von T_0 auf $1/8$ gesunken.

Eine Möglichkeit ist die Schätzung durch die Berechnung des gewichteten Durchschnitts des aktuellen und des vorhergehenden Wertes, dies wird als *Alterung* bezeichnet. Mit $a = 1/2$ ist die Alterung besonders einfach zu implementieren, da nur der neue Wert zur aktuellen Schätzung addiert werden muss und das Resultat durch zwei dividiert werden muss.

Ein weiteres Problem liegt darin, dass Jobs mit großer Ausführungszeit durch später in der Schlange eintreffende kurze Jobs verzögert werden. Dies kann dazu führen, dass solch ein Job nie zur Ausführung kommt (*Starvation*).

Aus diesem Grund eignet sich Shortest-Job-First hauptsächlich für Stapelverarbeitungssysteme, bei denen alle Jobs schon zu Beginn bekannt sind.

4.4.3.4 Garantiertes Scheduling

Bei diesem Algorithmus wird dem Benutzer ein Versprechen über die Performance des Systems gemacht und es wird versucht, diesem Versprechen gerecht zu werden: Ein einfache einzuhaltenes Versprechen ist: Bei n Benutzern erhält jeder $1/n$ der Prozessorzeit.

Das System muss nur überwachen, wieviel Rechenzeit ein Benutzer seit Beginn seiner Sitzung verbraucht hat und wie lange seine Sitzung bereits dauert. Anhand des Verhältnisses aus verbrauchter Prozessorzeit und zustehender Prozessorzeit kann entschieden werden, welcher Prozess zur Ausführung kommt. Es wird der Prozess mit dem kleinsten Verhältnis so lange ausgeführt, bis das Verhältnis über das des nächsten Konkurrenten angestiegen ist.

Für Echtzeitsysteme kann dieser Algorithmus folgendermaßen angepaßt werden: Um zu gewährleisten, dass die absoluten Zeitschranken eingehalten werden, wird der Prozess als erster ausgeführt, bei dem die Gefahr am größten ist, dass die Zeitschranke nicht eingehalten werden kann. Ein Prozess, welcher in 10 Sekunden beendet sein muss, erhält eine höhere Priorität als ein Prozess, der erst in 10 Minuten beendet sein muss, dies wird auch als *Earliest Deadline First* bezeichnet.

4.4.3.5 Trennung von Strategie und Mechanismus

Bei vielen Anwendungen kommt es vor, dass nicht alle Prozesse eines Systems um den Prozessor konkurrieren, sondern dass ein Prozess viele Kinderprozesse unter seiner Kontrolle hat. In einem solchen Fall weiß der Hauptprozess, welcher der Kinderprozesse im Augenblick der wichtigste oder zeitkritischste ist und welcher der unwichtigste. Die bisherigen Scheduler sehen es nicht vor, Eingabedaten von Benutzerprozessen bezüglich der Scheduling-Entscheidungen anzunehmen. Hieraus resultiert, dass der Scheduler nicht die optimalen Entscheidungen treffen kann.

Ein Ansatz um dieses Problem zu lösen, ist die Trennung von *Scheduling-Mechanismus* von der *Scheduling-Strategie*. Hierbei werden die Scheduling-Algorithmen geeignet parametrisiert, damit die Parameter vom Benutzerprozess gesetzt werden können. Der Betriebssystemkern stellt einen Systemaufruf bereit, mit dem ein Prozess die Prioritäten seiner Kinderprozesse setzen kann, damit ist der Vaterprozess in der Lage, das Scheduling seiner Kinderprozesse zu beeinflussen, ohne dass er selbst das Scheduling durchführt. In diesem Fall ist der Mechanismus ein Teil des Kerns, aber die Strategie wird durch einen Benutzerprozess vorgegeben.

Welcher Algorithmus der geeignete ist, muss von Fall zu Fall entschieden werden. Neben den hier aufgeführten gibt es noch eine Reihe weiterer Algorithmen, die sich durch spezielle Eigenschaften auszeichnen und somit für bestimmte Aufgaben besonders geeignet sind.

4.4.4 Speicherverwaltung

Neben der verfügbaren CPU-Rechenzeit ist der Speicher eine weitere Komponente die nicht in unbegrenztem Umfang zur Verfügung steht. Gründe hierfür sind zum einen die Kosten (selbst bei fallenden Speicherpreisen) und zum anderen Kriterien die durch die Umwelt des Systems vorgegeben sind (z.B. Größe, Energieverbrauch oder Gewichtsbegrenzungen).

Es ist also wichtig, es von Seiten des Betriebssystems zu ermöglichen, den vorhanden Speicher möglichst effektiv zu nutzen. Außerdem ist notwendig, dem Compiler Anweisungen geben zu können, an welcher Stelle im Speicher Daten abgelegt werden sollen, da es unterschiedliche Speichertypen mit unterschiedlicher Zugriffscharakteristik gibt. Hierdurch ist es möglich, die Performance sowie die Vorhersagbarkeit eines Programms zu erhöhen und zugleich die Kommunikation mit der Umwelt sicherzustellen.

Der Teil des Betriebssystems, welcher diese Aufgabe übernimmt, heißt *Speicherverwaltung*. Seine Aufgaben umfassen die folgenden Punkte:

- Verwaltung von freiem und belegtem Speicher.
- Zuweisung von Speicherbereichen an Prozesse, wenn sie diesen benötigen. Freigabe, wenn sie abgeschlossen sind.
- Durchführung von Auslagerungen zwischen dem Hauptspeicher und der Platte, falls der Hauptspeicher nicht groß genug ist, um alle Prozesse auf einmal zu halten.

4.4.5 Systemaufrufe

Die Kommunikation zwischen Benutzerprogrammen und dem Betriebssystem geschieht über Systemaufrufe, mittels dieser fordert das Programm vom Betriebssystem die gewünschten Dienste an. Jeder Systemaufruf entspricht eine Bibliotheksprozedur. Diese schreibt die Parameter des Systemaufrufs an eine definierte Stelle (z.B ein Prozessorregister) und führt dann einen TRAP-Befehl aus, um das Betriebssystem zu starten, d.h. in den Supervisormode zu wechseln. Mit der Bibliotheksprozedur werden die Details eines TRAP-Befehls versteckt und der Systemaufruf bekommt das Aussehen eines gewöhnlichen Prozeduraufrufs.

Nachdem die Kontrolle an das Betriebssystem übergegangen ist, werden die Parameter auf Gültigkeit geprüft und die geforderte Operation wird ausgeführt. Nach Ausführung wird vom Betriebsystem eine Statuscode in ein Register geschrieben, welcher über Erfolg oder Misserfolg der Operation Auskunft gibt. Mittels eines RETURN FROM TRAP-Befehls wird die Kontrolle wieder an die Bibliotheksprozedur zurückgegeben. Diese kehrt dann zum Aufrufer zurück und

übergibt den Statuscode in Form eines Funktionswertes, eventuell mit zusätzlichen Werten in den Parametern.

Bis hier wurde beschrieben, wie sich Betriebsysteme nach außen darstellen (d.h. die Schnittstelle zum Programmierer). Um aber weitere Aussagen bezüglich dieser Systeme machen zu können, ist es notwendig, die innere Organisation zu untersuchen. Hier wird der Schwerpunkt auf das analysierte Betriebsystem Linux gelegt.

4.5 Struktur von Betriebssystemen

Für die später folgenden Betrachtungen sind zwei unterschiedliche Strukturen von Interesse. Zum einen monolithische Systeme und zum anderen Client-Server Modelle (auch als Microkernel bezeichnet).

4.5.1 Monolitische Systeme

Unter diese Kategorie fällt auch das betrachtete Linux. [147] bezeichnet diese Organisationsform als "Die große Masse". Das Betriebssystem ist als eine Menge von Prozeduren geschrieben, von denen jede Prozedur eine andere aufrufen kann, wenn sie diese benötigt. Jede Prozedur hat eine definierte Schnittstelle in Form von Parametern und Ergebnissen. Alle im System vorhandenen Prozeduren sind global im System sichtbar. Im Gegensatz zu einer Struktur, die aus Modulen besteht und in der große Teile der Informationen nur lokal zu Modulen sind und es nur offizielle Einstiegspunkte von außen gibt.

Eine Struktur wird in monolithische Systeme durch die Benutzung der beschriebenen Privilegienebenenebenen des Prozessors erreicht (vergl. Seite 24): Alle Systemaufrufe (siehe Absch. 4.4.5), welche das Betriebssystem bereitstellt, legen ihre Parameter an wohldefinierter Stelle - den Registern oder dem Stack - ab und führen dann einen TRAP-Befehl aus, welcher auch als *Kernelaufruf* oder *Supervisor Call* bezeichnet wird.

Durch diesen Befehl wird der Prozessor aus dem *Usermode* in den *Supervisormode* umgeschaltet und die Kontrolle geht an den Kernel (siehe (1) in Abb. 4.7). Im für das Betriebssystem vorgesehenen Kernmodus sind alle Befehle erlaubt. Das Betriebsystem ermittelt über einen Index ((2) in Abb. 4.7) welcher Systemaufruf auszuführen ist. Die Operation (3) in Abb. 4.7 ist schließlich die auszuführende Dienstprozedur. Ist deren Aufruf beendet, wird die Kontrolle an das Benutzerprogramm zurückgegeben.

Dies führt zu einer Struktur mit prinzipiell drei Schichten:

1. Benutzerprogramme

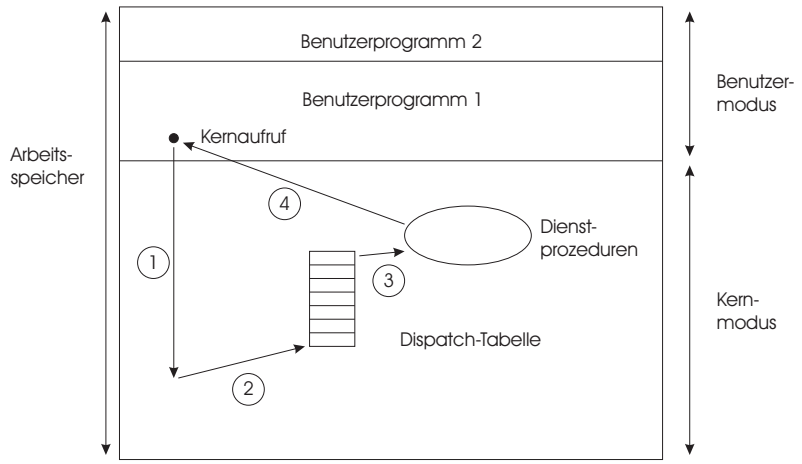


Abbildung 4.7: Systemaufrufe in einem monolithischen Kernel [147].

2. Dienstprogramme

3. Hilfsprozeduren

Forschungen im Bereich Betriebssysteme widmen sich zunehmend so genannten Microkernels.

4.5.2 Microkernel

Bei Microkernel basierten Betriebssystemen wurde ein Großteil des Codes aus dem Betriebssystem in höhere Schichten ausgelagert, so dass ein minimaler Kern übrig bleibt. Die im Kern verbleibenden Funktionalität ist nur die Kommunikation. Alle anderen Funktionen des Betriebssystems sind als Benutzerprogramme realisiert (Abb. 4.8, obere Ebene). Benutzerprozesse erhalten einen Dienst durch Senden von Nachrichten an den Serverprozess. Denkbar ist diese Architektur auch für die Verwendung in verteilten Systemen. Ob ein Dienst lokal auf dem System verfügbar ist oder ob er auf einem entfernten System ausgeführt wird, ist für den Klienten nicht interessant [128]. Ein Microkernel erzwingt beim Entwickler einen modularisierten Ansatz, da alle Ebenen des Systems sehr eigenständige Programme sind, die mit den anderen Teilen des Systems über streng definierte Schnittstellen kommunizieren.

Bei Microkernel-basierten Systemen wird der Entwickler gezwungen, einen modularisierten Ansatz zu verwenden, da es sich bei den als Benutzerprogrammen realisierten Diensten um relativ unabhängige Anwendungen handelt, die nur über eine sehr genau definierte Schnittstelle angesprochen werden können.

Ein weiterer Vorteil dieser Architektur ist die gute Portierbarkeit auf andere

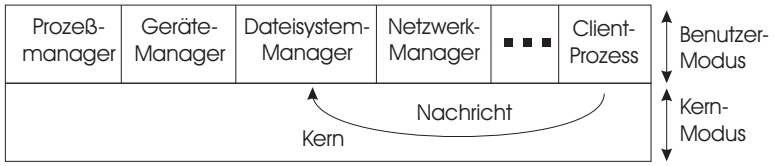


Abbildung 4.8: Dienstanforderung in einem Mikrokernel basierten Bestriebssystem

Hardwareplattformen, da alle hardware-abhängigen Komponenten als gekapselte Einheiten realisiert sind. Bei der Speichernutzung können diese Systeme ebenfalls Vorteile gegenüber monolithischen Systemen haben, da nicht benötigte Funktionen ausgelagert oder vollständig entfernt werden können. Warum monolitische Systeme und hier ganz besonders auf Linux basierende Systeme ihre Berechtigung haben, wird in Abschnitt 7 verdeutlicht.

4.6 Zusammenfassung

Zum Anfang dieses Kapitels wurden Betriebssysteme unter zwei Sichtweisen betrachtet: Als Betriebsmittelverwalter und als erweiterte Maschine. Die erste Sichtweise beinhaltet die Aufgabe die verschiedenen Teile des Systems effizient zu verwalten; die zweite Sichtweise, dem Benutzer eine einfach zu benutzende virtuelle Maschine bereitzustellen, welche leichter zu handhaben ist, als das reelle System. Hiermit wird zusätzlich ein sicherer Betrieb des System gewährleistet.

Anschließend wurden ein wesentlichen Konzept von Betriebsystemen erläutert: die Prozesse. Betriebssysteme benutzen ein Modell, welches aus sequenziellen Prozessen besteht, die parallel ausgeführt werden. Jeder Prozess besitzt einen Zustand und kann als virtueller Prozessor angesehen werden.

Die Kommunikation und das Synchronisieren zwischen Prozessen kann durch verschiedene Mechanismen erreicht werden: Im Kern-Modus kommen die Nichtunterbrechbarkeit, das Sperren der Interrupts, Kernel-Semaphoren und Spin-Locks zum Einsatz. Für die Kommunikation im Benutzer-Modus existieren Pipes, FIFOs (named Pipes), IPC-Semaphoren und Shared Memory als Kommunikationskanäle.

Der Scheduler hat die Aufgabe, zu bestimmen, welcher Prozess als nächster ausgeführt werden soll. Hierfür gibt es verschiedene Algortithmen mit speziellen Eigenschaften, wie z.B. Antwortzeit, Effizienz und Fairness.

Abschließend wurden die zwei für die spätere Betrachtung wichtigen Strukturen von Betriebssystemen vorgestellt: ein Monolithischer- oder ein Micro-Kernel.

Kapitel 5

Entwicklungswerkszeuge und Standarisierung

5.1 Motivation

Die ersten Real time Systeme waren in Assembler programmiert. Der Grund hierfür liegt darin, dass Hochsprachen für die meisten Computersysteme noch nicht verfügbar waren und so die Assembler Sprache der einzige Weg war, effizient auf die Hardware-Ressourcen zuzugreifen. Das Problem beim Assembler liegt darin, dass eine größere Nähe zur Hardware als zum Problem besteht. Bei der Entwicklung müssen Details betrachtet werden, die wenig mit dem eigentlichen Algorithmus zu tun haben. Aus diesem Grund wurden Hochsprachen entwickelt und eingeführt.

Für die Programmierung von Echtzeit-Applikationen unter Linux kommt als Hochsprache nur *C* oder *C++* in Frage - dies als Vorgriff: Bei der Entwicklung von Kernelmodulen gibt es keine Alternative. Für Applikationen im User-Space bieten alle echtzeitfähigen Varianten nur APIs für die Entwicklung von C/C++ Programmen.

Deshalb wird im ersten Teil dieses Kapitels ein Überblick über die Entwicklungswerkzeuge eines GNU-Systems gegeben.

Ein weiteres Thema dieses Kapitel ist der POSIX-Standard. Dieser Standard wird im zweiten Teil vorgestellt, da er für den Vergleich der Systeme an einigen Stellen von Bedeutung ist und so an späterer Stelle einfach auf diese Erläuterungen zurückgegriffen werden kann.

5.2 Die GNU-Werkzeuge

Die vom GNU-Projekt bereitgestellten Entwicklungswerkzeuge umfassen die folgenden Komponenten [108]:

- Compiler
- Linker
- Debugger
- Assmebler
- Tools zur Erstellung portabler Konfigurationsdateien
- Diagnosetools

Diese Programme ermöglichen die Entwicklung für Systeme angefangen beim 8-bit Microkontroller und Digitalen Signal Prozessoren - auf denen aufgrund der zu geringen Ressourcen kein Linux läuft - bis hin zu Großrechnern.

Es fällt auf, dass in der Liste keine integrierte Entwicklungsumgebung vorhanden ist, wie dies bei anderen Entwicklungswerkzeugen üblich ist. Hier kommt wieder das Unix-Konzept zu Tage: viele kleine Tools, die alle eine spezielle Aufgabe erfüllen.

Es existieren auch unter Linux leistungsfähige integrierte Entwicklungsumgebungen, nur sind sie nicht Teil des GNU-Werkzeugkastens (vergl. Absch. 5.3.2).

5.3 Entwicklungsumgebungen

Auch bei den Entwicklungsumgebungen sieht es so aus, wie an vielen Stellen im Linux-Umfeld: Der Benutzer hat die Wahl. Prinzipiell lassen sich zwei Wege unterscheiden:

5.3.1 Die Kombination aus Editor und Komandozeile

Diese klassische Methode besteht daraus, den Code mit dem Editor seiner Wahl zu erstellen und anschliessend von der Komandozeile die Befehle zum Compilieren, Linken und Starten des Programms zu geben. Vereinfacht wird dies durch die Benutzung von*makefiles* in denen Umgebungsvariablen definiert und die Objektdateien eines Projekts in Kombination mit Compileroptionen angegeben werden können. Viel mehr Komfort ist hier nicht möglich.

5.3.2 Integrierte Entwicklungsumgebungen

Den vollen Komfort bieten hier integrierte Entwicklungsumgebungen, wie z.B. die freie Variante *KDevelop* [103] oder das kommerzielle properitäre Produkt *Code Warrior*. Diese Umgebungen bieten eine Reihe von nützlichen Funktionen:

- Quellcode-Editor mit Syntax-Highlighting.
- Integrierte Werkzeuge zur Erstellung der Dokumentation direkt aus den Quelltexten.
- Compilieren, Starten und Debuggen eines Programmes auf Knopfdruck.
- Verwenden eines Versionskontrollsystems zum Bearbeiten großer Projekte durch Arbeitsgruppen.
- Hilfestellung bei der Erstellung von Projekten (s.g. *wizards*).

Die Entwicklungsumgebung sorgt während der Entwicklung dann für den Aufruf der GNU-Werkzeuge.

Für welche der beiden Wege man sich entscheidet, ist zum einen persönlicher Geschmack, zum anderen durch die Rahmenbedingungen des Projekts vorgegeben. Der Abschnitt 8.2.3.3 beschäftigt sich eingehender mit diesem Aspekt.

Es schließt sich der zweite Teil dieses Kapitels mit einer kurzen Erläuterung zum POSIX-Standard an.

5.4 POSIX

POSIX, das "**P**ortable **O**perating **S**ystem **I**nterface", ist ein Dokument, welches von der IEEE erarbeitet und von ANSI und ISO standarisiert wurde. Aufgabe des POSIX-Standards ist die Portabilität von Applikationen auf Ebene des Source Codes. Es werden also eindeutige Schnittstellen in Form von Funktionen definiert, die ein Betriebssystem bereitstellen muss, um diesen Standards zu genügen. Hierdurch wird gewährleistet, dass ein POSIX konform geschriebener Code auf allen POSIX Betriebsystemen kompilierbar ist.

Die aktuellen Standards für die System-APIs beinhalten die folgenden Klassifizierungen [109]:

5.4.1 IEEE 1003.1 - System API

Dieser Standard beschreibt die Basis-Konzepte für UNIX-artige Betriebsysteme. Eingeschlossen sind: Prozessdefinition, File-System, Basis I/O-Funktionen und die grundlegende APIs.

5.4.2 IEEE 1003.1b - Realtime & I/O Erweiterungen

Früher als POSIX.4 bezeichnet - ist POSIX.1b auf die Interprozess-Kommunikation und Synchronisation fokusiert. Es werden Queues, Semaphoren und priorisierte Signale behandelt.

Linux erfüllt diesen Standard nicht (ausgenommen mmap(). Linux implementiert die weiterverbreiteten SVR4 Versionen der selben Mechanismen.

5.4.3 IEEE 1003.1c Threads

Ursprünglich als POSIX.4a bekannt, spezifiziert POSIX.1c die APIs und Semantik für POSIX Threads (pthreads). Hier herrschen zum Teil Verflechtungen mit POSIX.1b.

Linux implementiert Pthreads als eine Standard-Benutzer-Bibliothek, welche die Threads auf das native Linux Kernel Thread-Interface abbildet.

5.4.4 IEEE 1003.13 - Minimal Realtime System Profile

Da der POSIX-Standard ein sehr umfangreiches Interface festlegt, wurde mit POSIX 1003.13 ein so genanntes minimales Echtzeit-Profil definiert, welches nur einen kleineren Teilsatz der Systemdienste fordert und es somit ermöglicht, kleine und effiziente Kernel mit Echtzeiteigenschaften zu entwickeln.

5.5 Zusammenfassung

Für Linux gibt es einen leistungsfähigen freien Werkzeugsatz zur Softwareentwicklung, die vom GNU-Projekt verwaltet werden. Auch integrierte Entwicklungsumgebungen sind vorhanden. Wie diese in der Entwicklung eingesetzt werden können, wird in der abschließenden Analyse beurteilt.

Der POSIX 1003.1b Standard (vor 1993 als POSIX.4 bezeichnet) erlaubt die Verwendung eines Betriebsystems in weichen Echtzeit-Situationen. Standard Linux und die echtzeitfähigen Linuxvarianten werden auf ihre POSIX-Konformität überprüft, um herauszuarbeiten, inwieweit bestehende POSIX Software unter Linux portierbar ist.

Kapitel 6

Anforderungen an Realtime-Systeme in der Automatisierungstechnik

6.1 Motivation

Ob ein echtzeitfähiges Betriebssystem für eine bestimmte Aufgabe geeignet ist, kann nur anhand mehrerer Kriterien beurteilt werden. Aufgabe dieses Kapitels ist es, diese Kriterien zu formulieren und anschließend eine Zusammenstellung von Kriterien zu machen, anhand derer ein Vergleich der Varianten möglich ist.

Ein Teil dieser Kriterien resultiert aus den bereits in Kapitel 2 beschriebenen Anforderungen.

Es ergeben sich vier Hauptkriterien: Kosten, Sicherheit, Flexibilität und Performance (siehe Abb. 6.1). Die Reihenfolge spiegelt auch zugleich die Gewichtung der einzelnen Bereiche in der vorliegenden Arbeit wider. In dieser Arbeit liegt der Schwerpunkt nicht auf der Performance der einzelnen Systeme, sondern eindeutig auf den Besonderheiten, die durch das Modell der Freien Software bedingt sind.

Diese Kriterien werden im einzelnen genauer formuliert. In jedem Bereichen gibt es Unterpunkte, die für das Thema "Linux in der Automatisierungstechnik" besonders interessant sind.

Anschließend wird ein Bewertungsschema formuliert, anhand derer im Kapitel 8 eine Beurteilung der Eignung eines Systems für ein spezielles Problem - die Lösung einer speziellen Automatisierungsaufgabe - stattfindet.

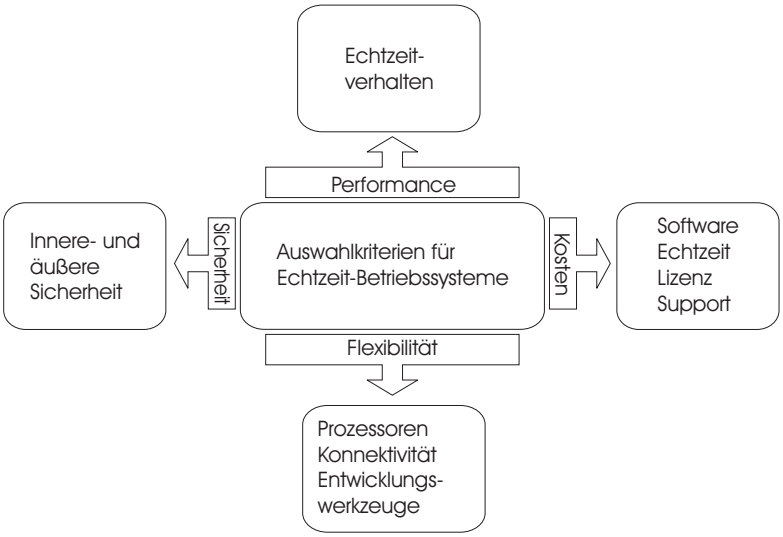


Abbildung 6.1: Die Hauptkriterien für die Auswahl eines Echtzeitbetriebssystems.

6.1.1 Kosten

Alle ökonomischen Konsequenzen, die beim Einsatz eines bestimmten Betriebssystems entstehen, sind in diese Kategorie eingeordnet. Dies können Kosten sein, die direkt durch den Erwerb von Lizenzen entstehen, durch die Notwendigkeit, eine bestimmte Hardware einsetzen zu müssen oder durch die indirekten Kosten für technische Unterstützung. Somit lässt sich der Bereich Kosten in zwei Bereiche einteilen: einen Bereich der sich mit den Kosten beschäftigt, welche durch technische Notwendigkeiten des Betriebsystems entstehen und einem Bereich, der sich mit den Kosten beschäftigt, welche durch Lizenzgebühren und Support entstehen.

6.1.1.1 Kosten für die Software

Bei der Auswahl eines Betriebssystems ist ein entscheidende Faktor immer der Preis, der für das System zu zahlen ist. An erster Stelle stehen hier normalerweise die Lizenzkosten - im Linuxumfeld in der Regel nicht vorhanden. In der Automatisierungstechnik kommt aber als zweiter Punkt hinzu, wie die Kosten sich über der Einsatzzeit einer Anlage verhalten. Das Stichwort Investitionsschutz ist hier von entscheidender Bedeutung.

Da es bei den echtzeitfähigen Linuxvarianten nicht nur um Freie Software handelt, sondern auch kommerzielle Versionen existieren, werden die Konsequenzen hieraus ebenfalls ein Kriterium sein.

Weiterhin spielen in diesem Kontext auch Spoftwarepatente eine entscheidende

Rolle. Auf diese wird bei der Analyse ein weiterer Augenmerk gelegt.

6.1.1.2 Kosten für die Echtzeit

Bedingt durch das ausgewählte Betriebssystem, entstehen Kosten für die benötigte Architektur. Wie verhält sich Linux in diesem Bereich, wie sehen die Minimalanforderungen für ein System aus?

6.1.2 Flexibilität

In diese Kategorie fallen drei Bereiche: Anzahl der unterstützten Prozessoren, Konnektivität und Entwicklungswerkzeuge.

6.1.2.1 Prozessorunterstützung

Zum ersten ist es wichtig, welche Prozessor-Familien vom Betriebssystem unterstützt werden. Oft lässt es sich nicht vermeiden, eine Applikation auch auf einer anderen Plattform einzusetzen. Eine breitere Basis der unterstützten Prozessoren schränkt deutlich weniger ein. Weiterhin ist hierdurch die Verbesserung der Skalierbarkeit gewährleistet. Bestimmte Prozessorfamilien decken ein breites Leistungsspektrum ab.

6.1.2.2 Konnektivität

Aus den in Abbildung 2.6 bestehenden Verbindungen zur Anlage über Hardwareinterfaces, zu externen Systemen (z.B. Datenbanken) und der Fernwartung wird deutlich, dass das Betriebssystem zum einen die notwendigen Hardwarekomponenten unterstützen muss und zum anderen auch die Kommunikationsprotokolle bereitstellen muss, damit eine rationelle Integration der Funktionalität möglich ist.

6.1.2.3 Entwicklungswerkzeuge

Für den Entwickler von echtzeitfähigen Systemen ist es wichtig, dass er durch einen funktionellen und ausgereiften Satz an Werkzeugen und Methoden unterstützt wird. Es ist ein Augenmerk bei der Untersuchung darauf zu richten, in wie weit die Systeme hier Unterstützung bieten.

6.1.3 Sicherheit

Für Echtzeitsystemen in der Automatisierungstechnik, insbesondere in sicherheitskritischen Systemen, gibt es zwei Bereiche der Sicherheit, die interessant sind. Diese resultieren direkt aus Abschnitt 2.4.3.

6.1.3.1 Innere Sicherheit

Unter diesen Aspekt fallen die Mechanismen, die das Betriebssystem bietet, um zu verhindern, dass z.B. unautorisierte Speicherzugriffe auf den Systembereich durchgeführt werden können (vgl. Absch. 4.4.4).

6.1.3.2 Äußere Sicherheit

Die Notwendigkeit, Systeme an das firmeninterne Intranet anzuschließen und damit von der Leitungsebene direkt und weltweit auf Daten aus der Produktion zugreifen zu können, hat weit reichende Folgen. Angefangen damit, nur bestimmten Personen oder Rechnern den Zugriff zu erlauben bis zu verschlüsselten Übertragung der Daten.

Weiterhin entstehen durch den Ruf nach Fernwartbarkeit und Fernadministrierbarkeit Probleme, dies auf gesicherte Weise zu ermöglichen. Schließlich sind gerade die automatisierten Anlagen ein unternehmens- und sicherheitskritischer Bereich, den es unbedingt zu schützen gilt:

> "We are using the efficiencies of technology and the Information Age to control everyday things like traffic lights, 911 systems, the environment of buildings, the communication network, and the power grid. We even control the water supply with computers." [50]

Diese Aussage von James Kallstrom unterstreicht, wie wichtig die Sicherheit solcher Systeme ist.

Welche Methoden, die Sicherheit eines Systems zu gewährleisten bietet das Betriebssystem?

6.1.4 Performance

Kriterien dieser Gruppe werden typischerweise in Zeiteinheiten gemessen und können mit der Performance anderer Betriebssysteme verglichen werden. In dieser Arbeit liegt der Schwerpunkt nicht auf der Ermittlung dieser Zahlen, vielmehr wird eine allgemeine Analyse der Systeme im Hinblick auf die realisierte Architektur und der daraus resultierenden Leistungsfähigkeit - d.h. der Art der

Echtzeitfähigkeit - gelegt. Hieraus lässt sich bereits sehr gut ableiten, für welchen Einsatzzweck die Varianten geeignet sind.

6.2 Die Vergleichskriterien

Um bei der späteren Analyse die Varianten vergleichen zu können, wird, basierend auf den bisher herausgearbeiteten Punkten, ein Bewertungsschema formuliert. Beim Vergleich der Varianten wird für jedes die Erfüllung der einzelnen Kriterien in ein Netzdiagramm eingetragen. Hierbei steht ein Wert von "6" für die vollständige Erfüllung eines Kriteriums und der Wert "0" für die Nicht-Erfüllung.

Mit Hilfe dieser Diagramme ist dann der Vergleich der Systeme untereinander möglich. Weiterhin können diese Diagramme dazu benutzt werden, die Eignung einer Variante für ein bestimmtes Aufgabe zu untersuchen. Dies geschieht exemplarisch für das bereits in der Einleitung erwähnte Prüfstand-System.

Für die vier Bereiche Kosten, Sicherheit, Flexibilität und Performance werden die Kriterien wie folgt bewertet:

6.2.1 Kosten

Open Source Hier bei wird die Frage gestellt, in wie weit die Variante ein Projekt ist, welches dem *Open Source Gedanken* folgt. Ein Wert von "6" bedeutet, das Projekt folgt zu 100% dem Gedanken und steht voll unter der GPL bzw. LGPL. Ein Wert von "0" beschreibt eine vollständig proprietäre Variante.

Community Projekt Dieses Kriterium widmet sich der Organisationsform der Variante. Handelt es sich um eine Gemeinschaft mit vielen gleichberechtigten Partnern "6" oder gibt es eine zentrale Stelle, die alleine festlegt, in welche Richtung die Variante weiterentwickelt wird "0".

Kommerzieller Support In welchem Masse ist kommerzieller Support durch den Hersteller oder durch dritte vorhanden und kann genutzt werden? Hierbei bedeutet ein Wert von "6", das sowohl kommerzieller als auch Support durch dritte vorhanden ist. Der Wert "0" bedeutet, es ist kein Support vorhanden.

6.2.2 Sicherheit

innere Sicherheit Wie gut unterstützt die Variante diesen Aspekt der Sicherheit. Ein Wert von "6" bedeutet eine sehr hohen Sicherheitslevel.

äußere Sicherheit Werden neuere Kernel-Versionen unterstützt? Diese bieten in der Regel nicht nur neue Funktionen, sondern beseitigen auch bestehende Sicherheitslücken. Standard Programme im User Space werden hier nicht betrachtet, da diese weitgehend unabhängig vom Kernel ausgetauscht werden können und auch nicht an die Echtzeitfähigkeit gebunden sind.

6.2.3 Flexibilität

Skalierbarkeit Wie gut skaliert die Variante? Hierbei ist besonders die Skalierung im Hinblick auf ein schlankes System von interesse. Ein Wert von "6" bedeutet ein System, welches sehr gut skalierbar ist.

Hardware Konnektivität Wie gut können bestehende Treiber in der Variante benutzt werden und wie breitgefächert ist die Treiberunterstützung generell? Ein Wert von "6" bedeutet hier, dass für alle gängige Hardware ein Treiber vorhanden ist.

Entwicklungswerkzeuge Wie gut wird die Variante durch die vorhandenen Entwicklungswerkzeuge unterstütz? Gibt es eventuell noch besonders abgestimmte Werkzeuge?

Modell für Softwareentwicklung Welche Anforderungen stellt die Variante an die Softwareentwicklung. Ist die Entwicklung mit besonderem Aufwand verbunden (Wert "0")?

6.2.4 Performance

Echtzeiteigenschaft In welche Kategorie der Echtzeitfähigkeit gehört die Variante? Hierbei steht "6" für harte Echtzeit und "3" für weiche. Ein Wert von "0" bedeutet, die Variante ist nicht für den Einsatz im Echtzeitbereich geeignet.

6.3 Zusammenfassung

In diesem Kapitel wurden die im Rahmen dieser Arbeit interessanten Aspekte bezüglich des Einsatzes von echtzeitfähigen Linuxvarianten in der Automatisierungstechnik formuliert und ein kleiner Satz an Kriterien zusammengestellt, mit dem es möglich ist, einen Vergleich der Varianten untereinander, sowie die Einschätzung der Eignung einer Variante für ein bestimmtes Projekt durchzuführen. Wieweit diese Kriterien von den Linuxvarianten erfüllt werden, wird in Kapitel 8 analysiert. Vorrangig werden dort Kriterien betrachtet, bei denen es signifikante Unterschiede zu proprietären Systemen gibt.

Im Anschluss werden der Standard Linux Kernel und die echtzeitfähigen Varianten vorgestellt.

Kapitel 7

Linux und echtzeitfähige Linuxvarianten

7.1 Motivation

In diesem Abschnitt wird der Schwerpunkt auf die Strukturen der untersuchten Betriebssysteme gelegt. Zuerst wird der unmodifizierte "Standard"-Kernel betrachtet. Ausgehend hiervon werden die verschiedenen Möglichkeiten herausgearbeitet, wie man Echtzeitfähigkeit erreichen kann. Abschließend werden realisierte Varianten vorgestellt und bezüglich ihres Design analysiert.

7.2 Linux im Überblick

7.2.1 Systemübersicht

Der Linux Kernel isoliert betrachtet ist nutzlos; er ist Teil eines größeren Systems, welches als Ganzes nützlich ist. Abbildung 7.1 zeigt die Zerlegung eines GNU/Linux Systems in die wichtigsten Teilsysteme.

Anwenderprogramme	z.B. Datenbanken
Betriebssystemdienste	Dienste als Teil des Betriebssystems z.B. Window-Manager, Login-Shell
Linux Kernel	Der für diese Betrachtungen interessante Teil.
Hardwarecontroller	Alle physikalischen Komponenten einer Installation, z.B CPU, Speicher, Festplatten

Abbildung 7.1: Zerlegung eines Linuxsystems in die wichtigsten Subsysteme [19].

```
sys_read:

    send request to disk
    sleep until data is ready at disk
    read block from disk
    read block from disk
    read block from disk
    send other request to disk
    sleep until data is ready at disk
    read block from disk
    read block from disk
    read block from disk
    send other request to disk
    ...
```

Abbildung 7.2: Struktur eines Read Systemaufrufs.

Die Zerlegung folgt dem Schichtenmodell, welches in [51] diskutiert wird. Jedes Teilsystem kommuniziert nur mit angrenzenden Teilsystemen. Der Kernel für sich besteht wiederum aus weiteren Teilsystemen, die aber nach außen unsichtbar sind.

7.2.2 Organisation des Kernels

Linus Torvalds entwickelte die erste Version des Betriebssystems auf einem 386er System. Aus diesem Grunde entschied er sich auch für eine monolithische Struktur des Systems. Da, bedingt durch die Kommunikation zwischen den verschiedenen Ebenen eines Microkernel basierten Systems, die Performance zu schlecht gewesen wäre.

Auch wenn Linux für den Benutzer als präemptives Betriebssystem erscheint, ist es von der Kernel Architektur her nicht präemptiv, da Systemfunktionen vom Scheduler nicht unterbrechbar sind.

Hieraus resultieren Blockierzeiten, die im Bereich von zehntel Mikrosekunden - z.B. Ändern der Priorität eines Prozesses - bis in den Bereich von Sekunden gehen - z.B das Lesen eines großen Datenblocks von Festplatte. Hieraus resultieren unter Umständen relativ lange Antwortzeiten. Aus diesem Grund sind viele Operationen bereits in kleiner Blöcke aufgeteilt, um diese Zeit zu reduzieren (auf ca. 10 bis 30 Millisekunden). Die Abbildung 7.2 verdeutlicht dies.

Weiterhin existieren Bereiche, in denen Interrupts gesperrt sind, um eine Integrität des Systems zu gewährleisten.

Der Kernel ist für sich gesehen kein Prozess sondern ein Prozessmanager. Wenn ein Benutzerprozess auf Kernelressourcen zugreifen will, führt dieser einen *Systemaufruf* aus. Jeder Systemaufruf übergibt einen Satz an Parametern und wechselt anschließend vom Benutzer in den Kernelmodus (siehe auch Absch. 4.4.5).

In einem monolithischen System existieren im Kernel Treiber, Netzwerkfunktionen, Speichermanagement, Dateisysteme und einiges mehr nebeneinander. Hieraus resultiert zwar ein Geschwindigkeitsgewinn, allerdings auf Kosten der Wartbarkeit. Schon bei kleinen Änderungen muss der komplette Kernel neu compiliert werden.

An dieser Stelle wurde der Kernel um ein Modulkonzept erweitert. Module sind ein Kernelfeature, welches viele Vorteile eines Mikrokernels erreicht, ohne das die Performance leidet. Ein Modul ist ein Objekt, dessen Code einfach in den Kernel eingebunden werden kann und bei Nicht-Benutzung genauso einfach wieder entfernt werden kann. Ein Modul wird aber nicht - im Gegensatz zum Mikrokernel - im Benutzermodus ausgeführt, sondern im Kernelmodus.

7.2.2.1 Prozessmodell

Zur Laufzeit besteht Linux nur aus Prozessen, d.h. Programmen, die sich in der Ausführung befinden. Aufgabe ist es, diese Prozesse zu verwalten, das heißt Rechenzeit und Kommunikationsmechanismen bereitzustellen und zu vergeben. Zur Verwaltung müssen desweiteren Dienste zur Erzeugung und Zerstörung der Prozesse bereitgestellt werden. Das kann auf einer oder auf mehreren CPUs stattfinden. Das Betriebssystem bildet diesen Vorgang auf die zugrundeliegende Hardware ab und stellt eine einheitliche Schnittstelle nach "oben" zum Benutzer bereit. Teil der Prozessvewaltung ist der Scheduler, der eine zentrale Rolle im System spielt.

7.2.2.2 Speicherverwaltung

Jeder Prozess läuft in seinem eigenen privaten Adreßraum. Ein im Benutzermodus laufender Prozess bezieht sich auf seinen eigenen Stack sowie Daten- und Code-Bereich. Im Kernel-Modus bezieht sich ein Prozess auf den Kernel Daten- und Code-Bereich und benutzt einen anderen Stack.

Die Schwierigkeit bei der Speicherverwaltung besteht in der sparsamen Verteilung der teuren Betriebsmittel. Je seltener Speicher auf die Festplatte ausgelagert werden muss, desto weniger Zeit nimmt die Verwaltung in Anspruch. Algorithmen hierzu finden sich in [147]. In Linux wurde das Konzept des *Demand Paging* implementiert - im Gegensatz zu traditionellen Auslagerungsverfahren wie dem Swapping, bei dem der ganze Prozess aus dem RAM auf die Festplatte ausgelagert wurde.

Beim *Demand Paging* ist der gesamte Hauptspeicher in Seiten unterteilt, die auf Anforderung ein- bzw. ausgelagert werden können. Speicherseiten des Kernelsegments dürfen nicht ausgelagert werden, da die zum Ein- und Auslagern benötigten Funktionen immer im Speicher verfügbar sein müssen [18]. Es besteht in Linux die

POSIX1003.1b-konforme Möglichkeit das Paging mittels der Funktionen `mlock()` und `munlock()` zu verbieten. Dies ist besonders für deterministisches Verhalten im Echtzeitbereich interessant.

7.2.2.3 Dateisysteme

Das Design von UNIX-artigen Betriebssystemen ist auf sein Dateisystem ausgerichtet. Vom Kernel wird ein strukturiertes Dateisystem über die unstrukturierte Hardware gelegt und die hieraus resultierende Abstraktion wird im ganzen System verwendet. Dies findet sich auch in den Gerätetreibern wieder [29].

7.2.2.4 Gerätesteuerung

Jede Systemoperation wird am Ende auf ein physikalisches Gerät abgebildet, für das spezieller Code für die Steuerung vorhanden sein muss. Ausnahmen sind hier eigentlich nur der Speicher und der Prozessor. Dieser Steuerungcode wird Gerätetreiber genannt. Für jedes Peripheriegerät muss entsprechender Code vorhanden sein.

7.2.2.5 Netzwerkbetrieb

Gerade der Betrieb im Netzwerk ist für Linux immer von besonderer Bedeutung gewesen. Der Kernel muss dies unterstützen, da die meisten Operationen nicht prozess-spezifisch sind: Eingehende Pakete sind asynchrone Ereignisse. Sammeln, Identifizieren und Weiterleiten werden vom Kernel erledigt, bevor ein Prozess die Daten verarbeiten kann. Weiterhin sind Funktionen zum Routing im Kernel integriert [145][29].

7.2.3 Interrupts

Gerade für die spätere Betrachtung der Echtzeitfähigkeit ist die Behandlung von Interrupts von zentraler Bedeutung und wird deshalb an dieser Stelle ausführlich dargestellt. Später ist es dann einfach möglich darzustellen, wie die Varianten dieses Problem behandeln.

Grundsätzlich ist ein Interrupt nichts anderes als eine Spannungsänderung auf einer speziellen Leitung, die von einem Hardwaregerät zum Prozessor bzw. zu einem Interrupt-Controller führt. Ein Interrupt dient als wichtigster Mechanismus bei der Ansteuerung von Hardware-Ressourcen. Er teilt der Software mit, das ein asynchrones Ereignis vorliegt, das von einem Interrupthandler bearbeitet werden muss.

Einem Interrupt-Handler können mehrere Interrupt-Service-Routines (*ISR*) zugeordnet werden, da sich mehrere Geräte einen IRQ teilen können. Bei Auftreten eines solchermaßen geteilten Interrupts werden alle zugehörigen Service-Routinen ausgeführt, damit die für die Behandlung der Anfrage notwendige Operationen ausgeführt werden.

7.2.3.1 Top Half und Bottom Half

Da die Interruptleitung eines gerade bearbeiteten Interrupts solange gesperrt ist, wie die ISR läuft, ist es wichtig, deren Ausführungszeit zu minimieren. Da dies nicht immer möglich ist, kann man bei Linux den Handler in zwei Teile teilen: die obere Hälfte (*top half*) und die untere Hälfte (*bottom half*). Die obere Hälfte wird direkt beim Auftreten ausgeführt. Die untere Hälfte kommt erst zu einem späteren Zeitpunkt zur Ausführung, in der der Interrupt nicht mehr gesperrt ist.

7.2.3.2 Software Interrupts

Software Interrupts wurden mit den Kerneln der Version 2.4 eingeführt. Sie ähneln den beschriebenen unteren Hälften, allerdings mit einem Unterschied: Während untere Hälften strikt nacheinander (serialized) ausgeführt werden, ist es bei Software Interrupts möglich, dass mehrere CPUs in einem System den gleichen Software Interrupt zur gleichen Zeit ausführen. Hierdurch ist z.B. bei Netzwerkoperationen ein großer Performance-Gewinn zu verzeichnen.

7.2.4 Prozesse und Threads

In vielen Betriebssystemen gibt es die Unterscheidung zwischen Prozessen und Threads.

Linux Threads sind nach dem "one-to-one" Modell implementiert worden: Jeder Thread ist ein separater Prozess im Kernel. Der Kernel Scheduler behandelt sie wie Prozesse.

Es gibt folgende Vorteile des "one-to-one" Modells:

- Minimaler Overhead für CPU-intensives Multiprozessing (mit einem Thread pro Prozessor).
- Minimaler Overhead an I/O Operationen.
- Eine einfache und robuste Implementierung (der Kernel Scheduler erledigt die gesamte Arbeit).
- Threads blockieren sich nicht gegenseitig.

Nachteilig sind das Auftreten von mehreren Kontextwechsel beim Gebrauch von Mutex und Condition Variablen, die alle aufgrund des Modells durch den Kernel bearbeitet werden. Allerdings ist dieser Zustand akzeptabel, da die Prozesswechselzeiten unter Linux sehr gering sind.

7.2.5 Systemzeit und Zeitgeber

Man kann zwischen zwei unterschiedlichen Zeitmessungen unterscheiden, die vom Linux-Kernel durchgeführt werden:

- Ermitteln der aktuellen Zeit und des Datums.
- Verwalten von Timern (Zeitgebern), die dazu dienen, den Kernel oder ein Benutzerprogramm zu benachrichtigen, wenn ein bestimmtes Zeitintervall abgelaufen ist.

Hierzu sind zwei Hardware-Uhren für den Kernel von Bedeutung:

7.2.5.1 Real Time Clock (RTC)

Die Real Time Clock wird von Linux nur dazu benutzt, die Uhrzeit und das Datum zu bestimmen.

7.2.5.2 Programmable Intervall Timer

Dieser Timer löst nach einer programmierbaren Zeitspanne einen Interrupt aus. Linux programmiert diesen Timer auf eine Frequenz von $100Hz$, d.h. alle 10 Millisekunden wird eine Interrupt ausgelöst. Das komplette Linux-System ist von diesem Rhythmus abhängig.

Generell lässt sich sagen, dass eine höhere Frequenz zu einem besseren Antwortverhalten des Systems führt - ein für echtzeitfähige Systeme sehr wichtiger Gesichtspunkt. Alle echtzeitfähigen Varianten greifen deshalb in die Programmierung des *Programmable Intervall Timers* ein.

Das Antwortverhalten des Systems hängt sehr stark davon ab, wie schnell ein Prozess durch einen höher-priorisierten Prozess unterbrochen werden kann. Bei jedem Auftreten des Timer-Interrupts prüft der Kernel, ob höher priorisierte Prozesse auf Ausführung warten. Dies hat allerdings einen entscheidenden Nachteil: Die CPU befindet sich eine größere Zeitspanne im Kern-Modus und nicht mehr im Benutzer-Modus, hieraus resultiert für Benutzer-Programme eine verlangsamte Ausführung.

Im Hinblick auf Echtzeitfähigkeit sind Timer nicht geeignet diese zu erreichen: Timerfunktionen werden immer von Bottom-Halfs (vgl. Absch. 7.2.3.1)

ausgeführt, diese werden jedoch unter Umständen nicht sofort beim auftreten des Interrupts ausgeführt, sondern erst mit einer Verzögerung von bis zu einigen hundert Millisekunden [18].

7.2.6 Der Scheduler

Der Scheduler ist der Teil des Kernels, der entscheidet, welcher laufbereite Prozess als nächstes Rechenzeit der CPU zugewiesen bekommt. Der Scheduler wird zu folgenden Zeitpunkten im System aufgerufen:

- Der Scheduler wird beim Verlassen des Kernel Modus nach dem Beenden einer Systemfunktion aufgerufen, wenn zuvor durch eine Interruptroutine die Notwendigkeit hierzu signalisiert wurde.
- Wenn ein Prozess oder Thread im Benutzermodus den Scheduler direkt oder indirekt aufruft, beispielsweise, weil er die Funktion *sleep()* startet.
- Wenn ein Prozess keine Systemfunktion ausführt und durch einen Uhrentick unterbrochen wird. Dies geschieht nur dann, wenn der Prozessor vor der Unterbrechung im Benutzermodus war.
- Innerhalb einer Systemfunktion im Kernel Modus kann der Scheduler aufgerufen werden, wenn z.B. auf ein belegtes Betriebsmittel zugegriffen werden soll. Der aktive Prozess wird dann innerhalb einer Systemfunktion schlafen gelegt. Dieses Verhalten hat nichts mit den später erwähnten *Preemption Points* zu tun, da hier nur der Scheduler aufgerufen wird, wenn eine Ressource belegt ist. Beim Antreffen eines freien Betriebsmittels wird die Systemfunktion ohne Start des Schedulers beendet.
- Wenn kein Prozess im System rechenbereit ist, wird der Idle-Prozess gestartet. Dieser ruft in regelmäßigen Abständen den Scheduler auf.

Nachdem das Auftreten des Schedulers im System erläutert wurde, handelt der nächste Abschnitt von den Algorithmen.

7.2.7 Austauschbarkeit des Schedulers

Durch die Systemfunktionen *sched_setscheduler()* und *pthread_setschedparam()* ist es möglich, die Scheduling Politik eines Prozesses bzw. Threads zu setzen. Linux stellt drei Arten des Schedulings zur Verfügung: SCHED_FIFO , SCHED_RR und SCHED_OTHER . SCHED_OTHER ist hierbei für normale Prozesse gedacht. Die beiden anderen Verfahren sind für Echtzeitaplikationen, die zeitkritische Vorgänge bearbeiten und daher eine deterministische Kontrolle über diverse Prozesse benötigen vorbehalten.

7.2.7.1 SCHED_OTHER

Hierbei handelt es sich um einen Standard "time-sharing" Scheduler, wie er bei UNIX-Betriebssystemen üblich ist. Die Priorität wird den Prozessen selbständig zwichen -20 und +20 zugeteilt, nach Aussen haben alle Prozesse unter diesem Algorithmus die statische Priorität 0. Man kann dem Scheduler allerdings nahelegen, bestimmten Prozessen mehr Rechenzeit zuzuteilen.

7.2.7.2 SCHED_FIFO

Dieser Scheduling-Algorithmus entspricht POSIX.1b FIFO für Echtzeit-Prozesse. Das "First In-First Out" Scheduling kann nur mit Prioritäten von 0 bis 99 belegt werden. D.h., wenn ein Prozess mit dem SCHED_FIFO Verfahren gescheduled wird, werden alle Prozesse, die mit SCHED_OTHER Verfahren verwaltet werden, schnellstmöglich unterbrochen. SCHED_FIFO ist ein einfacher Algorithmus ohen "time-slicing": Wenn ein SCHED_FIFO Prozess aufgrund eines höher prioren SCHED_FIFO Prozesses unterbrochen wird, bleibt er am Kopf der Liste stehen und wird, sobald alle höher prioren Prozesse blockieren, wieder aktiv (vgl. Abb. 4.6). Für jede Priorität existiert eine solche Liste, so dass die Reihenfolge zuerst durch die Priorität festgelegt wird und wenn mehrere gleich priore Prozesse vorhanden sind, die Liste die Reihenfolge bestimmt.

7.2.7.3 SCHED_RR

Das Round-Robin Verfahren wurde bereits in Abschnitt 4.4.3.1 beschrieben. Ein blockierender hoch priorisierter Prozess, der auf eine I/O Operation wartet, unterliegt einer gewissen Antwortzeit, bevor er gescheduled wird. Der Scheduler folgt dem Standard für POSIX round-robin Echtzeit-Prozesse.

7.2.8 Linux POSIX Konformität

GNU Linux ist ein Betriebssystem, welches "vorgibt" POSIX konform zu sein (vgl. Absch. 5.4). Dies ist ein Widerspruch, da Konformität das Durchlaufen und Bestehen eines Testzyklus bedeutet. Es hat nur eine POSIX konforme Linux Distribution gegeben, diese wird aber nicht mehr gepflegt [114] und die beim Booten erscheinende Meldung

```
POSIX conformance testing by UNIFIX
```

entspricht so nicht den Tatsachen.

Generell hält sich Linux an den POSIX Standard und die meisten für UNIX vorhandenen Programme können mit wenig Aufwand unter Linux kompiliert und ausgeführt werden [18].

Die *GNU C library* ist die Standard C Bibliothek für im GNU System [55]. Diese Bibliothek unterstützt die folgenden Standards:

- ISO C 99 [100]
- POSIX.1c
- POSIX.1j
- POSIX.1d
- UNIX98 [99]

Die GNU C library in der Version 2 ist momentan nur für GNU Systeme verfügbar (vgl. Absch. 3.3.0.9). Geplant ist, sie auch auf andere Systeme zu portieren.

7.3 Wege zur Echtzeit

Unter speziellen Randbedingung ist die Performance oder das Verhalten des Systems bezüglich der Reaktion auf Ereignisse von entscheidender Bedeutung. Aus diesem Grunde wurden verschiedene Modifikationen des Linux-Kernels entwickelt, die dieses Verhalten verbessern. Diese Verbesserungen lassen sich zum Beispiel bezüglich ihrer Auswirkungen auf die Antwortzeiten in Hard- und Soft-Realtime-Systeme unterteilen. Eine genaue Definition der Begriffe wurde bereits in Abschnitt 2.2 gegeben.

Eine andere und für die Betrachtung in dieser Arbeit geeignetere Klassifizierung ist die Unterteilung bezüglich des realisierten Designs der Systeme. Es gibt grundsätzlich vier Wege, die zur Echtzeitfähigkeit führen:

1. Unmodifizierter Linux-Kernel oder "einfach schnell genug".
2. Optimierung von Kernel und Treibern.
3. Ersetzen des Kernels.
4. Transparenter Scheduler und präemptiver Kernel.

Linux, in seinen aktuellen Varianten (2.2/2.4 Kernel-Varianten), ist kein Echtzeit-System im Sinne der Definition. Das aktuelle Design von Linux ist optimiert in Hinsicht auf Durchsatz - zu Lasten von Antwortzeiten und Determinismus: Der Scheduler wählt die auszuführenden Threads nach dem Grundsatz der Fairness aus. Threads mit niedriger Priorität können - um Starvation zu vermeiden (siehe Absch. 4.4.3.3) - nach Kriterien des Schedulers als nächste zur Ausführung kommen.

Der Linux Kernel ist nicht präemptiv - mit relativ langen Blockierzeiten, in denen kein Rescheduling erfolgen kann. Dies kann erst erfolgen, nachdem der Kernel einen Systemaufruf oder eine andere interne Aufgabe komplett bearbeitet hat und die Kontrolle zum aufrufenden Programm zurück kehrt (Abb. 7.3).

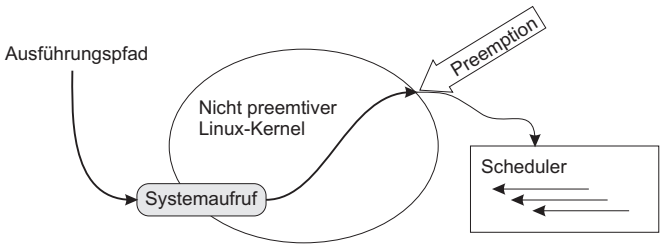


Abbildung 7.3: Nicht präemptiver Linux Kernel

7.3.1 Unmodifizierter Linux-Kernel: "einfach schnell genug"

Microsoft Windows und auch Sun/Solaris werden bereits seit einiger Zeit für harte und weiche Echtzeitaufgaben eingesetzt. Auf Grund der gestiegenen Leistungsfähigkeit der Intel und SPARC Hardware werden die Schwächen des Betriebssystems ausgeglichen. Diese Tatsache ist auch für Linux gültig. Auf Intel und SPARC Plattformen bietet Linux eine vergleichbare Performance bei gleicher Belastung des Systems. Linux sticht hier durch einen geringeren Ressourcenverbrauch und somit Preisvorteil hervor. Bei Solaris kommen noch die Kosten für die spezielle Hardware hinzu, die nicht vom Preisverfall bei Intel-Hardware profitieren kann. Linux erreicht auf Maschinen mit weniger als 200Mhz CPU-Frequenz Antwortzeiten im sub-Millisekunden Bereich.

Auf aktueller Hardware (z.B. ein 800Mhz Pentium III) liegt der Mittelwert der Antwortzeit mit einem unmodifizierten 2.4er Kernel bei 60 Mikrosekunden [84].

7.3.2 Optimierung von Kernel und Treibern

Bei vielen Anwendungen lassen sich die Echtzeitanforderungen auf einen relativ kleinen Bereich der Hard- und Software-Umgebung begrenzen, z.B. bei Audio- und Video-Anwendungen. Mit dem oben genannten Vorgehen - einfach "schnell genug" zu sein - lässt sich dieser Anwendungsbereich abdecken.

Probleme gibt es allerdings bei höheren Datenraten oder bei mehren parallel laufenden Anwendungen, dies führt dann dazu, dass Deadlines nicht mehr eingehalten werden.

Linux bietet an dieser Stelle die Möglichkeit, mit ein wenig Aufwand den Kernel und die entsprechenden Treiber hinsichtlich der gestellten Anforderungen zu optimieren. MontaVista Software [111] hat über 250 Embedded-Anwendungen auf dieser Basis entwickelt.

Linux-Gerätetreiber bieten dem Entwickler auf der einen Seite eine breite Basis an Entwicklungsressourcen, auf der anderen Seite beherbergen sie eine potentielle Quelle für Performance-Einbußen, wenn sie nicht optimal programmiert sind. Durch die offenen Quellen bietet sich zumindest die Möglichkeit, die Treiber zu entwickeln und hinsichtlich der gestellten Anforderungen zu optimieren.

Der Schlüssel liegt darin, alle im System verwendeten Treiber einer Prüfung zu unterziehen, um zu gewährleisten, dass keiner der Treiber einen Flaschenhals darstellt. Linux bietet die Möglichkeiten zu solchen Untersuchungen zum einen durch die offenen Programm-Quellen und zum anderen durch die dynamisch zu ladenden Kernel-Module.

7.3.3 Ersetzen des Kernels

Ein verallgemeinerter Ansatz der Optimierung von Gerätetreibern ist es, einen zweiten Kernel einzusetzen. Anstatt nur die wenigen kritischen Hardwareinterfaces zu analysieren, bietet dieser Ansatz ein allgemeines Echtzeitbetriebssystem mit eigenem Thread-Modell. Der Linux-Kernel läuft hierbei als Thread mit niedriger Priorität (Idle Thread) unter der Kontrolle des Echtzeitbetriebssystems.

Die Kommmunikation erfolgt hierbei über Shared-Memory oder über verschiedene IPCs. Die Echtzeitanwendungen werden als Kernel-Module geschrieben. Als Vorteil ist hierbei der vollständige Zugriff auf die Linux-Interna zu sehen, allerdings mit dem entscheidenden Nachteil, das es keinen Speicherschutz durch das Prozess-basierte Programmiermodel von Linux gibt.

7.3.4 Transparenter Scheduler und präemptiver Kernel: Erhalt der Linux-APIs

Bei bestehender Akzeptanz gegenüber den existierenden Linux-APIs, ist der eleganteste Ansatz der, die vorhandenen Mechanismen und Schnittstellen weiter zu benutzen und Linux bezüglich der Antwortzeiten zu optimieren. Dies läßt sich durch zwei Schritte erreichen:

7.3.4.1 Scheduler

Die bereits im Linux Thread-Modell vorhandenen Echtzeit-Verfahren (POSIX.1c, siehe Absch. 5.4.3) nutzbar zu machen, ist der erste Schritt. Theoretisch sind diese Richtlinien mit 99 Prioritätsebenen bereits im standard Linux-Kernel vorhanden, in der Realität werden sie aber zu oft überschrieben oder gar ganz ignoriert.

An dieser Stelle wird ein Scheduler implementiert, welcher diese Richtlinien korrekt umsetzt oder auf geeignete Weise für ein besseres Echtzeitverhalten sorgt. Der Algorithmus, welcher für den Scheduler eingesetzt wird, hängt von der jeweiligen Implementierung ab.

7.3.4.2 Vollständig präemptiver Linux-Kernel

Von den Linux-Entwicklern ist es lange Zeit für unmöglich gehalten worden, ein vollständig präemptives System zu erreichen, ohne die bestehenden Linux APIs zu verändern. Interessanterweise besteht aber schon seit längerer Zeit im standard Linux-Kernel die Technik für Preemption: Die SMP Version (**S**ymmetrisches **M**ulti **P**rozessing) ist präemptiv und bietet ein deterministisches Reaktionsverhalten, wenn die Konfiguration auf Uni-Prozessor-Systeme angepaßt wird.

Bei Anwendung dieser Technik erreicht man auf normalen Pentium-Systemen Task-Antwortzeiten unter 2,5 Mikrosekunden [84].

7.4 Die Varianten im Einzelnen

Betrachtet man die Landschaft der echtzeitfähigen Linux-Varianten, so findet man die oben beschriebenen Ansätze in den einzelnen Systemen - ganz oder teilweise umgesetzt - wieder.

7.4.1 RT-Linux

RT-Linux(**R**eal**T**ime-**L**inux) ist eine Erweiterung des Linux-Systems, um zeitkritische Aufgaben zu lösen. Im wesentlichen ist RT-Linux ein kleines echtzeitfähiges Betriebssystem, welches das Linux-System als weiteren Task ausführt. Hierbei ist Linux der Task mit der niedrigsten Priorität.

Linux wird hierdurch vollständig unterbrechbar (präemptiv) und kann, bei Bedarf, vollständig angehalten werden, um die CPU ganz für die wichtigeren Echtzeitaufgaben zu reservieren.

Gleichzeitig bleiben aber alle anderen Möglichkeiten die Linux bietet vollständig erhalten (X-Windows-System, Netzwerkfunktionalität, Entwicklungsumgebungen).

7.4.1.1 Hintergrund und historische Entwicklung

RT-Linux wurde am *New Mexico Institut of Technology, Scorro NM* von Victor Yodaiken und Michael Barabanov entwickelt. Ausschlaggebend war der Mangel an Geld, um ein kommerzielles RTOS zu kaufen [43].

Da Echtzeitsysteme und normale Betriebssysteme gegensätzliche Designziele haben, ist es nicht verwunderlich, das eine optimale Lösung für beide nicht gefunden werden kann. Die bei RT-Linux gewählte Lösung benutzt den gleichen Ansatz, wie das experimentelle Betriebsystem *MERT* von AT&T's Bell Labs [11]. Das Prinzip hinter diesem System lässt sich folgendermaßen definieren:

> " If a service or operation is inherently non Realtime, it should be provided by the standard OS and not by the Realtime components." [8]

Weitere Gesichtspunkte waren, das System klein, transparent, modular und erweiterbar zu halten. Transparent bedeutet, keine *black boxes* und vollständiger Determinismus. Modular heißt, das System stellt einfache Dienste bereit, um Funktionsblöcke für ein wachsendes System zu erstellen. Die einfachste RT-Linux Konfiguration unterstützt einzig hochfrequente Interruptbehandlung [8].

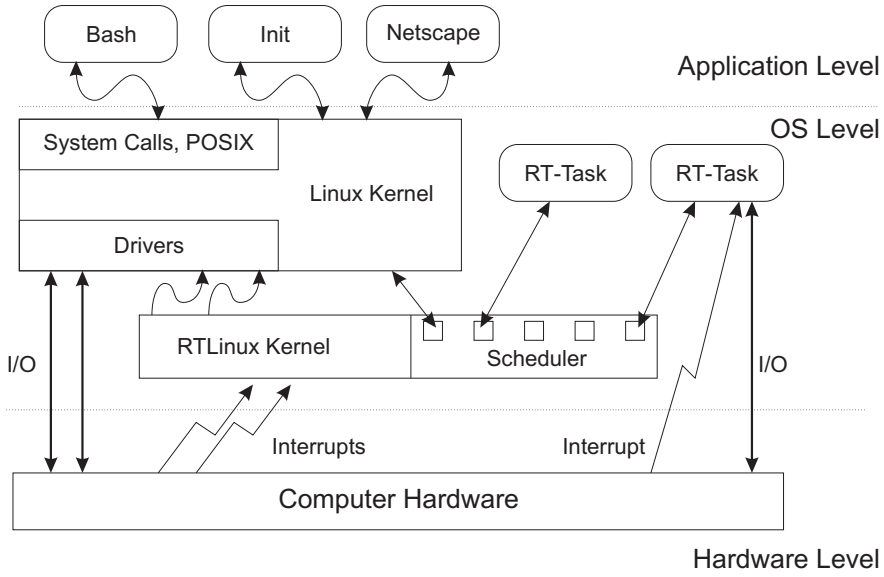


Abbildung 7.4: Implementation von RT-Linux.

7.4.1.2 Design und Implementierung

Den Linux-Kernel zu einem vollständig präemptiven Kernel mit kleinen Interruptverzögerungen zu machen, bedeutet eine tiefgreifende Restrukturierung. Der Ansatz, den RT-Linux verfolgt, ist wesentlich einfacher und eleganter und erlaubt es trotzdem, mit der Weiterentwicklung des Kernels Schritt zu halten.

Die meisten Betriebssysteme, Linux eingeschlossen, schalten die Interrupts in kritischen Bereichen des Kernels ab, um eine Synchronität zu erreichen (vergl. Absch. 4.4.2). Dies bedeutet, das der Kernel nicht präemptiv ist und Tasks, die im Kernel-Modus laufen, nicht dazu gebracht werden können, die CPU einem anderen Task zur Verfügung zu stellen, egal welche Priorität der Task hat. Um ein zeitliches Verhalten von Echtzeit-Tasks zu garantieren, wird das Ein- und Ausschalten der Interrupts von RT-Linux nur simuliert und dadurch der Linux-Kernel vollständig präemptiv.

Da der standard Linux-Kernel nur als Task mit der niedrigsten Priorität läuft, kann es passieren, dass er für längere Zeit "einfriert". In dieser Zeit ist die CPU damit beschäftig, die Echtzeit-Aufgabe zu bearbeiten.

Alle Interrupts werden grundsätzlich vom RT-Kernel abgefangen und nur dann an den Linux-Kernel weitergereicht, wenn keine Echtzeit-Tasks abgearbeitet werden müssen. Dies bedeutet, dass alle Interrupts dem Echtzeit-Kernel zugänglich sind. Der standard Linux-Kernel kann sie trotzdem noch bei Bedarf abschalten, da er nur die entsprechenden Makros von RT-Linux aufruft. Der Echtzeit-Kernel an sich ist nicht präemptiv, aber seine Routinen sind klein und schnell genug, um

keine signifikante Verzögerung hervorzurufen (Abb. 7.4).

Verschiedene Scheduling-Algorithmen können als Modul mit RT-Linux eingesetzt werden: Prioritäten basiert (siehe 4.4.3.2), Rate Monotonic Scheduler und Earliest Deadline First (siehe 4.4.3.4). Dies macht RT-Linux flexibel sowohl mit zeit- als auch mit ereignisgesteuerten Aufgaben umgehen zu können. Die Interruptverzögerungen liegen dabei an den Grenzen der Hardware [7].

7.4.1.3 RT-Linux POSIX Konformität

RT-Linux unterstützt POSIX 1003.13 (vgl. Absch. 5.4.4). Diese Entwicklungsumgebung besteht aus einem einzelnen, multi-threaded POSIX-Prozess, welcher allein auf der Maschine läuft. In diesem Model werden die RT-Linux Tasks zu POSIX-Threads und die RT-Linux Interrupthandler werden zu Signalhandlern. Der Linux-Kernel läuft als Thread mit der niedrigsten Priorität.

In einem SMP oder Multiprozessor-System besitzt jeder Prozessor einen eigenen Realtime-Prozess.

7.4.1.4 Lizenz und Distribution

RT-Linux ist ein hartes Echtzeitbetriebssystem. RT-Linux führt den Linuxkernel als Thread mit der niedrigsten Priorität aus. Hierdurch wird der Linuxkernel vollständig unterbechbar und es ist garantiert, dass ein Echtzeitprozess oder Interrupthandler niemals durch eine nicht-echtzeit Operation verzögert wird.

RT-Linux wird entwickelt und unterstützt von FSMLabs (www.fsmlabs.com).

Das besondere an RT-Linux ist das Patent, welches der Entwickler Victor J. Yodaiken auf die RT-Linux zugrunde liegenden Mechanismen hält [159]. Dieses Patent läßt sich relativ einfach beschreiben. Es umfasst die Methode, mit der RT-Linux seine Echtzeitfähigkeit erreicht. Zwei Techniken sind der Kern dieser Methode:

- Das Ausführen eines normalen Betriebssystems (hier Linux) als niedrig priorisierter Prozess innerhalb eines Echtzeitsystems. Dem normalen Betriebssystem ist es nicht möglich, das Echtzeitsystem zu blockieren.

- Die Platzierung einer Emulationsschicht zwischen den Hardwareinterrupts und dem normalen Betriebssystem. Linux arbeitet mit den Interrupts, ohne etwas von dieser Schicht zu bemerken. Im Hintergrund hat RT-Linux die volle Kontrolle.

Es finden sich noch weitere Punkte in der Patentschrift, die beiden hier beschriebenen sind aber der wesentliche Teil.

An diesem Punkt werfen sich einige Fragen auf, die im Abschnitt 8.2.1.2.2 behandelt werden: Wie hängt das Patent mit der GPL zusammen? Ist das Patent durchsetzbar? Welche Konsequenzen ergeben sich noch?

7.4.1.5 Eigenschaften und Charakteristik

RT-Linux	
Entwicklungs Plattform	Linux
Unterstützte Ziel-Plattformen	x86 (ab 486er), PowerPC, MIPS, Alpha
Unterstützte Compiler	GNU
Unterstützte Werkzeuge	GNU
Unterstützte Standards	POSIX 1003.13 kompatibel, POSIX 1003.1b, 1c, 1d, 1j teilweise
Entwicklungs Methode	Cross, Native
Betriebssystem liegt vor als	Source
Garantierte Interrupt-Anwortzeit	$< 20us$
Distribution	kostenlos (OpenRTLinux, download), kommerziell (RTLinux/Pro)
Support	Telefonisch (kommerziell), Maillinglist(frei)
Entwicklungswerkzeuge	GNU Werkzeugsatz

Tabelle 7.1: Eigenschaften und Charakteristik RT-Linux

7.4.2 RTAI

Die Struktur von RTAI (**R**eal **T**ime **A**pplication **I**nterface) und RT-Linux sind sich relativ ähnlich. RTAI benutzt das Modell des *Hardware Abstraction Layers* (HAL) um mit dem Linux-Kernel zu interagieren. Genau wie RT-Linux handelt es sich bei RTAI-Linux um ein kleines Echtzeitbetriebssystem, welches außerhalb des eigentlichen Linux-Kernels sitzt.

7.4.2.1 Hintergrund und historische Entwicklung

Begonnen hat die Entwicklung von RTAI-Linux im Jahr 1996 am *Department of Aerospace Engineering* am *Politecnico di Milano*. In den späten 80'er Jahren wurde ein eigenes RTOS basierend auf DOS entwickelt. Das Konzept des HAL wurde um die für Echtzeit notwendigen Services erweiter. Hierdurch entstand der Begriff RTHAL (**R**eal **T**ime **H**ardware **A**bstraction **L**ayer).

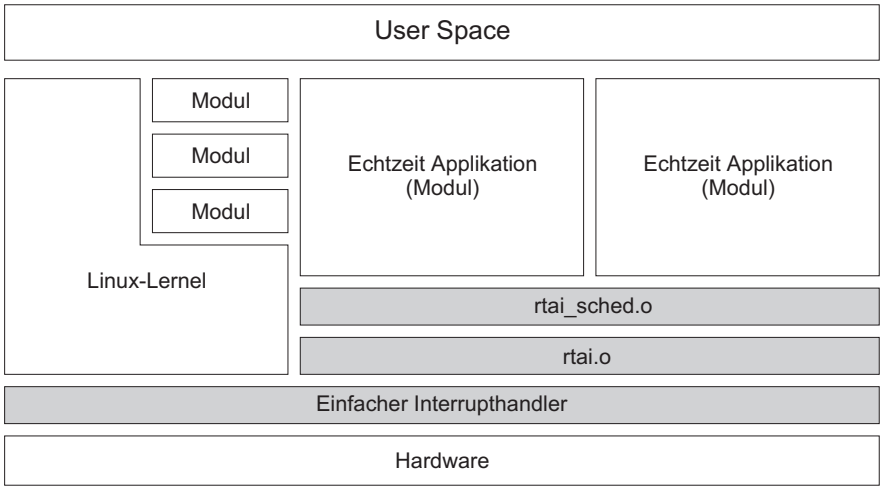


Abbildung 7.5: Implementation von RTAI-Linux.

Die ersten Versionen für den Linux-Kernel 2.0.xx benutzten noch den RT-Linux-Patch, da eine vollständige Implementierung des RTHAL's daran scheiterte, dass im Linux Kernel noch an zu vielen Stellen die Interrupts ge- bzw. entsperrt wurden. Der Patch wurde erweitert um Funktionen wie Semaphoren und Intertask Messaging. Weiterhin wurde auch die Behandlung von Floating-Point-Operationen unterstützt.

Zeitgleich mit dem Kernel-Serie 2.2.xx im Jahre 1999 erschien auch die erste vollständig eigenständige RTAI-Version welche UP- und SMP-Systeme unterstützte. Im September 1999 wurde RTAI-Linux um LXRT erweitert (**L**inu**X** **R**eal **T**ime). LXRT ermöglicht das Entwickeln und Testen von Echtzeitanwendungen im Userspace, bevor sie als Module in den Kernel portiert werden.

7.4.2.2 Design und Implementierung

Das Konzept von RTAI, dem Echtzeit-Kernel und RTHAL, dem Hardware Abstraction Layer, basiert auf der Nutzung von Linux-Kernel Modulen. Neben dem Modul um den HAL im Kernel zu installieren, ist auch der Rest von RTAI modular aufgebaut (siehe Abb. 7.5).

7.4.2.2.1 RTHAL Der Hardware-Abstraction-Layer (RTHAL) hat die primäre Aufgabe, Interrupts abzufangen und bei Bedarf weiterzuleiten. Die Interrupts werden entweder an den standard Linux Kernel oder an Echtzeit-Tasks weitergeleitet. Solche Interrupts, die für einen geschedulten Realtime Task gedacht sind, werden direkt an den Task gesandt, während die anderen Interrupts an den standard Linux Kernel übergeben werden und dort normal behandelt

werden.

Linux ist nie in der Lage die Interrupts zu sperren. Stattdessen werden alle Interrupts von RTAI verwaltet. Wenn Linux die Interrupts sperren möchte, erkennt RTAI die Anfrage und sorgt dafür, dass sich Linux - im Hinblick auf kritische Bereiche - so verhält, als seien wirklich alle Interrupts gesperrt. Dies wird dadurch erreicht, dass RTAI die Funktionen, um Interrupts zuzulassen oder zu sperren, emuliert und Interrupts an Linux "durchreicht".

7.4.2.2.2 Die Echtzeit-Dienste und Tasks Die Aufgabe, einen Echtzeit Scheduler und Dienste bereitzustellen, geschieht in Form von Modulen. Es werden hierzu die bereits im Linux Kernel vorhandenen Mechanismen benutzt, um die benötigten Funktionen als Modul zu laden. Es stehen drei Scheduler zur Verfügung:

UP Scheduler: Dieser Scheduler ist für Plattformen mit einem Prozessor vorgesehen, welche den 8254 als Timer benutzen. Unterstützt werden one-shot und periodic als Schedulingverfahren. Periodisch bedeutet: die Programmierung des Timers geschieht so, dass er in regelmäßigen Intervallen einen Interrupt auslöst, der ein Rescheduling veranlasst. Im Gegensatz dazu steht das one-shot Verfahren. Hierbei wird der Timer so programmiert, dass er nach einer festgelegten Zeitspanne genau einen Interrupt auslöst, der den Scheduler aufruft. Fur die neue Generierung eines Interrupts muss der Timer neu programmiert werden, was einen größeren Aufwand bedeutet, als beim periodischen Verfahren, allerdings sind so auch unterschiedlich lange Intervalle möglich, nach denen ein Rescheduling erfolgen kann.

SMP Scheduler: Dieser Scheduler ist für Multi-Prozessor Maschinen gedacht, die entweder 8254 oder APIC basiert sind. Der *APIC* ist der so genannte **A**dvanced **P**rogrammable **I**nterrupt **C**ontroller in Multiprozessorsystemen. Dieser hat u.a. die Aufgabe, die auftretenden Interrupts den einzelnen CPUs zuzuteilen. Tasks können an eine CPU gebunden werden oder symmetrisch auf einen Cluster von CPUs laufen. Der Scheduler kann auch auf Systemen eingesetzt werden, die nur einen Prozessor haben, aber deren Kernel mit SMP-Option kompiliert wurde.

MUP Scheduler: Der MUPs Scheduler ist vergleichbar mit dem SMP Scheduler. Der Unterschied ist hauptsächlich, dass es möglich ist, one-shot und periodic Scheduling - letzteres basierend auf periodischen Timern mit unterschiedlichen Perioden - simultan zu nutzen.

Genau wie die Dienste werden auch die Echtzeit Tasks behandelt. Sie werden als Kernel Module geladen. Sie haben so direkten Zugriff auf den HAL und die RTAI Dienst-Module.

7.4.2.2.3 LXRT Um die Entwicklung von Echtzeit Tasks zu erleichtern, wurde von den Entwicklern das LXRT-Modul zu RTAI hinzugefügt. Dies erlaubt die Entwicklung von Echtzeit-Tasks im User Space, mit der Möglichkeit, auf das RTAI API zuzugreifen.

Dies ist eine Besonderheit, die nur in RTAI existiert und die Entwicklung sehr vereinfacht, da sich Fehler in einem User-Space Prozess in der Regel nicht auf die Stabilität des Gesamt-Systems auswirken, Fehler in Kernelmodulen haben in der Regel gravierende Auswirkung (Absturz des Gesamt-Systems).

7.4.2.3 Lizenz und Distribution

RTAI wird am *Department of Aerospace Engineering* am *Politecnico di Milano* entwickelt und gepflegt (www.rtai.org). Es untersteht der GPL.

Hierdurch ist es auch nicht durch das RT-Linux Patent beeinträchtigt, obwohl es die im Patent beschriebenen Mechanismen benutzt. Eine Diskussion hierzu findet sich in Abschnitt 8.2.1.2.2.

7.4.2.4 Eigenschaften und Charakteristik

RTAI-Linux	
Entwicklungs Plattform	Linux
Unterstützte Ziel-Plattformen	x86 (ab 486er), PowerPC, ARM, MIPS, StrongArm
Unterstützte Compiler	GNU
Unterstützte Werkzeuge	GNU
Unterstützte Standards	POSIX 1003.1c kompatibel , (Pthreads, inclusive Mutexes und Condition Variables) POSIX 1003.1b kompatibel (nur Pqueues)
Entwicklungsmethode	Cross, Native
Betriebssystem liegt vor als	Source
Garantierte Interrupt-Anwortzeit	$< 20us$
Distribution	kostenlos (download)
Support	Maillinglist(frei)
Entwicklungswerkzeuge	GNU Werkzeugsatz

Tabelle 7.2: Eigenschaften und Charakteristik RTAI-Linux

7.4.3 KURT

KURT (**K**ansas **U**niversity **R**eal-**T**ime) ist ein System, welches weichen Echtzeitanforderungen genügt. Allerdings kennzeichnen die Entwickler von KURT diese Eigenschaft als *firm* [120][64]. Die Bezeichnung *firm* wird folgendermaßen abgeleitet: Viele Anwendungen - als Beispiel werden Video-Anwendungen aufgeführt - haben sowohl hohe Ansprüche an das Zeitverhalten als auch den Bedarf, auf Systemdienste zugreifen zu können. Der erste Punkt wird den harten Echtzeitsystemen zugeordnet, der zweite den weichen Echtzeitsystemen. Analysiert man das System anhand der Kriterien Reaktionszeit und Determinismus, so stellt man fest, dass nur weiche Echtzeit - im Sinne der in Abschnitt 2.2 gegebenen Definition - erreicht wird [4].

7.4.3.1 Hintergrund und historische Entwicklung

Die Entwicklung von KURT wurde im Jahr 1998 am *Information and Telecommunication Technology Center* der *University of Kansas* begonnen [118]. Anstoß für die Entwicklung war der Bedarf an einem echtzeitfähigen Betriebssystem, welches sowohl Anforderungen bezüglich von vorhersagbaren Reaktionszeiten genügt und zugleich den Zugriff auf sämtliche Systemdienste des Betriebssystems bietet. Als weiteres Ziel galt, das Ganze auf Basis handelsüblicher Hardware - Desktop PCs - zu realisieren, um auch eine preiswerte Alternative gegenüber Systemen zu erhalten, welche auf spezielle Hardware angewiesen sind.

Die Entwicklung wurde von einigen zum Teil kommerziellen bzw. proprietären echtzeitfähigen Betriebssystemen beeinflusst. Das von Microsoft entwickelte echtzeitfähige System *Rialto* benutzt den Hardware Timer des Systems auf gleiche Weise wie KURT [10]. Der Timer wird aperiodisch betrieben, um eine feinere Auflösung beim Timing zu erreichen. Da es aber ein proprietäres System ist, schied es für eine Nutzung am ITTC aus.

Das in Stanford entwickelte System *SMART* benutzt den gleichen Scheduling-Algorithmus, bietet aber nicht die gleiche Auflösung beim Timing [75].

Alle diese Systeme bieten zwar eine gute Performance und stellen den Applikationen genügend Dienste bereit, schieden aber dann aus Kostengründen aus.

7.4.3.2 Design und Implementierung

Die Implementation von KURT teilt sich in zwei Schritte auf: Zuerst wird die zeitliche Auflösung des Systems verbessert, da die Echtzeit-Anwendungen eine höhere Anforderungen in diesem Bereich stellen (s.o.). Die vom unmodifizierten Linux gebotene Auflösung der Uhr von 10 Millisekunden ist zu groß. Der zweite Schritt besteht darin, den standard Linux-Scheduler zu ersetzen.

7.4.3.2.1 Erhöhung der zeitlichen Auflösung Im standard Linux ist der Hardware Timer Chip so programmiert, dass in festen Zeitintervallen ein Interrupt generiert wird. Bei jedem Auftreten des Interrupts wird die Softwareuhr aktualisiert und es wird geprüft, ob Prozesse auf Ausführung warten. Die Softwareuhr ist nichts anderes als ein Zähler der dieser Interrupts seit Start des Systems aufsummiert.

Da der Kernel nur bei Eintreten eines Timer-Interrupts die Liste der wartenden Prozesse überprüft, ist die Länge der Zeitspanne zwischen dem Auftreten der Interrupts die kleinste sinnvolle Zeitheinheit, in der Prozesse geschedult werden können. Diese Zeitspanne (auch *Jiffy* genannt) bestimmt also die zeitliche Auflösung des Kernels.

Der einfachste Ansatz ist, den Timerchip so zu programmieren, dass er die CPU mit einer höheren Frequenz unterbricht. Dies würde die Länge eines Jiffy verkürzen und so die zeitliche Auflösung erhöhen. Der Nachteil dieser Lösung ist, dass dieser Ansatz zu einem erheblichen Anwachsen des Verwaltungsoverheads führt, da die CPU - unabhängig davon, ob Aufträge zu schedulen sind - häufiger unterbrochen wird.

Die von den Entwicklern realisierte Lösung dieses Problems sieht so aus: Es besteht ein großer Unterschied zwischen der zeitlichen Auflösung und der Frequenz des Auftretens von wartenden Aufträgen, d.h. die Applikationen haben zwar Deadlines im Mikrosekunden-Bereich, aber es tritt nicht jede Mikrosekunde eine Deadline ein. Es wird also ein Mechanismus eingebaut, der dafür sorgt, dass mit einer Genauigkeit von Mikrosekunden Timer-Interrupts generiert werden können, aber nicht zu jeder Mikrosekunde.

Der Timer-Chip wird so programmiert, dass er die CPU nicht mit einer festen Frquenz unterbricht, sondern es wird ein Timer-Interrupt generiert, wenn der nächste Auftrag zu Auführung kommen muss. Dies führt dazu, dass die CPU nur dann unterbrochen wird, wenn es nötig ist und nicht - wie oben beschrieben - immer in festen Intervallen.

Da viele Kernel-Subsysteme auf das regelmäßige Aktualisieren des Jiffy-Counters angewiesen sind, werden neben den Interrupts bedingt durch die auszuführenden Prozesse, noch die entsprechenden Interrupts generiert um diesen periodischen "Herzschlag" zu generieren.

7.4.3.2.2 KURT Scheduler Wie oben schon erwähnt, besteht der zweite Teil der KURT-Implementation daraus, einen neuen Scheduling-Algorithmus zu den bestehenden Standards hinzuzufügen.

Der KURT-Scheduler sorgt dafür, dass Echtzeitaufgaben zu den festgelegten Zeiten ausgeführt werden. Es sind weiterhin drei Betriebsarten eingeführt worden: In der ersten Art (*normal Mode*) verhält sich das System wie ein standard Linux-

System, in der zweiten (*focused mode*), werden nur solche Prozesse gescheduled, welche dem KURT-Algorithmus zugeordnet sind. Im dritten Modus (*mixed Mode*) kommen auch noch die normalen Prozesse zur Ausführung.

Im focused Mode können normale nicht-echtzeit Prozesse nicht mit den Echtzeit-Prozessen interferieren. Die Echtzeit-Prozesse haben aber vollen Zugriff auf alle Systemfunktionen, wie z.B. Netzwerkprotokolle und Hardware-Treiber.

Das KURT-System besteht aus mehreren Kernel-Modulen, welche alle bestimmte Aktivitäten ausführen und einem Basis-Modul, welches dafür sorgt, dass die anderen Module zu einem bestimmten Zeitpunkt ausgeführt werden. Die Echtzeit-Anwendungen übergeben diesem Basis-Modul eine Datei, in der eine Liste mit den gewünschten Zeiten steht, an denen die Anwendung zur Ausführung kommen soll.

7.4.3.3 Lizenz und Distribution

KURT wird vom *Information and Telecommunication Technology Center* der *University of Kansas* entwickelt und gepflegt. KURT unterliegt der GPL.

7.4.3.4 Eigenschaften und Charakteristik

KURT-Linux	
Entwicklungs Plattform	Linux
Unterstützte Ziel-Plattformen	x86 (ab486er), PowerPC, ARM, MIPS, StrongArm
Unterstützte Compiler	GNU
Unterstützte Werkzeuge	GNU
Unterstützte Standards	POSIX 1003.1c kompatibel , (Pthreads, inclusive Mutexes und Condition Variables) POSIX 1003.1b kompatibel (nur Pqueues)
Entwicklungsmethode	Cross, Native
Betriebssystem liegt vor als	Source
Garantierte Interrupt-Anwortzeit	$< 20us$
Distribution	kostenlos (download)
Support	Maillinglist(frei)
Entwicklungswerkzeuge	GNU Werkzeugsatz

Tabelle 7.3: Eigenschaften und Charakteristik KURT-Linux

7.4.4 Montavista-Linux

MontaVista Linux ist eine Linux Distribution, welche auf die Bedürfnisse von Entwicklern für Embedded und Echtzeit-Systeme zugeschnitten ist. Spezielle Werkzeuge ermöglichen die Cross-Plattform-Entwicklung.

7.4.4.1 Hintergrund und historische Entwicklung

Seit dem Jahr 1999 bietet MontaVista Linux basierte Lösungen für den Embedded-Markt an. Zuerst unter dem Namen "Hard Hat Linux". Diese Distribution war speziell für Embedded-Anwendungen ausgelegt, d.h zum Beispiel Booten ohne angeschlossenes Keyboard und Monitor sowie einem kleinen Footprint - d.h. mit kleiner Speicherbelegung - des Systems. Echtzeitfähigkeit kam dann bei späteren Versionen hinzu. Diese Distribution wurde 2002 dann in MontaVista-Linux umbenannt.

7.4.4.2 Design und Implementierung

Ähnlich wie bei KURT gliedert sich die Implementierung von MontaVista-Linux in zwei Teile: den neuen Scheduler und den so genannten *Preemption-Patch*.

7.4.4.2.1 Der Scheduler Der von MontaVista eingesetzte Scheduler ist vor dem fairness-basierten Scheduler angesiedelt. Er sorgt dafür, dass die Prioritäten eingehalten werden: erst wenn keine Echtzeit-Threads auf Ausführung warten, wird die Kontrolle an den standard Scheduler übergeben. Dieser Ansatz berücksichtigt die korrekte Einhaltung der Prioritäten, löst aber nicht das Problem der Unterbrechbarkeit des Linux-Kernels. Die Arbeit an diesem Scheduler ist als Patch frei im Internet verfügbar [5].

7.4.4.2.2 Präemption-Patch Da beim standard Linux Systemaufrufe nicht unterbrechbar sind und daraus resultierend Zeitverzögerungen für höher priorisierte Prozesse im Bereich von mehreren zehn Millisekunden entstehen können, wurde von MontaVista der Kernel so verändert, dass Systemaufrufe unterbrechbar sind und die Verzögerungen um Größenordnungen geringer ausfallen. Die Änderungen, die im Patch enthalten sind, umfassen folgende Bereiche im Kernel.

Der Patch modifiziert die Implementierung der Spin Locks, indem er sie von der spezifischen Implementation bei einem symmetrischen Multiprozessor System (SMP) in präemptive Locks verändert. Die Sperrfunktion ist dafür verantwortlich, den Wiedereintritt in Bereiche der Kernelsoftware zu steuern (vergl. Absch. 4.4.2).

Weiterhin wird die Interrupt-Handling-Software angepasst: es wird ein Rescheduling nach einem Interrupt ermöglicht, wenn ein Prozess mit höherer Priorität

ausführbar wurde, selbst wenn der unterbrochene Prozess im Kernelmodus lief.

Durch diese Änderungen wird der Kernel generell unterbrechbar - mit kleinen nicht-unterbrechbaren Bereichen, die den im Spin Lock gehaltenen Abschnitten beim SMP-Kernel entsprechen. Als Resultat wird die Reaktionzeit auf Prozessebene sowohl beim Durchschnitt als auch beim Maximalwert drastisch gesenkt [91]. Dieser Patch ist ebenso im Sinne von Freier Software im Internet verfügbar [83].

7.4.4.3 Lizenz und Distribution

Montavista Linux wird als vollständiges Open Source Project von *MontaVista Software Inc.* [111] entwickelt und gepflegt. Es ist vergleichbar einer Linux-Distribution von SuSE oder Red Hat.

7.4.4.4 Eigenschaften und Charakteristik

Montavista-Linux	
Entwicklungs Plattform	Linux, Windows
Unterstützte Ziel-Plattformen	x86 (ab 486er), PowerPC, ARM, MIPS, StrongArm, SPARC, Super-H
Unterstützte Compiler	GNU
Unterstützte Werkzeuge	GNU und Eigenentwicklungen
Unterstützte Standards	POSIX 1003.1
Entwicklungsmethode	Cross, Native
Betriebssystem liegt vor als	Source, Objekt
Garantierte Interrupt-Anwortzeit	Plattformabhängig
Distribution	kommerziell
Support	Telefonisch (kommerziell),
Entwicklungswerkzeuge	GNU Werkzeugsatz Target Configuration Tool

Tabelle 7.4: Eigenschaften und Charakteristik Montavista-Linux

7.4.5 TimeSys-Linux

Ebenso wie MontaVista Linux ist TimeSys Linux auf den embedded Bereich ausgerichtet. Spezielle Entwicklungswerkzeuge stehen zur Verfügung.

7.4.5.1 Hintergrund und historische Entwicklung

TimeSys Linux ist als Embedded Linux Distribution gedacht. Der Kernpunkt ist der TimeSys Linux Kernel, dieser Kernel beseitigt viele der Einschränkungen bezüglich der Echtzeitfähigkeit des standard Linux Kernels. TimeSys stellt dem Kernel ein Reihe von Entwicklungstools zur Seite, diese basieren aber nicht auf Linux sondern auf Windows und sind für die Cross-Plattform-Entwicklung interessant.

7.4.5.2 Design und Implementierung

Um die Einschränkungen für Echtzeitanwendungen des standard Kernels zu beseitigen, wurde der Kernel an einigen Stellen derart modifiziert, dass die bestehenden Vorteile nicht beeinträchtigt werden. Auch bei dieser Implementation bleiben also die grundlegenden Eigenschaften von Linux erhalten.

7.4.5.2.1 Vollständige Unterbrechbarkeit Wie bereits mehrfach erwähnt, ist der Linux Kernel nicht präemptiv. Der Kernel prüft erst nachdem er einen Betriebssystemaufruf komplett abgearbeitet hat, ob ein neuer Prozess zur Ausführung kommen muss.

Der TimeSys Linux Kernel prüft, ob ein höher priorisierter Task zur Ausführung kommen muss, am frühestmöglichen Punkt. Dies ist beim Auftreten eines Interrupts der Fall. Der Kernel prüft auf ein Rescheduling nach jeder Rückkehr aus einem Interrupt - auch wenn er sich vorher im Kernelmodus befand.

7.4.5.2.2 Schedulable Interrupt Behandlung Im TimeSys Linux Kernel ist jedem Interrupt ein IRQ-Thread zugewiesen, welcher die obere Hälfte dieses Interrupts ausführt. Wenn ein Interrupt auftritt, bestätigt der Kernel dies dem Interrupt-Kontroller und er maskiert den Interrupt. Anschließend wird der zugehörige IRQ-Thread gestartet. Der Interrupt-Handler, welcher vom Treiber installiert wurde, wird innerhalb dieses IRQ-Threads ausgeführt. Da der IRQ-Thread nur ein weiterer Kernel-Thread ist und der Kernel vollständig unterbrechbar ist, konkurriert dieser mit den anderen um CPU-Zeit. Der Interrupt-Handler ist genauso mit einer Priorität versehen, wie alle anderen Tasks auch. Es ist bei Bedarf möglich, einem IRQ eine niedrigere Priorität zuzuweisen, als einem User-Task.

7.4.5.2.3 Schedulable SoftIRQ Behandlung Im standard Linux Kernel werden die unteren Hälften der Interrupts durch einen Mechanismus, bezeichnet als *SoftIRQs*, behandelt.

TimeSys Linux ordnet diesen SoftIRQs einen SoftIRQ-Thread zu. Dieser Thread kann eine Priorität zugewiesen bekommen, welche die Echtzeiteigenschaften und die Auswirkungen auf das Gesamtsystem kennzeichnet. Hieraus resultiert, dass alle Aspekte bei der Interruptbehandlung auf priorisierte Art gehandhabt werden.

7.4.5.2.4 Der Scheduler Der standard Linux-Scheduler wird bei TimeSys-Linux durch einen sogenannten *Multilevel-Bitmapped-Scheduler* ersetzt. Dieser trifft die Entscheidungen über den als nächstes zur Ausführung kommenden Task in konstanter Zeit.

7.4.5.2.5 Spin Locks zu Mutexes Um die Konsistenz der Daten im Kernel zu gewährleisten, ist eine Unterbrechung von Systemfunktionen im standard Linux-Kernel nicht erlaubt.

TimeSys Linux erlaubt es Systemfunktionen zu unterbrechen und muss deshalb die Daten schützen. Genau wie MontaVista Linux benutzt TimeSys hierzu die bereits im Kernel vorhandenen Spin Locks, die für die Unterstützung von SMP-Systemen notwendig sind. Auf Systemen mit nur einem Prozessor werden die Spin Locks in Noops (*no operation*) - mit möglicher Sperrung der Interrupts - umgesetzt. TimeSys ersetzt, wann immer möglich, Spin Locks durch Mutexes in beiden Systemen.

7.4.5.3 Lizenz und Distribution

TimeSys Linux basiert auf dem freien unter der GPL stehenden präemptiven Kernel.

TimeSys bietet zusätzliche Module mit speziellen Eigenschaften als Ergänzung zum Kernel an: TimeSys Linux/Realtime fügt Priority Inheritance für Mutexes hinzu. TimeSys/CPU unterstützt die CPU-Reservierung welche garantiert, dass eine CPU für zeitkritische Aufgaben verfügbar ist. TimeSys/Net unterstützt die Reservierung von Netzwerkbandbreite für kritische Threads.

7.4.5.4 Eigenschaften und Charakteristik

Timesys-Linux	
Entwicklungs Plattform	Linux, Windows
Unterstützte Ziel-Plattformen	x86, PowerPC, AR, MIPS, StrongArm, SPARC, Super-H
Unterstützte Compiler	GNU
Unterstützte Werkzeuge	GNU
Unterstützte Standards	POSIX 1003.1
Entwicklungsmethode	Cross, Native
Betriebssystem liegt vor als	Source, Objekt
Garantierte Interrupt-Anwortzeit	Plattformabhängig
Distribution	kommerziell
Support	Telefonisch (kommerziell), Maillinglist(frei)
Entwicklungswerkzeuge	TimeStorm (Entwicklungsumgebung) TimeWiz (Softwaredesigntool) TimeTrace (Debugging Wekzeug)

Tabelle 7.5: Eigenschaften und Charakteristik Timesys-Linux

7.4.6 RED-Linux

Genau wie MontaVista- und TimeSys-Linux wird beim RED-Linux-Projekt der Ansatz verfolgt, den Linux-Kernel bezüglich seines Echtzeitverhaltens zu verbesseren und nicht eine eigenes RTOS unterhalb des Kernels anzusiedeln, wie bei RT-Linux und RTAI.

7.4.6.1 Hintergrund und historische Entwicklung

Die Entwicklung von RED-Linux findet am *Department of Electrical and Computer Engineering* der *University of California, Irvine* statt. Begonnen hat sie im Jahr 1998 als Grundlage für Forschung im Bereich Realtime Systeme, um eine kostengünstige und offene Plattform zu erhalten [81]. Deshalb wurde die Entscheidung getroffen, die möglichen Anwendungsbereiche von Linux um den Bereich der Echtzeit- und Embedded-Anwendungen zu erweitern. Diese Entwicklung wird RED-Linux (**R**eal-Time and **E**mbedde**D**-Linux) genannt. Der Linux-Kernel wird um Echtzeitfähigkeit erweitert, gleichzeitig bleiben aber alle anderen Eigenschaften erhalten.

Es handelt sich hier um ein Forschungsprojekt, welches unter anderem zur Untersuchung von unterschiedlichen Scheduling-Algorithmen dient. Die letzte Version basiert auf dem Linux-Kernel 2.2.14 aus dem Jahr 2000. Ist also nicht mehr auf dem aktuellen Stand.

7.4.6.2 Design und Implementierung

RED-Linux fügt dem Linux Kernel drei Komponenten hinzu: einen eigenen Scheduler, einen sogennanten *Microtimer* und eine Emulation der Interrupts durch Software. Die letztgenannten Mechanismen wurden von RT-Linux (vgl. Absch. 7.4.1) portiert. Zusätzlich wird die Unterbrechbarkeit des Kernels verbessert, indem Systemaufrufe in kleinere Blöcke unterteilt werden.

7.4.6.2.1 Blockierzeiten im Kernel Wie in den meisten UNIX-Systemen ist auch bei Linux ein Systemaufruf eine Operation, die nicht unterbrochen werden kann (*atomic operation*). Im Gegensatz zu MontaVista- und TimeSys-Linux wird bei RED-Linux allerdings kein vollständig unterbrechbarer Kernel realisiert, sondern es wird ein Model verwirklicht, welches als *cooperative preemtion* Model (CP) bezeichnet wird [81]. Es werden im Kernel Kontrollpunkte (*preemption points*) eingerichtet, an denen der Kernel prüft, ob es wartende Echtzeit-Prozesse gibt und diese bei Bedarf ausführt: Der Code, welcher bei einem *Preemption Point* ausgeführt wird, sieht ungefähr so aus:

```
if (real_timejob_waiting) scheduler();
```

Wenn es keine wartenden Jobs gib, ist es auch nicht notwendig, den Scheduler aufzurufen. Der Overhead sind in diesem Falle nur die für den Vergleich notwendigen Operationen.

7.4.6.2.2 Der Scheduler Genau wie beim MontaVista-Linux wird ein zusätzlicher Scheduler eingesetzt, welcher dafür sorgt, dass die Echtzeitanforderungen eingehalten werden. Das besondere am Scheduler ist, dass er die drei populärsten Scheduling-Algorithmen für Echtzeit-Anwendungen in einem Framework vereint.

Die Algorithmen Es sind die drei folgenden Algorithmen implementiert: Prioritäten basiertes, Zeit basiertes (time driven) und Anteil basiertes Scheduling (shared driven).

Das erste Konzept beruht auf festen oder dynamischen Prioritäten, nach denen entschieden wird, welcher Task als nächstes zur Ausführung kommt. Beim Zeit basierten Scheduling wird anhand der aktuellen Zeit eine planmäßige Ausführung

eines Task durchgeführt. Der letzte Ansatz teilt jedem Task einen (festgelegten) Anteil an den zu Verfügung stehenden Ressourcen (CPU-Zeit) zu und teilt die Ausführungszeit dann gleichmäßig entsprechend dem vorher festgelegten Anteil den Tasks zu.

Realisiert sind diese Algorithmen in Form eines Frameworks, welches anhand von vier Attributen - *Priorität*, *start_time*, *finish_time* und *budget* - entscheidet, welcher Algorithmus zutreffend ist.

Priorität Die Priorität eines Tasks legt fest, wie wichtig der Job, im Vergleich zu anderen im System, ist.

Start_time Die Startzeit legt fest, wann ein Task ausgeführt werden kann, ein Task kann nicht vor seiner Startzeit ausgeführt werden.

Finish_time Dieser Zeitpunkt ist die Deadline für einen Task. Nach diesem Zeitpunkt muss der Task beendet werden, auch wenn er noch nicht fertig ist.

Budget Das Budget repräsentiert den Umfang der Ausführungszeit, der für diesen Job zu reservieren ist.

Diese Faktoren können in Kombination oder einzeln als Entscheidungsgrundlage für den Scheduler benutzt werden.

Die Komponenten des Schedulers Analog zu Abschnitt 4.4.3.5 wird der Scheduler in zwei Komponenten zerlegt (Abb. 7.6). Eine davon ist der *schedule allocator*. Er setzt die oben beschrieben Attribute für die Echtzeit-Prozesse entsprechend dem gewählten Scheduling Alogrithmus um . Die zweite Komponente ist der *schedule dispatcher*. Dieser analysiert die aktuellen Werte der Attribute, wählt anhand dieser einen Task aus der Warteschlange und bringt ihn zur Ausführung. Die Art, wie der *allocator* die Attriute setzt und der *dispatcher* einen Task auswählt, reflektieren den Scheduling-Algorithmus.

7.4.6.3 Lizenz und Distribution

RED-Linux wird in dieser Form nicht mehr weiterentwickelt. Es ist vielmehr in REDICE Linux überführt. Deshalb wird hier auf eine weitere Beschreibung verzichtet und auf den nächsten Abschnitt verwiesen.

7.4.7 REDICE Linux

REDICE Linux ist die Weiterentwicklung von RED Linux. Im Jahr 2000 wurde die Firma REDSonic Inc. [126] von Dr. Kwei-Jay Lin der Universität von Kalifornien in Irvine als Solution Provider für Linux Systeme gegründet.

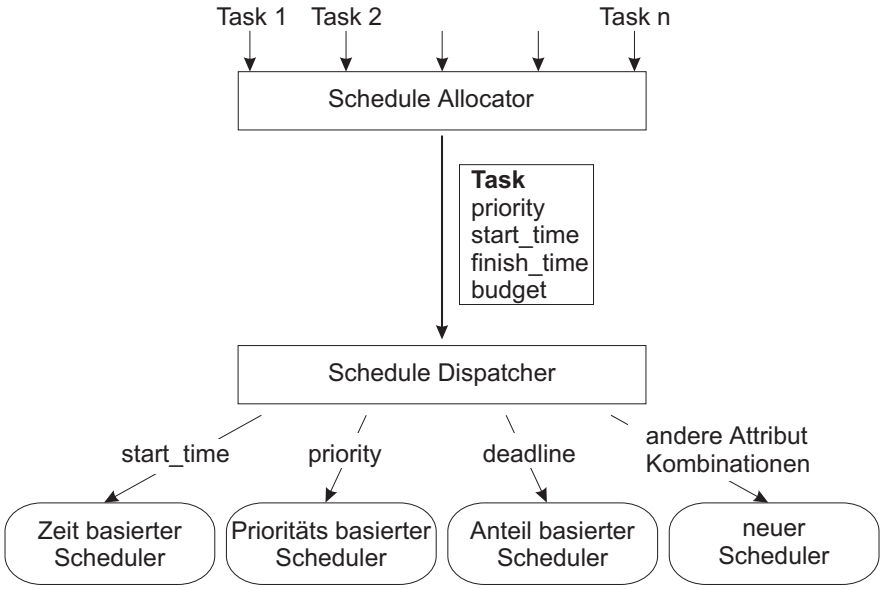


Abbildung 7.6: Das Scheduling-Framework von RED-Linux.

7.4.7.1 Hintergrund und historische Entwicklung

REDICE Linux ist ein Echtzeit Kernel, der speziell für den embedded Bereich gedacht ist. Er ist die Weiterentwicklung des RED Linux Kernels.

7.4.7.2 Design und Implementierung

REDSonic integriert für seine echtzeitfähige Linux-Variante zwei bereits vorgestellte Technologien (siehe Abb. 7.7) : RED-Linux, um Echtzeit im User Space zu ermöglichen und RTAI, um Echtzeit im Kernel zu ermöglichen. Ziel dieser Implementierung ist es, die Fähigkeiten beider Systeme in einem zu nutzen.

7.4.7.3 Lizenz und Distribution

REDSonics REDICE Linux ist eine echtzeitfähige Linuxversion, die einen präemptiven Kernel mit RTAI kombiniert. Hierdurch sind zum einen die harten Echtzeiteigenschaften von RTAI nutzbar, zum anderen die Vorteile eines präemptiven Kernels.

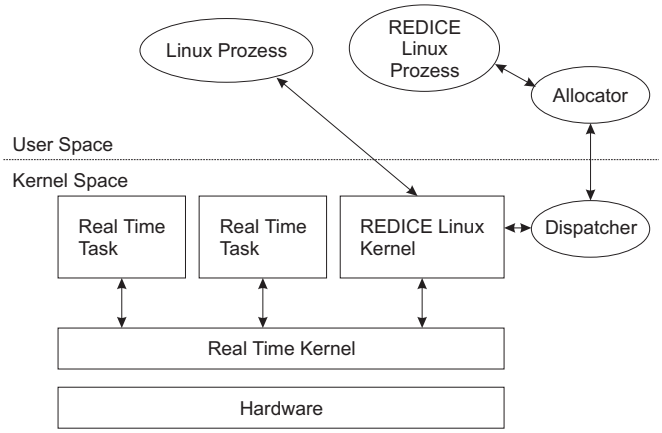


Abbildung 7.7: REDICE Linux Kernel.

7.4.7.4 Eigenschaften und Charakteristik

REDICE-Linux	
Entwicklungs Plattform	Linux, Windows
Unterstützte Ziel-Plattformen	x86, PowerPC, ARM, MIPS, StrongArm
Unterstützte Compiler	GNU
Unterstützte Werkzeuge	GNU
Unterstützte Standards	POSIX 1003.13, POSIX.1b, 1c, 1d
Entwicklungsmethode	Cross, Native
Betriebssystem liegt vor als	Source, Objekt
Garantierte Interrupt-Anwortzeit	$< 20us$(vergleiche RTAI)
Distribution	kommerziell
Support	Telefonisch (kommerziell), Maillinglist(frei)
Entwicklungswerkzeuge	RED-Builder XE (Entwicklungsumgebung) RED-Imgage-Builder (Crossdevelopment Werkzeug) RED-Probe (Debugging Werkzeug)

Tabelle 7.6: Eigenschaften und Charakteristik REDICE-Linux

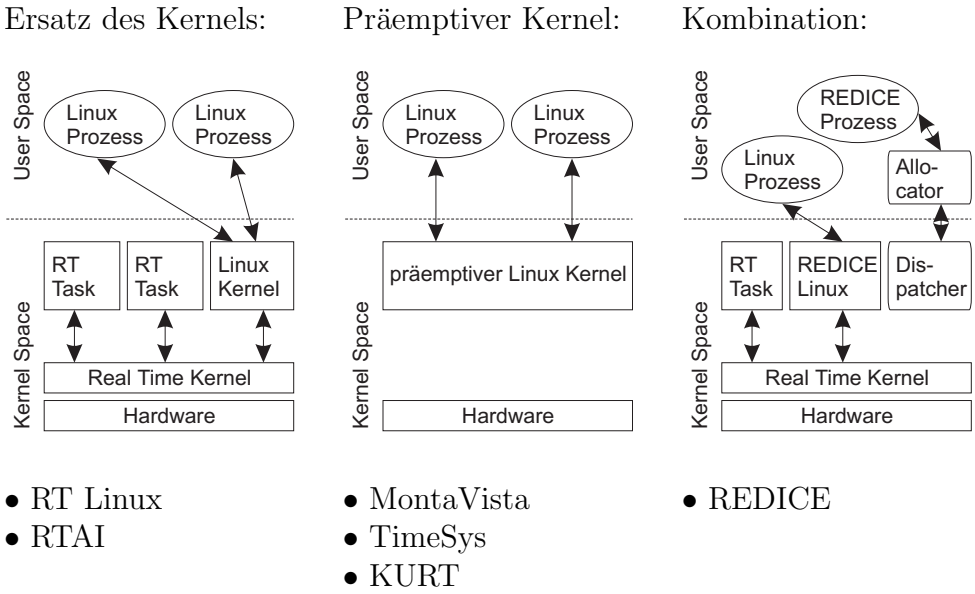
7.5 Zusammenfassung

Für die untersuchten Varianten ergibt sich die in Abbildung 7.8 dargestellte Situation bezüglich der realisierten Wege zu einem echtzeitfähigen Linux System.

Es gibt zwei Ansätze, welche das Prinzip aus Abschnitt 7.3.3 anwenden: RT-Linux (Absch. 7.4.1) und RTAI (Absch. 7.4.1) fügen einen zweiten Kernel ein, der für die Echtzeitfähigkeit sorgt.

Bei MontaVista- und TimeSys-Linux (Absch. 7.4.4) wird der Ansatz aus Abschnitt 7.3.4 verfolgt, den Kernel vollständig präemptiv zu machen und um einen Scheduler zu erweitern. Bei KURT (Absch. 7.4.3) wird nur ein ein Teil dieses Konzeptes angewendet, es wird zwar der Scheduler ersetzt, der Kernel allerdings nicht um die Unterbrechbarkeit erweitert.

Das von REDICE-Linux verwendete RED Linux verbessert die Unterbrechbarkeit des Kernels nur, ohne ihn vollständig präemptiv zu machen. REDICE Linux ist auch die einzige Variante, welche die zwei grundsätzlich unterschiedlichen Ansätze - Ersetzen auf der einen und Modifizieren des Kernels auf der anderen Seite - zusammen verwendet.



- RT Linux
- RTAI

- MontaVista
- TimeSys
- KURT

- REDICE

Abbildung 7.8: Überblick: Konzept der Varianten.

Kapitel 8

Analyse Charakteristika der einzelnen Varianten

8.1 Motivation

Nachdem das Design des standard Linux Kernels und der echtzeitfähigen Varianten vorgestellt wurde, erfolgt die Analyse dieser Varianten anhand der in Kapitel 6 formulierten Anforderungen. Hierbei beschränkt sich die Betrachtung nicht nur isoliert auf den Kernel, sondern betrachtet das komplette System (vgl. Absch. 7.2.1).

Zuerst erfolgt eine übergreifende Betrachtung der Varianten, die dann anschließend mit Hilfe des in Kapitel 6 formulierten Satzes an Kriterien konkretisiert wird.

Abschließend erfolgt exemplarisch eine Untersuchung der Varianten auf die Eignung für das in der Einleitung erwähnte Prozessbeispiel aus der Automatisierungstechnik. Hierdurch wird somit ein Entscheidungsprozess anhand der gefundenen Kriterien bei der Auswahl eines Betriebssystems dargestellt.

8.2 Analyse

Als erstes Kriterium wird das Kriterium *Kosten* untersucht, da sich in diesem Bereich die Aspekte der freien Software - ein Hauptgesichtspunkt dieser Arbeit - besonders bemerkbar machen. In engem Zusammenhang hiermit stehen auch einige Betrachtungen zum Kriterium *Sicherheit*, die hieran anschließen. Der Bereich *Flexibilität* findet als nächster Beachtung. Abschließend wird das Kriterium *Performace* herangezogen, da in diesen Bereich die Echtzeitfähigkeit - wichtig für

die Automatisierungstechnik - fällt.

8.2.1 Kosten

8.2.1.1 Kosten für die Software

Die Kosten in diesem Bereich resultieren u.a. aus dem verwendeten Lizenzmodell des eingesetzten Betriebssystems. Obwohl Linux nicht zum Bereich der proprietären Software gehört, ist es sinnvoll zuerst dieses Modell zu analysieren, um diesem anschließend das Modell der freien Software gegenüberzustellen. Einige der folgenden Schlüsse und Argumente lassen sich nicht direkt in Zahlen ausdrücken, werden aber doch dem Kriterium Kosten zugeordnet, da sie in engem Zusammenhang mit der Lizenzfrage stehen.

8.2.1.1.1 Proprietäre Software Wie bereits erwähnt, erwirbt der Kunde ein Nutzungsrecht an der Software, welches nur die vom Hersteller vorgesehene Benutzung erlaubt. Veränderungen und Übernahme Teile der Software in andere Projekte werden vom Hersteller ausgeschlossen. Da der Kunde die Quellen zu seinen Programmen nicht bekommt, ist er somit nicht in der Lage, Veränderungen vorzunehmen.

Dieses Modell hat seine Vorteile bei Standardsoftware, wie z.B. Textverarbeitung und CAD-Programmen. Es entstehen hohe Entwicklungskosten, die aber durch den Verkauf großer Stückzahlen ausgeglichen werden können. In der Theorie sollte der Hersteller dann auch in der Lage sein, einen entsprechenden Support anzubieten.

Hier fangen aber auch die Schattenseiten dieses Modells an. Die Kundenorientierung der Anbieter ist oft nicht groß. Die Fehlerbehebung geschieht nur spät, obwohl die Fehler bereits lange bekannt sind. Die Hotlines der Hersteller fertigen den Kunden dann mit dem Hinweis ab, dass das Problem bekannt ist und daran gearbeitet wird. Der Kunde wird durch die Entscheidung für eine Softwarelösung langfristig an einen Hersteller gebunden.

Die Folgekosten sind oft erheblich: Updates, um Fehler der Software zu beheben, verlangen in der Regel auch nach neuer leistungsfähigerer Hardware. Das führt dazu, dass für die einfachsten Aufgaben (z.B. das Erfassen von Lagerdaten) schon ein Pentium-III-Rechner eingesetzt werden muss. Die Systemkonstistenz macht dies erforderlich, da die Zusammenarbeit mit den Servern und anderen Rechner sicherzustellen ist.

Kommen dann noch Anforderungen des Kunden hinzu, die außerhalb der Standardsoftware liegen, zeigen proprietäre Modelle erhebliche Schwächen. Die Kosten für Systemanpassungen stehen in keinem Verhältnis zum Arbeitsaufwand.

Kommt es dann noch dazu, dass der Hersteller ein Produkt abkündigt (wie z.B.

Microsoft mit Windows NT 4.0) oder der Hersteller Konkurs anmeldet, steht der Kunde mit leeren Händen da und hat wenig Möglichkeiten, das Produkt noch weiter einzusetzen.

Hier sei folgendes Beispiel aufgeführt, welches sich ähnlich am Fachgebiet Mestechnik der Fakultät Maschinenbau an der Universität Dortmund abgespielt hat [150]: Im Einsatz befindet sich ein hochverfügbarer Linux-Cluster basierend auf einer proprietären Softwarelösung [148]. Zum Betrieb ist ein Hardware-gebundener Schlüssel auf jedem Rechner im Cluster notwendig. Woher sollen die Betreiber den neuen Lizenzschlüssel bekommen, nachdem eine Komponente des Servers nach 2 Jahren wegen eines Hardwaredefekts ersetzt werden musste, der Hersteller der Software aber in der Zwischenzeit Konkurs angemeldet hat. Die Investitionen sind verloren, obwohl das System selbst noch funktioniert. Gerade in der Automatisierungstechnik wiegt dieses Argument schwer, da die Anlagen eine sehr lange Lebensdauer haben.

8.2.1.1.2 Freie Software Die Begriffe *Open Source* und *Freie Software* sind in den letzten Jahren durch den Erfolg von Linux ins Licht der Öffentlichkeit gerückt. Bei aller Euphorie ist aber zu bedenken, dass nicht alleine die Offenlegung der Quellen alle oben beschriebenen Probleme löst. Solange die Rechte zur Verwertung weiter beim Hersteller liegen, kann der Benutzer sich zwar ansehen, warum ein Fehler auftritt und warum etwas nicht funktioniert, er kann es aber nicht selber beheben.

Lizenzen, die Änderungen am Quelltext erlauben, die Verwertung aber dem Hersteller vorbehalten bleibt, sind Wege, die gerade in letzter Zeit von großen Firmen nach Offenlegung ihrer Quellen begangen wurden. Diese führen aber nicht zu Software, die bezüglich Qualität und Funktionsumfang mit der entsprechenden freien Software mithalten kann. Schließlich ist die Motivation für Entwickler gering, Fehler in einem Quellcode zu suchen, der anschließend vom Hersteller kommerziell vermarktet wird.

Vergleicht man diese allgemeinen Aspekte bezüglich *proprietärer* und *Freier Software*, so ergibt sich folgendes Bild (siehe Tabelle 8.1).

Was bedeutet des konkret für den Einsatz in der Automatisierungstechnik? Bei vielen Entscheidern herrscht eine Kombination aus Angst, Unsicherheit und Zweifel bezüglich des Einsatzes Linux-basierter Plattformen in ihren Produkten.

In der Software steckt oft das ganze *Know How* einer Firma und so wird sie in der Regel nicht unter der GPL veröffentlicht. Hier besteht die Furcht, dass durch ein Gerichtsurteil festgelegt wird, die Quellen dieser Software offenzulegen, da sie unter die GPL falle: Durch die Benutzung des Linux-Systems sei die Software mit der GPL "infiziert".

proprietärer Software	freie Software
Vorteile:	**Vorteile:**
• Durch den Lizenzverkauf hat der Hersteller (große) finanzielle Mittel für Wartung und Pflege des Produkts. • Im Optimalfall kann der Kunde sich auf guten Support des Herstellers verlassen. • Kunde braucht sich um technische Probleme nicht zu kümmern.	• Anwender können Code frei für eigene Projekte verwenden, solange sie das Resultat wieder freigeben (Community-Prinzp). • Die Interaktion zwischen Benutzer und Programmierern ist bei freier Software meist sehr gut, da letztere an einer hohen Softwarequalität persönlich interessiert sind. • Investitionsschutz: Verschwindet ein Anbieter vom Markt, kann der Kunde eine andere Firma mit der Fortführung des Projekts beauftragen, ohne Lizenzprobleme zu erwarten. • Gute Unterstützung auch schwächerer Hardware, der Kunde muss nur Migrationsschritte mitmachen, die ihm Vorteile bringen. • Viele Dienstleistungsunternehmen bieten umfangreichen Support für Freie Software.
Nachteile:	**Nachteile:**
• Kunde kann technische Probleme nicht selber lösen oder durch Fremdfirma lösen lassen. • Bei kundenspezifischen Erweiterungen ist man der Preisgestaltung des Herstellers ausgeliefert. • Investitionsschutz ist bei lange laufenden Projekten nicht gewährleistet.	• Die Geschäftsmodelle von Firmen, die auf Freier Software basieren, sind zum Teil nicht unproblematisch. Die Frage nach einem tragfähigen Konzept stellt sich eher als bei proprietären Anbietern.

Tabelle 8.1: Vergleich proprietäre und freie Software

8.2.1.2 Die Lizenzfrage

Der Linux-Kernel unterliegt der GPL (vgl. Absch. 3.4.3.1), allerdings mit einer wichtigen Ausnahme, die in einer Präambel festgehalten wird:

> NOTE! This copyright does ∗not∗ cover user programs that use kernel services by normal system calls - this is merely considered normal use of the kernel, and does ∗not∗ fall under the heading of "derived work".

Diese Präambel besagt, dass das Ablaufenlassen von Programmen auf einem Linux-System nicht als *derived work* aufgefasst wird. Ohne diese Präambel würden alle Programme auf einem Linux-System automatisch der GPL unterliegen und proprietäre Software würde nicht möglich sein.

8.2.1.2.1 Fünf Typen - Die Auswirkungen Genauer betrachtet lassen sich basierend auf der GPL, LGPL und der obigen Präambel fünf Typen unterscheiden, die näher analysiert werden:

User-Space Anwendungen, die unter Linux laufen – Der Status dieses Typs ist klar: Alle Programme im User-Space können proprietär sein, vorausgesetzt, die Anwendung selbst ist nicht von einem GPL-lizensierten Produkt abgeleitet. Dies erklärt auch, warum so viele große Softwarefirmen - wie Oracle und SAP - eine Linux-Version ihrer proprietären Produkte anbieten können. Ein Programm unter Linux auszuführen bedeutet nicht, es auch unter der GPL veröffentlichen zu müssen.

Ladbare Module, die das normale Kernelinterface benutzen – Die Lage bezüglich dieses Typs ist nicht ganz so klar. Das Laden eines Moduls ist eine Form des dynamischen Linkens. Wenn geladen, wird das Modul gegen den GPL-lizensierten Kernel gelinkt und Dateien, die gegen GPL-Code gelinkt werden, müssen ebenfalls unter der GPL lizensiert werden. Hieraus könnte man schließen, dass Kernel-Module, einschließlich Treiber, GPL lizensiert sein müssen. Torvalds hat sich jedoch positiv gegenüber proprietären Treibern, die das normale Kernelinterface benutzen, geäußert. Die Kernel Entwickler haben dies - wenn auch widerwillig - akzeptiert und dulden die Existenz von proprietären Treibern.

Diese Toleranz hält schon eine ganze Zeit und viele Hardware-Hersteller mit ihren Treibern sind davon abhängig. Es ist allerdings klar, dass die Inhaber des Linux-Copyrights nicht sehr begeistert hiervon sind. Ein nicht zu vernachlässigender Nachteil dieser Politik ist auch, dass in Binärform vorliegende Module in der Regel an eine Kernelversion gebunden sind. Wird ein neuerer Kernel verwendet, so müssen vom Hersteller auch neue Module bereit gestellt werden.

Man sollte sich vorher also genau überlegen, ob man diesen Weg einschlagen möchte.

Ladbare Module, die nicht das normale Kernelinterface benutzen – Der dritte Typ ist der problematischste. Wenn einmal geladen, wird ein Modul zu einem wesentlichen Teil des Kernels. Im Prinzip ist es möglich, ein Linux so zu modifizieren, dass es ein spezielles Interface für ein Modul bereitstellt und dieses modifiziere Linux unter der GPL zu veröffentlichen. Passend hierzu wird ein Modul entwickelt, welches dieses Kernelinterface nutzt. Das Resultat wäre ein proprietäres Linux, welches in krassem Gegensatz zum Geist der GPL steht - weit außerhalb der Ausnahme Torvalds.

Eigenständige Module – Kernel Module sind sehr flexibel und haben ein breites Anwendungsspektrum. In manchen Fällen werden sie unabhängig von der Kernelfunktionalität eingesetzt. RT-Linux und RTAI (vgl. Absch. 7.4.1 und 7.4.2) sind hierfür Beispiele. Ihre Module modifizieren den Kernel in keiner Weise, es sind einfache Applikationen, die im Kernel Space laufen. Diese können wie der zweite Typ behandelt werden: Module die nicht in den Kernel eingreifen, können auch nicht auf ein nicht-standard Interface zugreifen.

Modifikationen an Linux selbst – Diese fallen natürlich unter die GPL. Modifikationen am Kernel und anschließendes Compilieren führen zu einer Arbeit abgeleitet von Linux und müssen deshalb unter der GPL veröffentlicht werden.

Trotz aller Unsicherheit ist die Entwicklung für Linux aus lizenztechnischer Sicht nicht kompliziert. Man sollte keine Module entwickeln, nur um das Linux Copyright herumzukommen. Wenn man wesentliche Funktionalitäten entwickelt, sollte man sie unter der GPL veröffentlichen.

Die Geschichte beweist, wenn man im Sinne der GPL entwickelt, läuft man nicht in Gefahr, Probleme mit den Copyright Inhabern zu bekommen. Viel mehr Aufmerksamkeit sollte das Problemfeld der Softwarepatente erhalten (vgl. Absch. 8.2.1.2.2).

Abschließend läßt sich sagen, dass die GPL von Linux selten ein Anlaß von Problemen für die Anwendungsentwicklung in der Automatisierungstechnik ist. Die Schlüssel-Technologie ist entweder die Hardware oder die Anwendungssoftware - beide können ohne Probleme proprietär gehalten werden.

In den seltenen Fällen, in denen der Treibercode nicht veröffentlicht werden soll, kann man sich immer noch in die "Grauzone" der Linux-GPL-Lizenz begeben und Module verwenden, die nicht im Quelltext vorliegen. Dieser Schritt sollte allerdings sehr genau überlegt sein: Ist es wirklich notwendig, die proprietäre Funktionalität in ein Modul zu binden oder gar den Kernel selbst zu modifizieren?

Wenn die Antwort "Ja" lautet: Wie groß ist der Schaden für die Firma wirklich, die Funktionalität mit der Gemeinschaft der Benutzer zu teilen? In den meisten Fällen können sowohl die Firma als auch die Gemeinschaft hiervon nur profitieren.

Das bisher Gesagte gilt grundsätzlich für alle unter der GPL stehenden echtzeitfähigen Linuxvarianten. Da aber alle Varianten - außer RTAI - keine reinen Open Source Projekte sind, begibt man sich immer in eine gewisse Abhängig von einem Hersteller, mit den beschriebenen Vor- und Nachteilen.

Allerdings ist auch RTAI nicht ganz unproblematisch.

8.2.1.2.2 Die Auswirkung von Softwarepatenten Hier werden die bereits gestellten Fragen im Zusammenhang mit dem RT-Linux Patent beantwortet. Auch die Auswirkungen auf RTAI werden analysiert.

Wie hängt das Patent mit der GPL zusammen? Victor Yodaiken hat - wie in Abschnitt 7.4.1.4 erwähnt - die hinter RT-Linux stehende Methode des Betriebssystems, das als Idle-Task eines Echtzeitsystems läuft, durch ein US-Patent schützen lassen. Momentan ist unklar, was dies für Folgen für die Entwicklung von proprietärer Software auf RT-Linux-Systemen hat. Der RT-Linux-Kernelpatch steht laut Lizenzdokumenten von RT-Linux unter der GPL. Für Freie Software ergeben sich also keine Probleme im Zusammenhang mit der Patentfrage.

Für proprietäre Software sieht die Lage anders aus. FSMLabs vergibt gegen Gebühr Lizenzen, die die Offenlegung des Quellcodes nicht erzwingen. Hieraus ergeben sich wiederum die bereits beschriebenen Probleme bezüglich des Investitionsschutzes. Was passiert, wenn FSMLabs den Support für RT-Linux einstellt oder aufgekauft wird, ist unklar.

Ist das Patent durchsetzbar? Grundsätzlich ist die Frage noch nicht geklärt, ob das Patent überhaupt Bestand hat. Aus Entwicklersicht sind die zwei in Abschnitt 7.4.1.4 beschriebenen Techniken nicht wirklich neu und eigentlich offensichtlich:

"OS on OS" Bereits in der Vergangenheit hat es viele Fälle gegeben, in denen ein Betriebssystem (*OS*) unter einem anderen Betriebssystem zur Ausführung kommt. Um ein Beispiel zu nennen: iRMX für Windows 3.1. Dies ist ein Echtzeitbetriebssystem, welches Windows als einen Task ausführt [110]. Oder RMOS von Siemens, welches ebenfalls diese Möglichkeit nutzt [105].

Diese Technik kann als offensichtlich bezeichnet werden - und ist somit nicht patentierbar. Es ist offensichtlich das:

- Betriebssysteme Programme ausführen.

- Ein Betriebssystem selbst ein Programm ist.

So ist die Kombination dieser beiden Tatsachen genau so offensichtlich:

- Es ist möglich, ein Betriebssystem als Programm in einem anderen auszuführen.

Das Abfangen der Interrupts Auch wenn dieser Ansatz nicht so offensichtlich ist, so ist er auch nicht unbedingt neu: In [26] wird genau diese Methode beschrieben. Das Auftreten von Interrupts wird akzeptiert, der entsprechende Handler aber erst später ausgeführt.

Ein weiterer interessanter Gesichtspunkt ist, dass genau dieses Verfahren in Michael Barbanovs *Master Thesis on RT Linux* beschrieben wird [7]. Barabanov ist der ursprüngliche Entwickler von RT Linux und arbeitet jetzt für FSMLabs – sic.

In seiner Arbeit weist Barabanov auf das Paper von Stodolsky hin, betont aber gleichzeitig, dass die Motivation beim Abfangen der Interrupts eine andere sei: Performance. Dies würde bedeuten, dass *Motivation* patentierbar ist?!

Zusätzlich an dieser Stelle zwei Zitate von Paolo Montegazza:

> Mr. Yodaiken has only been allowed to patent air, but air has been arround forever with nobody thinking to patent it.

Als Antwort auf die Frage, wie RTAI-Benutzer auf das Patent reagieren sollen:

> ..., they should act as if the patent did not exist. Remember that the patent is only valid in the USA, and the USA is not the world. Plus...the patent could also vanish like a soap bubble at the first legal test.

Ob daher die RT-Linux-Methode genug Alleinstellungsmerkmale aufweist, um einer rechtlichen Überprüfung des Patents standzuhalten, wird die Zukunft zeigen. Wenn es Gültigkeit hat, ist es nur in den USA gültig.

Als Konsequenz aus der Unsicherheit bezüglich des Patents kann gesagt werden, dass RTAI für den Einsatz in der Automatisierungstechnik die bessere Wahl ist:

- RTAI ist ein "echtes" Community-Projekt. Paolo Montegazza, der Kopf dieses Projekts, ist sehr an Weiterentwicklungen und Verbesserungen interessiert. Die Entwickler arbeiten sehr gut zusammen.

- Da RTAI unter der GPL steht, ist die Situation bezüglich der Entwicklung von proprietären Applikationen klar. Es ist möglich Anwendungen mit geschlossenem Quellcode zuentwickeln.

  Gleichzeitig hat man aber auch die volle Kontrolle über die Quellen des Betriebssystems, ist also in der Lage, Fehler oder Sicherheitslücken - hierzu im Abschnitt 8.2.2 mehr - eigenständig zu beseitigen.

Als Resultat dieser unklaren Sachlage gibt es nur wenige Firmen, die Echtzeitsysteme unter Linux entwickeln, obwohl beide Varianten ein großes Potential bieten [139]. Es zeigt sich deutlich, wie schädlich Software-Patente im Bereich der Freien Software sein können.

8.2.1.3 Kosten für den Support

Ein ganz wichtiger Punkt bei der Entwicklung von Automatisierungssystemen ist der Support. Sei dies durch den Hersteller des Betriebssystems oder einen Dienstleister. Auch im Linux-Umfeld gibt es diese bekannte Unterstützung. Firmen wie MontaVista oder TimeSys - als Entwickler einer echtzeitfähigen Linuxvariante - oder Firmen wie Lineo oder Ingenieurbüros bieten kommerziellen Linux-Support an. Diese Option unterscheidet sich wenig von proprietären Produkten wie WindowsCE oder QNX.

Der interessante Aspekt ist allerdings bei den komplett freien Varianten wie KURT oder RTAI die Unterstützung durch die Entwickler und Nutzergemeinschaft. Hierbei stellen die Newsgroups und Mailinglisten das mächtigste Instrument für den Support da. Bei Anfragen an den Support des Herstellers proprietärer Software kann es mitunter sehr lange dauern, bis eine Lösung zu einem Problems geliefert wird, wenn denn überhaupt eine Lösung angeboten wird.

Bei Freier Software waren traditionell die Kommunikationswege immer kurz. Der Anwender kann direkt mit dem Autor der Software per E-Mail Kontakt aufnehmen und ihm die auftretenden Probleme schildern. Auch die Unterstützung durch Newsgroups ist ein Vorteil freier Software. In diesen Gruppen diskutieren Anwender und Entwickler Probleme und Wünsche bezüglich der Software. Bei einer Anfrage in eine Newsgroup meist innerhalb eines Tages eine Lösung da:

> Sure, I gotta compile it in, but at least I get an answer today! [49]

Ein weiterer Vorteil ist, dass man auch in diesen Foren direkt mit den Entwicklern in Kontakt treten kann, da diese die Gruppen selbst verfolgen, um Fehler und Erweiterungswünsche berücksichtigen zu können. Der "Vater" von RTAI - Prof. Paolo Mantegazza beantwortet einen Großteil auf der RTAI-Mailingliste diskutierten Probleme selbst [104]. Einen kompetenteren Support kann man sich nicht wünschen. Zusätzlich ist dieser auch noch kostenlos.

Ressource	Controllerlösung	Linux-Lösung
Prozessor	68HC11 8bit	486/32bit
Takfrequenz	2Mhz	16Mhz
RAM	128Bytes	4MB
Flash-ROM	2kBytes	1MB

Tabelle 8.2: Vergleich der Ressourcenanforderungen für ein echtzeitfähiges Microkontroller- und Linux-System

Auch Firmenanfragen werden bereitwillig beantwortet, wenn sich die Firma zum Open Source Gedanken bekennt und selbst einen Beitrag zur Entwicklung leistet ist dies kein Problem. Wie dieser Beitrag aussieht - Spenden, Entwicklerzeit oder auch nur eigene Mitarbeit in den Foren - spielt dabei keine Rolle. Es wird nicht verlangt, Firmen Know How preiszugeben, sondern nur dort zu helfen, wo es möglich ist.

8.2.1.4 Kosten für die Echtzeit

In der Welt der Microkontroller war harte Echtzeit nie ein Thema: Es wird eine Interruptroutine in Verbindung mit einem Hardware-Timer eingesetzt. Bei Ablauf dieses Timers wird diese Routine auf jeden Fall ausgeführt und kann zeitkritische Aufgaben erledigen.

Allerdings wachsen die Anforderungen an solche Systeme ständig und durch die Anforderungen nach Netzwerkfähigkeiten, Benutzerschnittstellen usw. wurde es interessant, diese Aufgaben mittels eines echtzeitfähigen Bertiebssystem zu erledigen. Es ist allerdings offensichtlich, dass der Mehraufwand an Ressourcen gegenüber einer klassischen Microkontrollerlösung - hierunter fällt auch eine SPS - nicht zu vernachlässigen sind. Für die Minimalanforderungen ergeben sich die in Tabelle 8.2 aufgeführten Werte [138].

Ausgehend von diesen Ressourcenanforderungen lässt sich direkt der Preis ableiten. Die Kontroller basierte Lösung hat einen Preis von ca. 50 Euro, die Komponenten für das Linux-System liegen im Bereich von einigen hundert Euro. Beim Vergleich ist allerdings zu beachten, dass der Nachteil der höheren Kosten beim Linux-System eine wesentlich höhere Flexibilität bei der Software mit sich bringt - dies ist schließlich der Grund für den Einsatz eines solchen Systems.

8.2.2 Sicherheit

Probleme mit der Performance eines Systems in der Automatisierungstechnik werden in Zukunft immer mehr hinter Probleme bezüglich der Sicherheit zurück-

treten. Die verstärkte Nutzung lokaler und auch öffentlicher Netze für Aufgaben - wie Wartung und Überwachung der Systeme - wirft die Frage nach der Sicherheit dieser Kommunikation auf [137]. Gleichzeitig ist aber auch die innere Sicherheit des Systems wichtig, gerade der Einsatz von Kernelmodulen wirft hier einige Probleme auf, die im folgenden analysiert werden. Daran anschließend folgt die Analyse bezüglich der äußeren Sicherheit.

8.2.2.1 Innere Sicherheit

Ein Grund für den Einsatz eines Betriebssystems wie Linux sind das Vorhandensein von Mechanismen für den Speicherschutz, das Ressourcen-Managment und die Isolation von Prozessen. Diese ermöglichen die Entwicklung von komplexen und zugleich robusten Systemen für die Automatisierungstechnik.

Beim Einsatz eines Systems mit präemptiven Kernel (z.B. Montavista-Linux) können diese Mechanismen ohne Probleme benutzt werden, anders sieht dies beim Ersetzen des Kernels aus.

Ersetzen des Kernels Bei diesem Modell tritt das Problem auf, dass man, um echtzeitfähige Applikationen zu erhalten, Kernelmodule implementieren muss. Die oben erwähnten Mechanismen existieren aber im Kernel-Space nicht. Auf den ersten Blick scheint dies ein ganz entscheidendes Kriterium gegen dieses Modell zu sein. Genauere Überlegungen zeigen, dass sich dieser vermeintliche Nachteil zu einem gewissen Maße in einen Vorteil wandelt: Es wird übersehen, dass das Modell nicht dazu zwingt, die gesamte Anwendung im Kernel-Space laufen zu lassen.

Es ist absolut nicht notwendig, alles in den Kernel zu verlagern. Die Natur komplexer Systeme fordert nicht von jedem Bestandteil Echtzeitverhalten. Das in Abbildung 2.6 gezeigte System hat sicher im Bereich System-Kontrolle harte Echtzeitanforderungen. Die Bereiche Datenbank, Fernwartung und Anzeige sind aber ohne harte Echtzeitanforderungen. Diese Teilung der Anforderung ist für solche Systeme typisch.

Weiterhin ergibt sich nicht die Folge, dass echtzeitfähige User-Space Applikationen einfacher zu entwickeln sind [23][24][28]. Genauso wenig können sie ohne Einschränkungen auf das volle Linux-POSIX-API zugreifen. Für solche Applikationen sind Netzwerkzugriffe, Festplattenzugriffe, Speicherallokation und andere Dienste, die keine garantierte Antwortzeiten bieten, nicht nutzbar. Es ist also zwingend, die Bereiche die echtzeitfähig sein müssen, von den restlichen zu trennen. Es ist also weniger ein Problem, ob der Echtzeit-Bereich einer Applikation im User- oder Kernel-Space liegt, sondern viel problematischer ist es zu gewährleisten, dass die Echtzeitanforderungen überhaupt eingehalten werden.

Wenn man sich an folgende Regeln zum Umgang mit Kernelmodulen hält, erhält

man ein sicheres System. Das Modul muss von einem priviligierten Benutzer geladen werden und natürlich während der Benutzung überwacht werden.

- Laden des Moduls beim Starten des Sytsems
- Einrichten eines Kommunikationskanals (FIFO), mit der Erlaubnis für unpriviligierte Benutzer auf diesen zuzugreifen.
- Senden von Kommandos durch diesen Kanal als unpriviligierter Benutzer.
- Überprüfen des Kommandos und der Argumente auf Gültigkeit.
- Protokollieren dieser Ereignisse mit Zeitstempel und benutzerbezogener Information durch die Protokollfunktionen des Systems (vgl. Absch. 8.2.2.2).
- Genaue Dokumentation des Verhalten des Moduls, damit Abweichungen von diesem Verhalten gefunden und analysiert werden können.

Hält man sich an dieses Schema, so kann man dieses Modul mit einem guten Maß an Sicherheit benutzen. Viele der angesprochenen Aspekte gelten aber auch für die normale Programmentwicklung - z.B. die Überprüfung der Gültigkeit - und führen so nicht zu einem vertretbaren Mehraufwand.

Der etwas höhere Lernaufwand für die Entwicklung von Kernelmodulen ist gering im Vergleich zum benötigten Wissen und der Erfahrung, die für das richtige Design und die Programmierung des Echtzeitalgorithmus notwendig sind. Die Notwendigkeit, den Echtzeitteil als Modul in den Kernel-Space zu legen kann zu einem klaren Design führen, in dem Echtzeit- und nicht Echtzeitteile der Applikation strikt voneinander getrennt sind - dies kann nur als Vorteil gelten.

8.2.2.2 Äußere Sicherheit

GNU/Linux-Systeme sind bestens geeignet für sicherheitskritische Computer-Systeme. Die in Linux integrierten Sicherheitsmechanismen werden täglich von *"script kiddies"* und professionellen Hackern ausgetestet. Dies ist natürlich nicht die angenehmste Art der Sicherheitsüberprüfung, aber sicher eine der effektivsten. Ein System welches nur in ein paar hundert oder tausend Projekten eingesetzt wird, ist sicher nicht so stark auf Schwächen untersucht worden, wie ein GNU/Linux basiertes System.

Von diesen Fakten profitieren auch GNU/Linux basierte Systeme in der Automatisierungstechnik, da sie die gleiche Codebasis benutzen. Die Kombination aus intensivem Testen und die gleichzeitige Transparenz der Sicherheitsmechanismen - durch die offenen Quellen - machen Linux zu einer hervorragenden Basis für Applikationen mit hohen Ansprüchen an die Sicherheit.

Linux stellt folgende Sicherheitsmechanismen zur Verfügung [86]:

- Firewall- und Paketfilterfunktionen [18].
- Erkennen von Einbruchsversuchen (*intrusion detection*) ins System.
- Zugriffskontrolle auf Systemressourcen.
- Sichere Netzwerkdienste (z.B. SecureShell [123]).
- Gut konfigurierbare Protokollfunktionen.

Nimmt man diese Fähigkeiten zusammen, ergeben sich nicht nur die Möglichkeit im System ablaufende Vorgänge zu protokollieren und bei Bedarf einzugreifen, sondern es ist auch möglich Tendenzen zu erkennen und einzugreifen, bevor ein Versagen des Systems auftritt. Ein Großteil von Hardwaredefekten treten nicht abrupt auf, sondern kündigen sich vorher an. Wenn man ein Betriebsystem hat, welches hierfür Unterstützung anbietet, führt dies sowohl zur Verbesserung der Stabilität, Verfügbarkeit und Systemsicherheit.

8.2.3 Flexibilität

8.2.3.1 Unterstützte Prozessoren und Skalierbarkeit

Alle vorgestellten Varianten unterstützen mehrere Architekturen und Plattformen. Ursprünglich nur für die Intel-386-Architektur verfügbar, stehen heute alle Varianten auf mehreren Prozessorfamilien zur Verfügung. Darunter befinden sich etliche (PowerPC oder 486-Systems-on-Chip), die den Betrieb mit passiver Kühlung zulassen. Solche Prozessoren lassen sich mit relativ geringem Aufwand in Gehäuse mit einer hohen Schutzklasse einbauen und eignen sich somit für den Einbau direkt im Feld, ohne aufwendige Kühlungs- und Lüftungsmaßnahmen.

Ein weiterer Pluspunkt der aus der Verfügbarkeit für die unterschiedlichen Prozessorarchitekturen resultiert, ist die Skalierbarkeit. Betrachtet man z.B. die Intel x86-Familie, so werden in dieser am unteren Ende Systeme mit 486-CPU und Taktfrequenzen von $33 - 100 Mhz$ unterstützt. Diese billigen und stromsparenden Systeme werden in großen Stückzahlen für industrielle Anwendungen gebaut.

Am oberen Ende der Intel x86-Familie finden sich Industrie-PCs mit aktuellen Pentium-4 Prozessoren mit Taktfrequenzen von $> 1,7 Ghz$. Auch Doppel- und Mehrfachprozessorboards werden unterstützt.

8.2.3.2 Konnektivität

Dieses Thema wird unter Linux groß geschrieben. Als Unix-Derivat bringt das System volle Unterstützung für Ethernet und das TCP/IP-Protokoll mit.

8.2.3.2.1 Hardwarekomponenten Es werden nahezu alle am Markt befindlichen Netzwerkkarten unterstützt. Gleiches gilt auch für Hochgeschwindigkeitsnetze wie Giga-Ethernet.

Interessant für industrielle Anwendungen ist neben Ethernet vor allem die Möglichkeit zum Anschluss von Feldbuskomponenten. Inzwischen gibt es eine Reihe von Herstellern, die für ihre Karten Treiber für Linux zur Vefügung stellen. So können bereits CAN, Profibus und RS485 unter Linux betrieben werden. Allerdings bieten noch lange nicht alle Hersteller hier Treiber an, so dass gegebenen Falls nachgesehen werden muss, ob ein spezielles Board auch unterstützt wird. Eine gute Übersicht bietet [121].

Ähnliches gilt auch für digitale Ein/Ausgabekomponenten. Die Unterstützung der Hersteller nimmt in letzter Zeit rapide zu, jedoch kann nicht davon ausgegangen werden, dass alle Boards bereits unterstützt werden. Für Industriekameras und Framegrabber gilt dies ebenfalls.

Bei den Treibern der Hersteller kommt es häufig vor, dass einfach bestehende Treiber nach Linux portiert werden, ohne auf die spezifischen Vorteile der Plattform Rücksicht zu nehmen. Trotzdem sind schlecht portierte Treiber - wenn sie im Quelltext vorliegen - oftmals eine große Hilfe, da es mit ihrer Hilfe leicht möglich ist, ein eigenes Kernelmodul zu entwickeln. Es ist zu erwarten, dass sich der Trend der Firmen, Linux als gleichberechtigte strategische Plattform in der Automatisierungstechnik anzusehen, weiter fortsetzen wird.

8.2.3.2.2 Software Systeme in der Automatisierungstechnik benötigen nicht nur die Möglichkeit zur Fernadministration sondern in vielen Fällen (z.B. das Beispiel aus der Einleitung) auch die Möglichkeit zur Einspielung von Updates und Bugfixes über das Netzwerk. Linux bietet hierfür Unterstützung, die weit über die anderer Systeme (wie z.B WindowsCE oder QNX) hinausgeht. Hier sei nochmal auf die Wurzeln im Unix-Umfeld, mit den damit verbundenen Fähigkeiten im Bereich Netzwerk, hingewiesen.

Bei der Entwicklung ist oft die Reduzierung von Kosten eines der Hauptziele. Ein bedeutender Faktor für Systeme in der Automatisierungstechnik sind die Wartungskosten. Nicht nur die direkten Kosten für die Wartung der Hardware, sondern die Kosten verursacht durch Ausfallzeiten und die Updates der Software. Diese Kosten können reduziert werden, wenn das System geeignete Mechanismen bietet, die die Administration aus der Ferne ermöglichen, dies war ja auch ein ausschlaggebender Grund für das Zustandekommen dieser Arbeit.

Ein GNU/Linux basiertes System stellt die geforderten Mechanismen zur Verfügung:

- Statusüberwachung des Systems (Web-Interface oder die Protokollierung an zentraler Stelle).

- Gesicherter Fernzugriff auf das System erlaubt einen vollständigen Zugriff auf das System [123].
- Benachrichtigung des Personals über die verschiedensten Medien, je nach Dringlichkeit der Meldung (SMS, Mail oder Pager).
- Die Möglichkeit das System über das Netzwerk upzudaten, wahlweise das ganze System oder auch nur einzelne Komponenten.

Es stehen hierfür ausgiebig getestete Server und Clienten mit verschlüsselter Übertragung bereit. Als Web-Server muss auch nicht der komplexe *Apache-Server* eingesetzt werden [101]. Vielmehr existieren kompakte Lösung, entwickelt speziell für kleine System, z.B. der *Boa*-Web-Server [102].

Die ausgehenden Verbindungen sind unter GNU/Linux ebenfalls kein Problem, nutzbar sind neben den gebräuchlichen Netzwerktypen - vergl Absch. 8.2.3.2.1 - natürlich auch Wählverbindungen per Modem.

Ausleuchten des Potentials Die aufgeführte Liste an Möglichkeiten führt eine Reihe von Punkten auf, mit denen es möglich ist, ein System aus der Ferne zu Warten und Administrieren. Dies ist aber noch nicht alles, was GNU/Linux-System im Umfeld Netzwerk leisten kann. Ein solches System kann auch Dienste bereitstellen, die eine Einwahl per Modem ermöglichen. Das **N**etwork**F**ile**S**ystem kann dazu benutzt werden, von zentraler Stelle aus das Automatisierungssystem mit neuer Software zu versorgen oder die Protokolle auszulesen, indem das Dateisystem des Automatisierungssystems einfach auf dem Administrationssystem eingebunden wird. Es ergeben sich unbeschränkte Möglichkeiten des Zugriffs auf das System - alle in einer sicheren Weise per verschlüsselter Verbindung. Dies alles mit ausgiebig getesteter und weitverbreiteter Software ohne den Bedarf nach einem speziellen - wahrscheinlich einzeln zu lizensierenden und zu bezahlenden - Softwarepaket eines Anbieters.

8.2.3.3 Entwicklungswerkzeuge

Linux basiert, wie in Abschnitt 5.3 beschrieben, auf dem GNU-Werkzeugsatz. Dieser bietet einen sehr kompletten und nahtlosen Satz an Werkzeugen auch für die Cross-Plattformentwicklung.

Analysiert man die beiden Alternativen - Kommandozeile und integrierte Entwicklungsumgebung - bezüglich ihres Potentials, so ist natürlich die integrierte Entwicklungsumgebung das komfortablere und leistungsfähigere Werkzeug. Aber auch die Kommandozeile bietet einen nicht zu unterschätzenden Vorteil: Sie benötigt keine grafische Oberfläche, deshalb ist sie sehr gut geeignet, auch über schmalbandige Netzwerkverbindungen Modifikationen am Code vorzunehmen. Denkbar ist auch der Fall, in dem das System im Büro des Entwicklers

auf einem leistungsfähigen Rechner neu compiliert wird und anschließend auf das Zielsystem übertragen wird - das Ganze gesteuert nur über eine Modemverbindung mit nur 9600*baud* Übertragungsrate.

8.2.3.4 Software-Entwicklung

Neben den zu Verfügung stehenden Werkzeugen, sind aber auch Konsequenzen für die Entwicklung von Echtzeit-Applikationen interessant, die sich aus der verwendeten Architektur der Linux-Variante ergeben. Hier bietet sich eine zweigeteilte Betrachtung an:

8.2.3.4.1 Ersetzen des Kernels Es ergeben sich aus diesem Ansatz aber auch signifikante Auswirkungen für Entwickler: *ein eigenes Programmiermodell für Echtzeittasks* - Es ist notwendig, die Echtzeit-Tasks als Kernelmodule zu entwickeln. Diese benutzen dasselbe Format, wie die bekannten Module für Dateisysteme oder Treiber. Kernelmodule benutzen eine eigene API - im Gegensatz zum POSIX API, verfügbar für die Entwicklung normaler Linux-Prozesse. Dieses Interface ist aber zum großen Teil identisch mit dem bei der Treiber-Entwicklung verwendetem und auch ausgiebig dokumentiert (z.B. [29] [18]).

Von beiden Varianten wird auch eine Unterstützung für Fließkomma-Operationen im Kernel Modus angeboten. Dies ist deshalb von besonderer Bedeutung, da genau wie die Speicherschutzmechanismen, Methoden zur Fließkommabehandlung im Kernel-Modus nicht zur Verfügung stehen. Der Kernel sorgt zwar bei einem Prozesswechsel im User-Space für die Sicherung der Fließkommaregister, im Kernel Modus geschieht dies nicht. Beide Varianten bieten hierfür ein API an, welches die Funktionalität bereitstellt und für das Sichern der Register sorgt. Somit wird auch von diesen Varianten die geforderte Behandlung von reellen Zahlen unterstützt.

Die speziellen APIs die für die Handhabung der Echtzeitumgebung zuständig sind, sind neu im Linux Kernel. Hier gibt es die Auswahl zwischen der RT-Linux bzw. RTAI spezifischen API oder es ist unter beiden Varianten möglich, den POSIX-Standard zu benutzen.

Ein Ausweg aus dem beschriebenen Programmiermodell bietet RTAI in Form von LXRT: Dies ermöglicht die Entwicklung von Echtzeit-Tasks im User-Space. LXRT unterstützt das gleiche API wie RTAI, aber die Tasks laufen im User-Space. LXRT kann harte Echtzeit bieten, solange der LXRT-Task keine Linux-Systemaufrufe tätigt. LXRT wird hier von verbesserten Linux-Kernel Versionen profitieren - im Ausblick (Kapitel 9) zu diesen Versionen mehr. Begründen lässt sich dies mit den Problemen bezüglich der Nichtunterbrechbarkeit der Systemaufrufe in den aktuellen 2.2/2.4er Kerneln.

Für RT-Linux ist Echtzeitfähigkeit im User-Space für die kommende Version

3.1 angekündigt.

Es empfiehlt sich allerdings ein genauerer Blick auf die Entwicklung von Kernelmodulen. Natürlich erfordert die Entwicklung von Kernelmodulen andere Programmierfähigkeiten als das Entwickeln normaler (Linux-)Applikationen. Diese Fähigkeiten sind aber bei Entwicklern von echtzeitfähiger Software vorhanden, da diese sich sehr intensiv mit dem System und so auch den Hardwaretreibern, die als Modul realisiert sind, beschäftigen.

8.2.3.4.2 Präemptiver Kernel Für den Anwendungsentwickler ist dieser Ansatz unproblematischer, da die Echtzeit-Anwendungen grundsätzlich wie normale Anwendungen für den User-Space entwickelt werden - die Aussagen zum Softwaredesign gelten hier gleichermaßen. Es ergeben sich hier größere Konsequenzen für die Kernel-Entwickler.

Während die Änderungen am standard Linux Kernel bei RT-Linux und RTAI marginal sind, sind die Eingriffe in den Kernel bei diesem Ansatz komplizierter und damit unter Umständen auch schwerer zu warten.

Hier unterscheiden sich die beiden Ansätze - vollständig präemptiv und das Einfügen von *Preemption Points* - deutlich bezüglich der Wartbarkeit und damit der Implementation in neueren Kernelversionen.

Beim Einfügen von *Preemption Points* muss der Kernelcode genau analysiert werden und an den entsprechenden Stellen Modifikationen vorgenommen werden. Dies hat aber einen entscheidenden Nachteil: Beim Hinzufügen neuer Codebausteine, z.B. eines neuen Treibers, muss darauf geachtet werden, dass in diesem keine Teile enthalten sind, die große, nicht unterbrechbare Bereiche enthalten. Dies ist bei Treibern, die nicht im Quelltext vorliegen nicht feststellbar und auch nicht änderbar.

Der vollständig unterbrechbare Kernel nutzt Mechanismen für das SMP. Diese finden sich auch in neueren Kernelversionen wieder. Außerdem profitiert dieser Ansatz zugleich durch die Verbesserung dieser Mechanismen in neueren Versionen. Da diese ganz wichtig für die Skalierbarkeit in Multiprozessorumgebungen sind, sind hier weitere Entwicklungen zu erwarten.

Ein weiter Punkt ist, dass durch diesen Ansatz Code unterbrechbar wird, der hierfür nicht gedacht war. Die Entwickler von Treibern müssen also nichts tun, um ihren Treiber unterbrechbar zu machen. Der Code des Treibers wird, wenn benötigt, einfach unterbrochen. Das bedeutet, dass Treiber, die für SMP-Umgebungen entwickelt wurden, automatisch von der Unterbrechbarkeit profitieren. Hier liegt auch der Nachteil verborgen: Code der nicht für SMP-Umgebungen gedacht ist, wird wahrscheinlich auch nicht korrekt mit einem präemptiven Kernel funktionieren.

8.2.4 Performance

Wie bereits in Abschnitt 7.3 beschrieben gibt es grundsätzlich vier Wege zur Echtzeitfähigkeit. Welche Variante welche Vor- und Nachteile und damit für welche Einsatzgebiete geeignet ist, wird im Folgenden dargestellt. Es erfolgt demnach zusammenfassend eine Beschreibung *wie* die Echtzeitfähigkeit erreicht wird anhand derer analysiert wird *wie viel* Echtzeitfähigkeit erreicht wird.

Beim ersten Ansatz ist klar, dass mit ihm nur weiches Echtzeitverhalten erreicht werden kann. Es ist nicht vorhersagbar, wie - oder besser - wann das System auf ein Ereignis reagiert. Der zweite Weg, Optimierung von Kernel und Treibern ist bei komplexeren Aufgabenstellungen nicht gangbar, da die Komplexität die Optimierung des Systems erschwert bzw. unmöglich macht.

8.2.4.1 Ersetzen des Kernels

Die dritte Variante "Ersetzen des Kernels" oder auch *Interrupt Abstraction* genannt, belässt den Kernel-Code fast unverändert. Der Kernel läuft also ganz normal auf diesem System und damit auch alle Subsysteme wie Treiber, Benutzerprogramme usw. Für die Aussagen bezüglich der Echtzeitfähigkeit ist also keine Analyse des Kernelquelltextes notwendig. Allein der Hardware Abstraction Layer - hier stellvertretend für RT-Linux und RTAI - und damit verbunden der Echtzeit-Scheduler müssen analysiert werden. Dieser Code hat in beiden Varianten einen Umfang von ca. 64K. Ein großer Teil dieses Code ist auf Grund der Modulstruktur optional, so dass der Umfang des zu analysierende Code auf ca. 30K sinkt. Dieser Code wurde bereits von einer großen Anzahl an Experten untersucht und bietet in beiden Varianten harte Echtzeit [15].

Bei der Nutzung von RT-Linux oder RTAI benutzt man praktisch ein zweites Betriebssystem als Ergänzung zu Linux, dieses bietet deterministische Antwortzeiten in der Größenordnung von 15 Microsekunden [32].

8.2.4.2 Präemptiver Kernel

Bei diesem Ansatz wird der Linuxkernelcode mit dem Ziel modifiziert, die Zeit, in denen der Kernel sich in nicht unterbrechbaren Bereichen befindet, zu minimieren. Hierdurch soll die Verzögerung beim Auftreten von Interrupts und beim Aktivieren von Echtzeittasks minimiert werden.

Hierbei gibt es zwei Ansätze, Montavista und TimeSys modifizieren den Kernel derart, das er vollständig preämptiv wird. RED Linux und die Weiterentwicklung REDICE Linux fügen nur Kontrollpunkte (*preemption points*) in den Kernel ein, an denen wird geprüft, ob es wartende Echtzeit-Prozesse gibt und diese bei Bedarf ausgeführt.

Im Gegensatz zur vorher beschriebenen Technik, ist es bei diesen Varianten unmöglich, Garantien bezüglich der Antwortzeiten zu geben. In der Literatur findet man deshalb nur Aussagen wie *"the longest measured latency"* [157]. Der Grund hierfür liegt darin, dass es nicht möglich ist, jeden Ausführungspfad im Kernel zu analysieren und so lange dies nicht geschieht, ist es nicht möglich, eine Antwortzeit zu garantieren. Die Aussagen bezüglich der Antwortzeit solcher Systeme beruht also immer auf Messungen und nicht auf einer kompletten Analyse des Ausführungspfades.

Ein normaler Kernel hat einen Umfang von 300K bis 500K. Der Quellcode für ein kompletten Linuxkernel hat einen Umfang von $> 100 MByte$. Da der Quellcode ständigen Änderungen und Erweiterungen unterworfen ist, ist es unmöglich, eine allgemeingültige Aussage bezüglich des längsten nicht unterbrechbaren Abschnittes im Kernel zu machen. Der Aufwand hierzu ist einfach zu groß. Montavista hat für diesen Zweck zwar ein Werkzeug entwickelt, allerdings ist anzumerken, dass die Analyse hiermit nicht vollständig möglich ist [15]. Weiterhin ist es nicht möglich, in einer Testumgebung sämtliche denkbaren Fälle durchzuspielen, dies macht eine Analyse nur zu einer statistischen und nicht zu einer umfassenden und schlüssigen Aussage.

8.2.4.3 ...mit einem verbessertem Scheduler

Alle Varianten, die den gerade beschriebenen Ansatz verfolgen, verbinden ihn zusätzlich mit einem neuen Scheduler, um zu gewährleisten, dass die Echtzeit-Tasks auch die höchste Priorität erhalten und auch bestimmt zur Ausführung gelangen.

Ein präemptiver Kernel in Kombination mit einem neuen Scheduler führt aber aus den angeführten Gründen zu einem System mit weicher Echtzeitfähigkeit.

8.2.4.4 Echtzeiteigenschaften der Linuxvarianten

In Tabelle 8.3 findet sich abschließend ein Überblick über die Eigenschaften der echtzeitfähigen Linuxvarianten. Man sieht, dass es für manche Eigenschaften verschiedene Lösungen gibt. Montavista-, RED- und TimeSys-Linux bieten z.B. alle einen Ersatz zum normalen Linux-Scheduler für User-Space-Task an.

Es gibt also kein System, welches alle Eigenschaften besitzt, bezüglich der Echtzeitfähigkeit lässt sich klar sagen, dass, wenn harte Echtzeitanforderungen gestellt werden, die Wahl nur auf RT-Linux oder RTAI (hier ist auch REDICE eingeschlossen) fallen kann. Die in Abbildung 2.1 gezeigten Reaktionszeiten lassen sich problemlos erreichen. RT-Linux und RTAI lassen sich sogar in Bereichen in der Antriebssteuerung einsetzen.

Der Ansatz, den Kernel präemptiv zu machen bietet zwar deutliche Vorteile ge-

	MontaVista-Linux	TimeSys-Linux	REDICE	RT-Linux	RTAI	RTAI + LXRT	RED-Linux	KURT
Seperater RT Kernel	○	○	(3)	●	●	●	○	○
User-Space Programmierung	●	●	●	(2)	○	(1)	●	●
Verbesserte Kernel Preempt.	○	○	○	○	○	○	●	○
Vollständige Kernel Preempt.	●	●	○	○	○	○	○	○
Präzises Scheduling in Linux	●	●	●	○	○	○	○	○
Interrupt Response $< 1mS$	○	○	●	●	●	●	○	○
Interrupt Response $< 5mS$ in Linux	●	○	○	○	○	○	●	○
○ nicht erfüllt ● erfüllt (1) eingeschränkt (2) sehr eingeschränkt (3) mit RTAI								

Tabelle 8.3: Vergleich der Varianten bezüglich ihrer Performance

genüber dem standard Linux Kernel in Bezug auf weiche Echtzeit. Basierend auf diesem Modell aber ein System für harte Echtzeit zu entwickeln würde aber zu stark zu Lasten anderer Vorteile, die Linux bietet - wie die Entwicklungeschwindigkeit und damit die Verfügbarkeit von Treibern - gehen.

8.3 Schlussfolgerungen

In diesem Abschnitt werden die sich aus der Analyse ergebenen Konsequenzen diskutiert. Hierzu wird jede Variante anhand des bereits formulierten Satzes an Kriterien untersucht und so ein charakteristisches Profil für jede Variante erstellt. Dieses Profil wird dann dem Anforderungsprofil des Prozessbeispiels gegenübergestellt, um die Eignung der jeweiligen Variante für die Lösung der Aufgabe darzustellen.

8.3.1 Unmodifizierter Linux-Kernel

Der Linux-Kernel bzw. ein GNU-Linux System ist von Hause aus ein System, welches voll dem *Open Source* Gedanken folgt. Einzelne Distributoren binden Software in ihre Distributionen ein, die nicht zu 100 % diesem Gedanken folgt (wie z.B. SuSE mit YaST2), diese Feinheiten werden hier aber außer Acht gelassen, da es genügend Distributionen gibt, die diese Problematik nicht mit sich bringen. Die Entwicklung und der Support solcher Systeme folgt dem *Community*-Gedanken und bringt alle damit verbundenen Vorteile mit sich. In diesen zwei Kriterien erreicht solch ein System einen Wert von *6* für die Erfüllung der Kriterien.

Für Linux-Systeme gibt es mittlerweile nicht nur von den Distributoren wie SuSE oder Red Hat *professionellen Support*, zusätzlich leisten hier auch weitere Firmen Support. Dieser Support steht nicht hinter dem vergleichbarer Konkurenzprodukte zurück, so dass auch hier der Wert *6* erreicht wird.

Im Bereich der *inneren Sicherheit* ergibt sich das Problem, dass man in der Regel Kernelmodule implementieren muss, um das gewünschte Verhalten zu erreichen (z.B. direkter Zugriff auf bestimmte Hardware-Komponenten). Dies führt zu einer etwas eingeschränkten inneren Sicherheit des Systems, hieraus resultiert ein Wert von *4* für dieses Kriterium.

Die *äußere Sicherheit* eines solchen Systems in der Automatisierungstechnik kann vollständig gewährleistet werden. Zum einen stehen alle Mittel zur Verfügung das System abzusichern, zum anderen können auftretende Sicherheitsmängel in der Software durch kurzfristig zur Verfügung stehende Patches oder gegebenenfalls auch durch Eigenentwicklung beseitigt werden. Es ergibt sich für die Erfüllung dieses Kriteriums der Wert *6*.

Linux Systeme lassen sich zwar auch für Geräte mit wenig zur Verfügung stehenden Resourcen anpassen, dies wird aber nicht grundsätzlich unterstützt, so dass hier bei der Entwicklung zusätzlicher Aufwand dafür entstehen kann, das System auf die Zielplattform anzupassen. Das Kriterium *Skalierbarkeit* wird nur mit einem Wert von *4* erfüllt. Die anderen Varianten bieten hier eine bessere Unterstützung.

Bezüglich der *Hardwarekonnektivität* wird ein Wert von *5* erreicht. Für fast

alle gängige Hardware bestehen entsprechende Treiber. Im Vergleich zu anderen System ist die Unterstützung aber weniger vollständig.

Bei den *Entwicklungswerkzeugen* erreicht ein System mit einem unmodifzierten Linux Kernel eine Wert von *5*. Einziger Schwachpunkt ist hier das Fehlen an Werkzeugen, die den Entwickler bei der Arbeit an Aufgaben mit Echtzeitanforderungen unterstützen. Dies verwundert nicht, denn dieser Anwendungsbereich gehört nicht zu den primären Einsatzgebieten von Linux.

Die *Softwareentwicklung* unterliegt keinen besonderen Einschränkungen und erreicht einen Wert von *6* für dieses Kriterium.

Echtzeitfähigkeit lässt sich mit einem unmodifizierten Linux Kernel zwar unter bestimmten Randbedingungen erreichen (vergl. Absch. 7.3), allerdings ist dieser sicher nicht der standard Weg. Ein standard Linux-System bietet keine Echtzeitfähigkeit, die sich für Aufgaben in der Automatisierungstechnik nutzen lassen kann - es gibt keine garantierten Antwortzeiten, so dass für dieses Kriterium ein Wert von *1* erreicht wird.

In Abbildung 8.1 ist der Charakter von Linux in einem Netzdiagramm dargestellt.

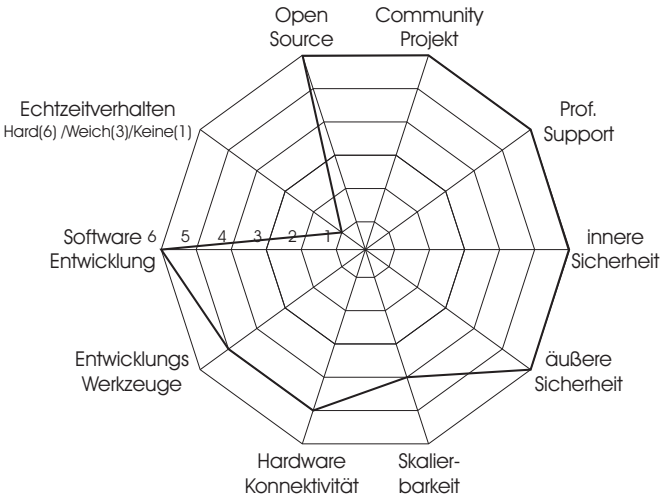


Abbildung 8.1: Charakteristika Linux

8.3.2 RT Linux

RT Linux existiert in zwei Varianten, die sich durch das verwendete Lizenzmodell unterscheiden, zum einen existiert eine Version, die unter der GPL steht, zum anderen eine Version, für die Lizenzkosten zu zahlen sind. In wie weit die GPL-lizensierte Version von FSMLabs ernst genommen werden kann, sei dahingestellt:

Es ergibt sich hier eher der Eindruck, der Anbieter will damit nur das kommerzielle Produkt bewerben, FSMLabs bucht die Kosten, die durch die GPL-Version verursacht werden, unter Marketing [158]:

> The GPL/RTLinux work that we do loses money – actually, we budget it under "marketing".

Für das patentierte RT-Linux gilt: Man sollte es nur verwenden, wenn man bereit ist, die durch die Patentlizenz formulierten Einschränkungen zu akzeptieren und sich so in die Abhängigkeit von FSMLabs begeben.

Die Philosophie, die hinter der Entwicklung von RT Linux steht, folgt also nur bedingt den Ideen der Open Source Community, hieraus resultieren die Werte von *4* für die Kategorie *Open Source* und *5* für *Community Projekt*. FSMLabs hat nur begrenzte Interesse, das Produkt RT Linux als Open Source und Gemeinschaftsprojekt zu pflegen.

Da man bei der Entwicklung von echtzeitfähigen Anwendungen unter RT Linux zwingend darauf angewiesen ist, Kernelmodule zu entwickeln - normale User Space Anwendungen profitieren schließlich nicht von RT Linux - wird bei der *inneren Sicherheit* nur ein Wert von *4* erreicht. Grund hierfür sind die in Abschnitt 8.2.2.1 aufgeführten Punkte.

Bei der *äußeren Sicherheit* macht sich die Notwendigkeit bemerkbar, die Patches für die Echtzeitfähigkeit erst an die neue Kernelversion anpassen zu müssen. Daher ist für die aktuellste Kernel-Version in der Regel kein Patch vorhanden. Es ist aber zu bedenken, dass die meisten Sicherheitslücken nicht durch Fehler im Kernel sondern durch Lücken in User Space Anwendungen verursacht werden. Es ergibt sich somit ein Wert von *5* für die äußere Sicherheit.

Da RT Linux nicht primär für die Anwendung im embedded Bereich vorgesehen ist, muss der Entwickler hier selber tätig werden und sich "sein" System mit den entsprechenden Eigenschaften zusammenstellen. Allerdings existieren einige Beispiel-Implementationen, die diese Arbeit erleichtern. Beim Kriterium *Skalierbarkeit* ergibt sich so ein Wert von *5*.

Bei der *Konnektivität* ergibt sich ein schlechterer Wert als beim unmodifizierten Linux Kernel. Dies läßt sich folgendermaßen begründen: Soll eine Hardwarekomponente innerhalb einer echtzeitfähigen Anwendung benutzt werden, so muss zuerst der Treiber - d.h. das Kernel Modul - angepasst werden. Es existieren erst wenige, speziell für diesen Einsatz modifizierte Treiber (z.B. für einige Netzwerkkarten und die serielle Schnittstelle [73]). Da aber der Quelltext zu den Hardwaretreibern in der Regel vorliegt, stellt dies aber kein unlösbares Problem dar. Hieraus resultiert der Wert *4*.

Bei den Entwicklungswerzeugen greift RT Linux auf die vorgestellten GNU Tools zurück. Diese sind aber nicht für die Entwicklung für echtzeitfähige Anwen-

dungen gedacht und bieten daher an dieser Stelle auch nur wenig Unterstützung - Bewertung: *4*.

Durch das eigene Programmiermodell für Echtzeittask (vergl. Absch. 8.2.3.4.1) ergibt sich hier ein erhöhter Aufwand im Vergleich zu Varianten, die den Ansatz verfolgen, den Kernel unterbrechbar zu machen. Dies führt zu einer schlechteren Bewertung *3*. Bei der Softwareentwicklung ist man gezwungen, den Nachteil zweier unterschiedlicher Programmiermodelle (für den Kernel- bzw. User-Space) in Kauf zu nehmen.

Der gewählte Ansatz, den Linux Kernel durch einen Echtzeitkernel zu ersetzen, ermöglicht es, Systeme mit harten Echtzeiteigenschaften zu realisieren - dies entspricht einem Wert von *6*.

In Abbildung 8.2 ist der Charakter von RT Linux in einem Netzdiagramm dargestellt.

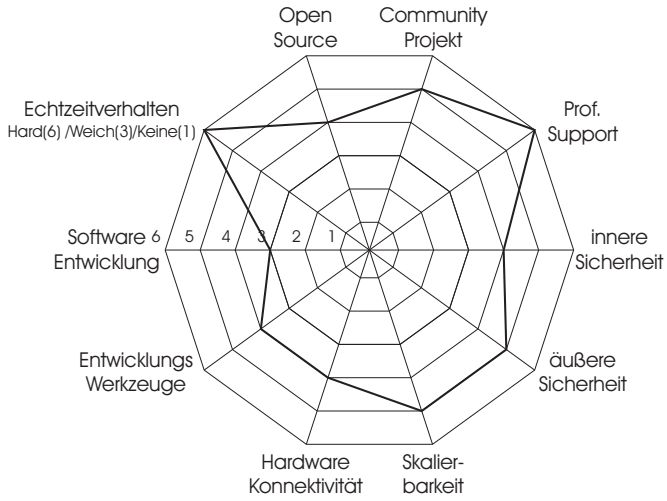


Abbildung 8.2: Charakteristika RT Linux

8.3.3 RTAI und RTAI mit LXRT

Obwohl technisch sehr eng mit RT Linux verwandt, zeigen sich bei den Kriterien *Open Source* und *Community Projekt* deutliche Unterschiede. RTAI Linux unterliegt vollständig der GPL und wird als offenes Projekt geführt, ganz im Stil der Linux Kernelentwicklung. Hieraus resultiert die vollständige Erfüllung beider Kriterien.

Bei der inneren Sicherheit gilt auf Grund der gleichen Idee, der Ersetzung des Kernels, das bereits für RT Linux Gesagte und somit auch die gleiche Beurteilung für dieses Kriterium.

Ein anderes Bild ergibt sich, wenn ein System beim Einsatz von LXRT beurteilt. Hierdurch wird es möglich, harte Echtzeit auch im User Space zu erreichen und somit auch eine bessere innere Sicherheit als bei RT Linux. Konsequenz ist eine bessere Bewertung von RTAI Linux mit LXRT im Bereich *innere Sicherheit 5*. Allerdings kann nicht ganz der Wert von User Space Programmen unter dem Standard Linux Kernel erreicht werden, da die im User Space laufenden Echtzeit-Tasks mit den entsprechenden Kernelmodulen kommunizieren und so unter Umständen durch Systemaufrufen die Echtzeitfähigkeit verloren gehen kann oder auch die Stabilität des Systems beeinträchtigt werden kann (vergl. Absch. 7.4.2.2.3).

Bei den weiteren Kriterien macht es sich deutlich bemerkbar, dass beide Systeme die gleiche Wurzel besitzen. Sie haben bezüglich der folgenden Kriterien die gleichen Eigenschaften:

Da RTAI den gleichen Ansatz verfolgt, um Echtzeitfähigkeit zu erreichen, ergibt sich für die *äußere Sicherheit* das gleiche Bild wie bei RT Linux (Wert *5*).

Auch RTAI ist primär nicht für den Einsatz als embedded Betriebssystem gedacht. Das ursprüngliche Einsatzgebiet sind Laborrechner an einem Universitätslehrstuhl [90]. So ergibt sich für die Skalierbarkeit ein Wert von *5*. Bei der Implementierung von embedded System wird der Entwickler nicht durch spezielle Werkzeuge unterstützt.

Die Aussagen bezüglich der *Konnektivität* von RT Linux lassen sich auf RTAI übertragen. Auch beim Einsatz von LXRT müssen die Treiber für die verwendete Hardware echtzeitfähig gemacht werden, da die Echtzeit des Systems immer noch durch das Ersetzen des Linux Kernels erreicht wird. RTAI erreicht bei der Bewertung dieses Kriteriums den Wert *4*.

Beim Kriterium *Entwicklungswerkzeuge* erreicht RTAI den gleichen Wert *4* wie RT Linux, da ebenfalls nur der GNU Werzeugsatz unterstützt wird und keine speziellen Werkzeuge geboten werden, die die Entwicklung echtzeitfähiger Anwendungen unterstützen.

Betrachtet man RTAI unter dem Gesichtspunkt der *Softwareentwicklung*, so ergibt sich die gleiche Beurteilung wie bei RT Linux. Um echtzeitfähige Anwendungen zu erstellen, müssen Kernelmodule erstellt werden - mit den bereits erwähnten Besonderheiten (vergl Absch. 8.2.3.4.1).

Beim Einsatz der Kombination RTAI mit LXRT ergibt sich eine bessere Bewertung für das Kriteriums *Softwareentwicklung*. Da die Entwicklung von echtzeitfähigen Anwendungen mit Einschränkungen im User Space möglich ist (vergl. Absch. 8.2.3.4.1), ist die Beurteilung für dieses Kriterium besser als beim alleinigen Einsatz von RTAI - es ergibt sich ein Wert von *5*.

Beide Varianten (RTAI und RTAI in Kombination mit LXRT) erreichen hartes Echtzeitverhalten und somit einen Wert von *6* für das Kriterium *Echtzeitverhal-*

ten.

In Abbildung 8.3 und 8.4 sind die Eigenschaften von RTAI und RTAI mit LXRT als Netzdiagramm dargestellt.

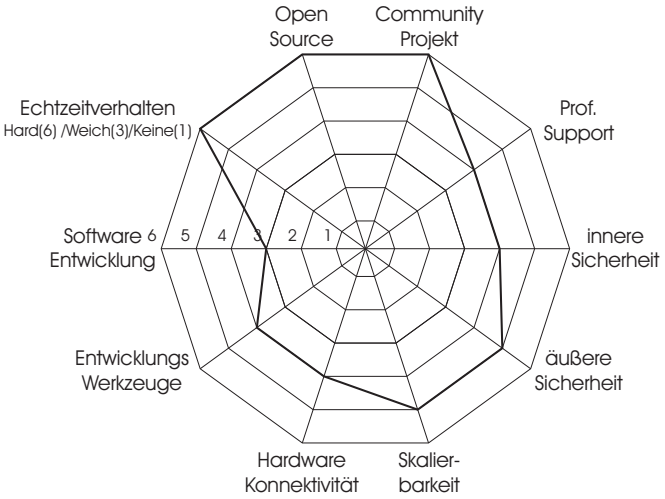


Abbildung 8.3: Charakteristika RTAI Linux

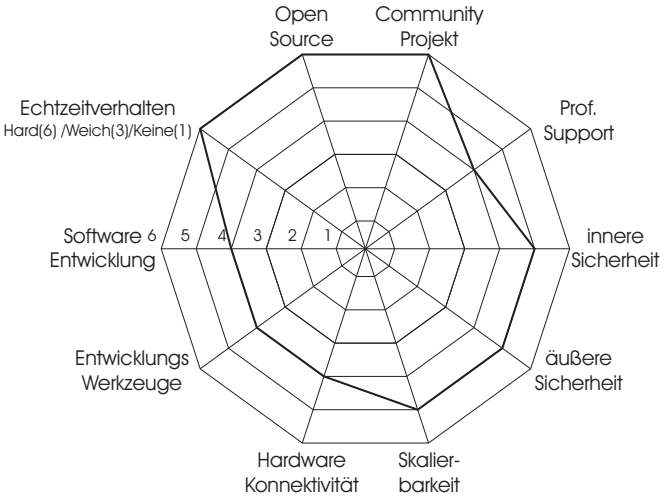


Abbildung 8.4: Charakteristika RTAI Linux mit LXRT

8.3.4 KURT

Genau wie RTAI Linux handelt es sich bei KURT um ein Projekt, das gänzlich unter der GPL steht. Die Entwicklung erfolgt im Sinne eines Community Projektes, zu dem jeder seinen Beitrag leisten kann. In den beiden Kategorien *Open Source* und *Community Projekt* erreicht KURT genau wie RTAI Linux den Wert *6*.

Beim Kriterium *professioneller Support* erreicht KURT nur einen Wert von *2*. Das Angebot an professionellem Support für KURT ist geringer als bei den beiden vorigen Varianten. Dies liegt daran, dass KURT bisher ausschließlich im universitären Umfeld eingesetzt wird (vergl. [125]). Begründen lässt sich dies mit dem sehr speziellen Einsatzgebiet, für den KURT vorgesehen ist (vergl. [120]).

Da KURT den Ansatz verfolgt, die Echtzeitfähigkeit durch Veränderungen am Kernel zu erreichen, kann die Programmierung von Automatisierungssoftware - wie beim Standard Linux Kernel - im User Space erfolgen. Daher ist die *innere Sicherheit* des Systems sehr hoch, dies spiegelt sich im Wert von *6* für dieses Kriterium wieder.

Bei der *äußeren Sicherheit* erreicht KURT eine schlechtere Bewertung als RT Linux und RTAI. Es werden nur ältere Kernel-Versionen unterstützt. Die Entwicklung von Kurt hängt hinter der von RTAI und RT Linux zurück. KURT erreicht hier nur den Wert *4*.

KURT stellt keine Anforderungen an die eingesetzten Hardwaretreiber, Treiber die für den standard Kernel existieren, können auch unter KURT eingesetzt werden. Es ergibt sich somit der gleiche Wert *5* wie für den standard Linux Kernel.

Beim Kriterium *Entwicklungswerkzeuge* erreicht KURT den gleichen Wert (*4*) wie die bisher untersuchten Varianten, da auch KURT nur den allgemeinen GNU Werkzeugsatz unterstützt.

Beim Kriterium *Softwareentwicklung* profitiert KURT davon, nicht die Entwicklung von Kernel Modulen für die Erreichung des Echtzeitverhaltens vorauszusetzen. Die Entwicklung beschränkt sich auf den User Space und erfordert nur die Anwendung der durch KURT bereitgestellten Funktionalität. So wird für dieses Kriterium der Wert *6* erreicht, der Unterschied zur Entwicklung unter dem standard Linux Kernel ist nur gering.

Bezüglich des Kriteriums *Echtzeitverhalten* erreicht KURT den Wert *3*, da KURT nur weiches Echtzeitverhalten bietet (vergl. Absch. 7.4.3).

8.3.5 Montavista Linux

Obwohl Montavista Linux im Gegensatz zu RTAI und KURT ein vollständige kommerzielle Distribution ist, erreicht es bei den Kriterien *Open Source* und

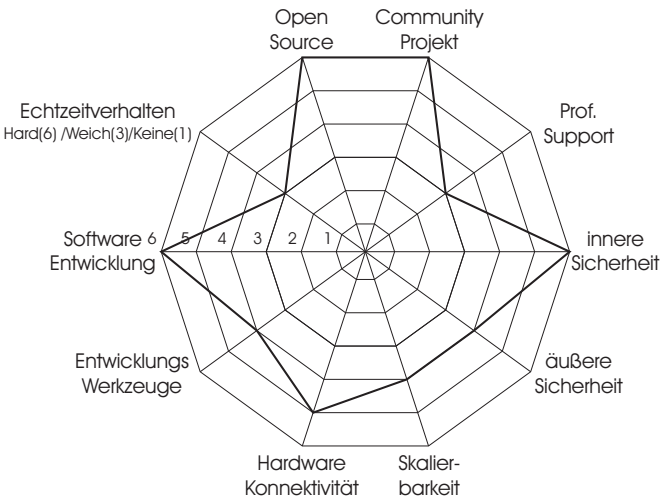


Abbildung 8.5: Charakteristika KURT Linux

Community Projekt den Wert *4*. Montavista unterstützt die Weiterentwicklung von Linux durch die Bereitstellung des in Montavista verwendeten Patches für die Unterbrechbarkeit und des neuen Schedulers. Eine gemeinschaftliche Entwicklung wie bei RTAI, KURT oder zum Teil auch bei RT Linux findet aber nicht statt.

Professioneller Support ist direkt durch den Distributor gegeben. Somit ergibt sich ein in diesem Bereich ein Wert von *6*.

Da Montavista Linux - genau wie das bereits untersuchte KURT - nur durch Modifikationen am Kernel die Echtzeiteigenschaften beeinflusst, erfolgt die Entwicklung von echtzeitfähigen Anwendungen im User Space und profitiert von allen bestehenden Sicherheitsmechanismen für Anwendungen. Für die *innere Sicherheit* ergibt sich so ein Wert von *6*.

Bei der *äußeren Sicherheit* erreicht Montavista Linux einen Wert von *5*, da neue Kernel-Versionen immer erst mit einiger Verzögerung die notwendigen Patches erhalten. Hier nochmals der Hinweis, dass viele Sicherheitslücken nicht durch Fehler im Kernel hervorgerufen werden, sondern durch Fehler in Anwendungen, die im User Space ablaufen.

Beim Kriterium *Skalierbarkeit* erreicht Montavista Linux den Höchstwert von *6*. Zieleinsatzgebiet von Montavista Linux war von Anfang an der Embedded Bereich. So stehen entsprechende Werkzeuge für die Entwicklung solcher Systeme zur Verfügung.

Für die *Hardwarekonnektivität* ergibt sich der gleiche Wert *5*, wie für den Standard Linux Kernel. Dadurch, dass der Kernel vollständig unterbrechbar wird, muss auch bei der Treiberentwicklung kein Mehraufwand getrieben werden. Generell lässt sich sagen, dass ein Treiber, der in einer Multiprozessorumgebung

funktioniert, auch problemlos unter Montavista Linux eingesetzt werden kann.

Montavista stellt zur Entwicklung von Anwendungen - neben den standard GNU Werkzeugen - einige spezielle Werkzeuge zur Verfügung. Diese ermöglichen es zum Beispiel das Timingverhalten der Anwendungen zu untersuchen und zu debuggen. Hierdurch wird ein Wert von *6* für den Bereich *Entwicklungswerkzeuge* erreicht.

Wie bereits erwähnt, ist es nicht notwendig, Kernel Module zu entwickeln, um Echtzeitverhalten unter Montavista Linux zu erreichen. Die Bewertung im Bereich *Softwareentwicklung* ist gleich mit der des Standard Linux Kernels *6*.

Wie in Abschnitt 8.2.4.2 beschrieben, wird durch den von Montavista gewählten Weg, das Echtzeitverhalten des Linux Kernels zu beeinflussen, nur weiches Echtzeitverhalten erreicht. Somit ergibt sich ein Wert von *3* in der Kategorie *Echtzeitverhalten*.

Abbildung 8.6 zeigt die Eigenschaften von Montavista Linux als Netzdiagramm.

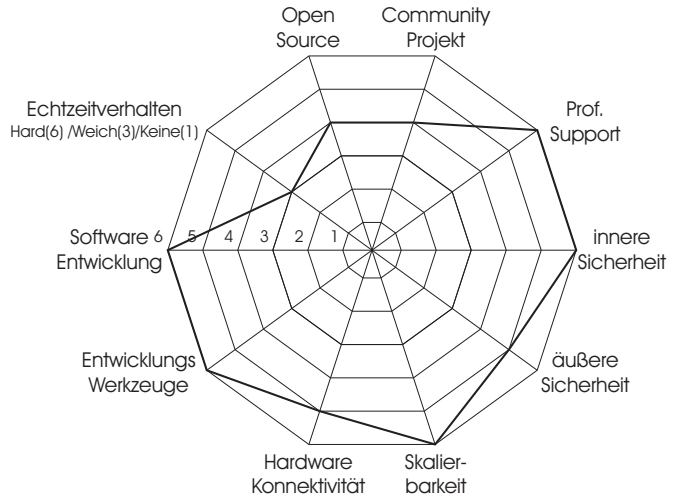


Abbildung 8.6: Charakteristika Montavista Linux

8.3.6 Timesys Linux

Timesys Linux ist eine kommerzielle Linux Distribution. Dies macht die in den Bereichen *Open Source* und *Community Projekt* schlechtere Beurteilung als RT Linux und RTAI deutlich. Auch gegenüber Montavista Linux fällt die Beurteilung schlechter aus: Timesys Linux ist zwar in einer GPL lizensierten Version erhältlich, diese entspricht aber nicht dem letzten Entwicklungsstand. Weiterhin umfasst sie nicht die volle Funktionalität. Um diese zu erhalten, sind weiter proprietäre Module erhältlich (vergl. Absch. 7.4.5). Timesys erfüllt also bedingt den

Open Source Gedanken. Die Entwicklung findet komplett in Eigenregie statt. Für beide Kriterien ergibt sich so nur ein Wert von *3*.

Professioneller Support ist direkt durch den Distributor gegeben. Somit ergibt sich ein in diesem Bereich ein Wert von *6*.

Die *innere Sicherheit* von Timesys Linux erhält die gleiche Bewertung wie Montavista Linux *6*. Dies liegt darin begründet, dass auch Timesys Linux das Echtzeitverhalten im User Space beeinflusst. Das heißt, echtzeitfähige Anwendungen können im User Space ausgeführt werden und müssen nicht als Kernel Modul realisiert werden. Genau wie unter Montavista Linux können so die Schutzmechanismen des Kernels eine hohe innere Sicherheit gewährleisten.

Bei der *äußeren Sicherheit* erreicht Timesys Linux nur den Wert *4*. Dies ist darin begründet, dass Timesys Linux nicht die aktuellsten Kernel unterstützt, es liegt ebenso hinter der aktuellen Entwicklung zurück wie z.B. Montavista Linux.

Bei der *Skalierbarkeit* erreicht Timesys Linux die gleiche Bewertung *6* wie Montavista Linux. Beide sind für den gleichen Einsatzbereich vorgesehen und es stehen Werkzeuge zur Unterstützung zur Verfügung.

Timesys Linux profitiert im Bereich *Hardwarekonnektivität* vom gewählten Ansatz die Echtzeiteigenschaften des Linux Kernels zu verbessern. Alle Treiber die unter dem Standard Linux Kernel zur Verfügung stehen können auch unter Timesys Linux benutzt werden. Es ergibt sich ein Wert von *5*.

Genau wie Montavista stehen für Timesys Linux spezielle Entwicklungswerkzeuge zur Vefügung, die über den Funktionsumfang des GNU Werkzeugsatzes hinausgehen. Timesys Linux erreicht in diesem Bereich die Bewertung *6*.

Die Bewertung im Bereich *Softwareentwicklung* erreicht genau wie das im vorigen Abschnitt beurteilte Montavista Linux den Wert *6*, da die Anwendungen die Echtzeitanforderungen stellen ebenfalls nur im User Space ablaufen.

Durch den gewählten Weg zur Echtzeitfähigkeit ist zugleich auch festgelegt, dass nur weiches Echtzeitverhalten erreicht werden kann. Somit ergibt sich in diesem Bereich ein Wert von *3*.

Abbildung 8.7 stellt die Charakteristik von Timesys Linux in einem Netzdiagramm dar.

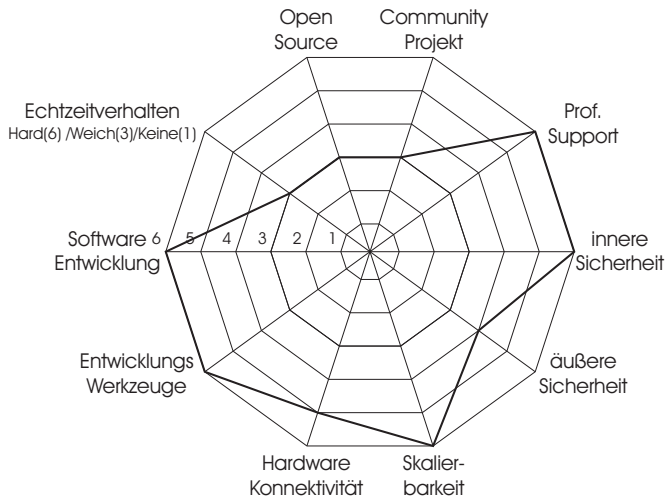


Abbildung 8.7: Charakteristika Timesys Linux

8.3.7 REDICE

Im Bereich *Open Source* und *Community Projekt* schneidet REDICE Linux genau so schlecht ab, wie Timesys Linux. Es wird nur der Wert *3* erreicht. Die Entwicklung erfolgt alleine durch den Hersteller und dieser ist auch nicht bestrebt, andere aktiv daran teilhaben zu lassen.

Professioneller Support ist direkt durch den Distributor gegeben. Somit ergibt sich ein in diesem Bereich ein Wert von *6*.

Bei der *inneren Sicherheit* muss eine zweigeteilte Betrachtung erfolgen: Will man harte Echtzeit für seine Anwendung, so man muss genau wie bei RTAI und RT Linux Kernelmodule entwickeln. Hieraus resultieren die gleichen Einschränkungen wie bei den beiden Varianten, es wird somit in diesem Fall nur ein Wert von *4* erreicht. Benötigt die zu entwickelnde Anwendung allerdings nur weiches Echtzeitverhalten, so können die Anwendungen für den User Space entwickelt werden. In diesem Fall ergibt sich ein Wert von *6* für dieses Kriterium.

Bei der *äußeren Sicherheit* erreicht REDICE Linux nur den Wert *4*. Dies ist darin begründet, dass genau wie bei Timesys Linux nicht der aktuellste Kernel unterstützt wird und die zugrundeliegende Kernelversion einige Schritte hinter der aktuellsten zurück liegt.

REDICE Linux ist für den Einsatz als embedded Betriebssystem vorgesehen, deshalb stehen auch Tools zur Verfügung, die bei der Entwicklung solcher Systeme helfen. Bei der *Skalierbarkeit* erreicht REDICE Linux die gleiche Bewertung (*6*) wie Montavista und Timesys Linux.

Für die *Hardwarekonnektivität* ergibt sich die gleiche zweigeteilte Situation wie beim Kriterium *innere Sicherheit*. Bei geforderter harter Echtzeitfähigkeit müssen

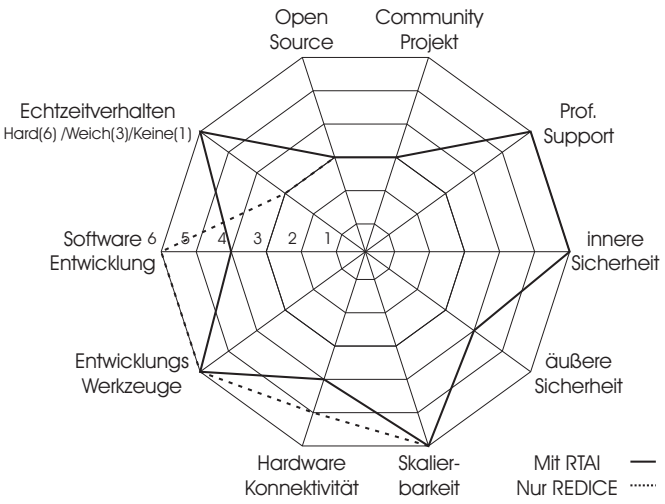


Abbildung 8.8: Charakteristika REDICE Linux

die Treiber für die Hardware entsprechend modifiziert werden (vergl. RTAI und RT-Linux). Aufgrund dieser Einschränkung erreicht REDICE in diesem Fall einen Wert von *4* in dieser Kategorie. Genügt weiches Echtzeitverhalten, so können wie bei Montavista oder Timesys Linux die vorhandenen Treiber weiter genutzt werden, es wird in diesem Fall der Wert *5* für dieses Kriterium erreicht.

Für REDICE Linux stehen speziell an diese Variante angepasste Entwicklungswerkzeuge für die Entwicklung zur Verfügung, die über den Funktionsumfang des GNU Werkzeugsatzes hinausgehen. Somit erreicht REDICE Linux in diesem Bereich die Bewertung *6*.

Soll Software entwickelt weren, welche hartes Echtzeitverhalten bietet, müssen genau wie bei RTAI Linux Kernel Module entwickelt werden. Hieraus resultieren die gleichen Nachteile bzw. Einschränkungen wie bei RT und RTAI Linux. Es wird in diesem Fall nur der Wert *3* für dieses Kriterium erreicht. Genügt weiches Echtzeitverhalten, so kann mit weniger Aufwand für den User Space entwickelt werden und es ergibt sich ein Wert von *6* für dieses Kriterium.

Im Bereich des *Echtzeitverhaltens* ergibt sich genau wie in den Bereichen *innere Sicherheit* und *Hardwarekonnektivität* eine zweigeteiltes Ergebnis: Bei der Nutzung der RTAI Komponente wird hartes Echtzeitverhalten erreicht *6*, nutzt man diese nicht, so kann nur weiches Echtzeitverhalten erreicht werden *3*.

Abbildung 8.8 stellt die Charakteristik von REDICE Linux in einem Netzdiagramm dar.

Im folgenden Abschnitt erfolgt die Formulierung der Anforderungen, die durch ein konkretes Prozessbeispiel an die Linux-Varianten gestellt werden und daran anschließend ein exemplarisches Auswahlverfahren der geeignetsten Variante.

8.4 Prozessbeispiel

Wie bereits in der Einleitung erwähnt, wurde die vorliegende Dissertation durch Probleme angestoßen, die bei der Weiterentwicklung bzw. Ergänzung der Software für die Prüfung mechanischer Druckregler auf den am Fachgebiet Meßtechnik entwickelten Prüfständen [94] auftraten. Eine detaillierte Beschreibung des Systems findet sich im Anhang A (S. 131).

Bei den Prüfständen handelt es sich um einen Anwendungsfall, der einen großen Teil der in der Automatisierungstechnik wichtigen Aspekte abdeckt:

1. Steuerung der Prozesshardware.
2. Anbindung an ein übergeordnetes PPS-System.
3. Visualisierung.
4. Fernwartbarkeit.

Formuliert man - analog der in Abschnitt 6 aufgestellten Kriterien - für dieses Beispiel die Anforderungen, die an ein auszuwählendes Betriebssystem gestellt werden, so ergibt sich folgendes Bild:

8.4.1 Kosten

Die am Fachgebiet gesammelten Erfahrungen haben dazu geführt, sich gegen eine proprietäre Lösung zu entscheiden. Beim ausgewählten Betriebssystem soll es sich um ein Projekt handeln, welches dem Open Source Gedanken folgt und welches keine proprietären Komponenten beinhaltet. Für beide Punkte ist ein Wert von *6* in diesen Kategorien anzustreben.

Auf kommerziellen Support kann ganz verzichtet werden, da die Entwicklung komplett in Eigenregie erfolgen soll - hier wird ein Wert von *0* angesetzt. Der Support durch eine Gemeinschaft aus Entwicklern und Anwendern über Mailinglisten oder Foren wird deutlich bevorzugt.

8.4.2 Sicherheit

Die *äußere Sicherheit* des System spielt keine überragende Rolle, da das System in einem durch eine Firewall geschützten Unternehmensnetz eingesetzt wird. Natürlich muss das System ein Mindestmaß an Sicherheit bieten, da Angriffe von innen nie grundsätzlich ausgeschlossen werden können. Als Anforderung ergibt sich in diesem Punkt ein Wert von *4*.

Die *innere Sicherheit* des System stellt etwas höhere Anforderungen, da die Systeme weitgehend autonom und ohne Wartung betrieben werden sollen. Ein vom Personal nicht feststellbarer Ausfallgrund der Software soll vermieden werden. Bei diesem Kriterium wird deshalb ein Wert von *5* gefordert.

8.4.3 Flexibilität

Bei Kriterium *Skalierbarkeit* aus dem Bereich Flexibilität ergeben sich die folgenden Anforderungen: Die Zielplattform soll weiterhin aus Intel kompatibler Hardware bestehen. Genauso soll das System weiterhin mit möglichst wenig Resourcen (RAM, Rechenleistung und Festplattenkapazität) betrieben werden können. Diese Anforderungen sind aber mehr unter dem Gesichtspunkt "einfacher PC" zu sehen, als embedded System mit sehr spezieller Hardware. Als Wert für dieses Kriterium wird ein Wert von *4* vorgegeben.

Beim Kriterium *Hardwarekonnektivität* werden die Anforderungen durch die beiden Kommunikationskanäle Serielle Schnittstelle (zur Kopplung mit der S7) und Netzwerkkarte (Anbindung an das PPS-System) beschrieben. Da es sich bei beiden Hardwarekomponenten um standard Komponenten handelt und nur die serielle Schnittstelle Echtzeitverhalten bieten muss, wird ein Wert von *4* für dieses Kriterium angenommen.

Die *Entwicklung der Software* sollte ohne besonderen Aufwand und Einschränkungen durch spezielle Programmiermodelle möglich sein, da eine lange Einarbeitungszeit vermieden werden soll - die Entwicklung wird teilweise durch Personal durchgeführt, welches nur eine beschränkte Zeitspanne am Fachgebiet beschäftigt ist. Für diesen Bereich wird ein Wert von *5* gefordert.

Der gleiche Wert wird auch für den Bereich *Entwicklungswerkzeuge* angesetzt. Die Echtzeitanforderungen des Systems sind zwar nicht sehr komplex, so dass speziell für diesen Bereich entwickelte Werkzeuge nicht nötig sind.

8.4.4 Performance

Da die Software auf dem PC die Aufgabe der SPS übernimmt und eine SPS hartes Echtzeitverhalten bietet, muss das eingesetzte Betriebssystem ebenfalls hartes Echtzeitverhalten unterstützen. Der Wert für dieses Kriterium ist mit *6* angesetzt.

8.5 Auswahl der Variante

Dargestellt in einem Netzdiagramm ergibt sich aus den formulierten Anforderungen das in Bild 8.9 dargestellte Anforderungsprofil.

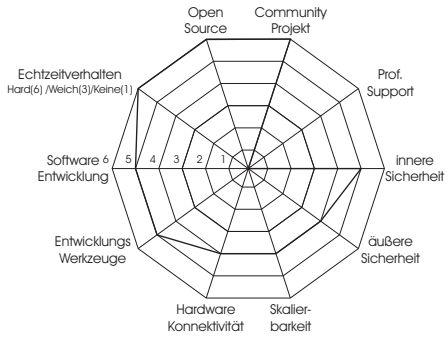


Abbildung 8.9: Anforderungen an das Betriebssystem

Dem gegenüber stehen die bereits in Abschnitt 8.3 herausgearbeiteten und in Abbildung 8.10 zusammengestellten Profile der Linux-Varianten.

Für jedes einzelne Kriterium kann die Erfüllung festgestellt werden.

Bei den drei dem Oberbegriff *Kosten* zugeordneten Kategorien *Open Source* und *Community Projekt* ergibt sich folgende Situation:

Linux, RTAI Linux, RTAI Linux mit LXRT und KURT Linux erfüllen die gestellten Anforderungen zu 100 Prozent. Linux ist sicher das Paradebeispiel für Open Source Software. RT Linux erfüllt die Anforderungen bezüglich dieser beiden Aspekte bereits weniger gut. Zwar gibt es RT Linux in einer GPL lizensierten Version, allerdings macht sich das Bestehen des Softwarepatents negativ bemerkbar: Man sollte RT Linux nur verwenden, wenn man bereit ist, die durch die Patentlizenz formulierten Einschränkungen zu akzeptieren.

Die drei anderen Varianten schneiden schlechter ab. Montavista unterstützt zwar die Weiterentwicklung von Linux durch die Veröffentlichung der zwei erwähnten Patches, dies hat aber wenig mit dem Produkt Montavista Linux zu tun. Dessen Entwicklung erfolgt genau wie bei den beiden Varianten Timesys und REDICE Linux in Eigen-Regie. Die den Varianten zugehörigen Werkzeuge werden nicht unter der GPL veröffentlicht, somit ist eine Abhängigkeit von einem Hersteller gegeben, die so nicht gewollt ist.

Das Kriterium *Professioneller Support* stellt keine Herausforderung dar, alle Varianten werden durch Dienstleister oder den Hersteller direkt unterstützt. Die Wahl der Variante wird hierdurch nicht eingeschränkt.

Im Bereich *Sicherheit* erfüllt das unmodifizierte Linux die Anforderungen am besten. Sowohl bezüglich der *inneren* als auch der *äußeren Sicherheit* bietet es den besten Schutz. Zum einen die Entwicklung und Ausführung der Anwendungen im User Space als auch die Aktualität des Kernels bieten eine hervorragenden Schutz des Systems. Bekannt werdende Sicherheitsprobleme, die eine Gefährdung des Systems von außen bedeuten, können entweder durch Eingriff in den Kernel oder

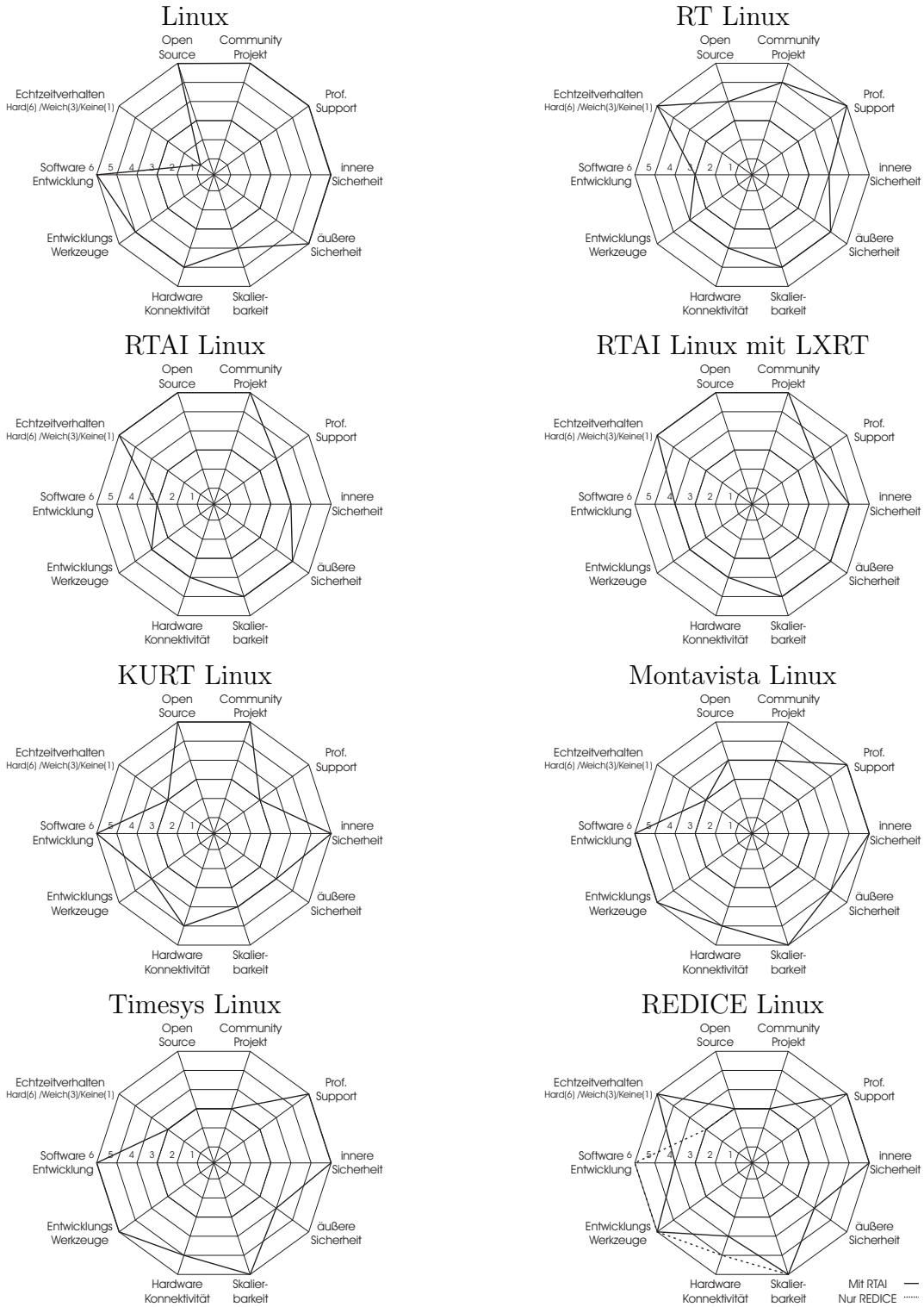


Abbildung 8.10: Charakteristika der einzelnen Varianten

die betroffene Anwendung selbst behoben werden. In der Regel stehen innerhalb kürzester Zeit (Stunden) Patches bereit, die den Fehler beseitigen und so den eigenen Eingriff unnötig machen.

Da alle anderen Varianten Eingriffe in den Kernel beinhalten, benötigen sie eine eigene Behandlung und profitieren so nicht direkt von der Weiterentwicklung des unmodifizierten Kernels. Die Patches, die die Echtzeitfähigkeit bewerkstelligen, müssen erst an neue Kernelversionen angepasst werden und sind somit nicht direkt verfügbar. Besonders schlecht schneiden hier Timesys und REDICE Linux ab, die eine sehr alte Kernelversion benutzen.

Alle Varianten, die eine Ausführung der Echtzeitanwendung im User Space ermöglichen, sind für das untersuchte Beispiel eine gute Wahl bezüglich der *inneren Sicherheit*. Montavista, Timesys und REDICE Linux bieten Echtzeitverhalten für Programme im User Space und profitieren von den Sicherheitsmechanismen, die dort vorhanden sind.

Mit kleinen Abstrichen bietet auch die Kombination aus RTAI und LXRT gute *innere Sicherheit*, da hier die Echtzeitfähigkeit auch im User Space zur Verfügung gestellt wird. Somit ist sichergestellt, dass die echtzeitfähigen Anwendungen nicht den Kernel zum Absturz bringen.

RT Linux und RTAI Linux, die ganz auf die Erstellung von Kernelmodulen angewiesen sind, um Echtzeitfähigkeit für Anwendungen zu erreichen, erfüllen die Forderung nach innerer Sicherheit am schlechtesten.

Betrachtet man die dem Bereich Flexibilität zugeordneten Kriterien ergibt sich folgendes Bild:

Das gewählte Beispiel stellt keine besonderen Anforderungen an die *Skalierbarkeit*. Die Anwendung soll auf einem handelsüblichen PC laufen und der Resourcenbedarf soll klein sein. Diese Forderung lässt sich von allen Varianten erfüllen.

Die Varianten Montavista, Timesys und REDICE Linux bieten hier durch die vorhandenen Werkzeuge Möglichkeiten, die über das geforderte hinausgehen. Diese Werkzeuge bieten die Möglichkeit, sehr kompakte Versionen des Betriebssystems zusammen mit der erstellten Anwendung zu erstellen, die dann mit geringem Speicherplatz auf dem Bootmedium auskommen.

Die Varianten RT, RTAI und Kurt Linux sind auch ohne großen Aufwand an die begrenzten Ressourcen anpassbar.

Um eine Entwicklung oder einen Eingriff in die Programme vor Ort - d.h. am Ort, an dem die Prüfstände aufgestellt sind - zu ermöglichen, ist es nicht notwendig, die komplette Entwicklungsumgebung samt Quellen auf dem Prüfstand bereit zu halten, viel mehr ist es problemlos möglich, diese über das Netzwerk einzubinden.

Beim Kriterium *Echtzeitverhalten* ist die Auswahl der Variante am einfachsten: Die Anwendung erfordert hartes Echtzeitverhalten, somit kommen nur vier Va-

rianten in Frage. RT, RTAI, RTAI mit LXRT und REDICE sind die einzigen Varianten, die dieses Kriterium erfüllen. KURT, Montavista und Timesys Linux bieten nur weiches Echtzeitverhalten und kommen somit nicht in Frage. Durch dieses letzte Kriterium wird die Auswahl der Variante am deutlichsten eingeschränkt.

Betrachtet man den Bereich Kosten - mit den zugeordneten Kriterien *Open Source*, *Community Projekt* und *Professioneller Support* - ergeben sich weitere Ausschlüsse bei der Auswahl: REDICE Linux mit seinen proprietären Komponenten erfüllt nicht die Forderung nach Open Source. Auch kann es nicht als *Community Projekt* bezeichnet werden. Gleiches gilt für die nur weiche Echtzeit bietenden Varianten Timesys und Montavista Linux.

KURT erfüllt die Kriterien *Open Source* und *Community Projekt*, bietet aber nur weiches Echtzeitverhalten.

Übrig bleiben nur die Varianten RT Linux, RTAI und RTAI mit LXRT. RT Linux scheidet aber auf Grund des bestehenden Software Patents als genutzte Variante aus. Die Kriterien *Open Source* und *Community Projekt* werden von RTAI besser erfüllt. Zudem steht mit der Kombination RTAI mit LXRT eine Variante zur Verfügung, die in Bezug auf den Aspekt der *Software Entwicklung* die Anforderungen besser erfüllt.

Kapitel 9

Zusammenfassung und Ausblick

9.1 Zusammenfassung

In der vorliegenden Arbeit wurde ein Analyse des Entwicklungsstandes von echtzeitfähigen Linuxvarianten durchgeführt und anhand dieser die generelle Einsatzmöglichkeit der Varianten in Automatisierungstechnik untersucht.

Zur Einführung in das Themengebiet wurde ein verallgemeinertes Steuerungssystem in der Automatisierungstechnik dargestellt (Abb. 2.6). Basierend auf diesem Modell werden in Kapitel 6 die wichtigen Kriterien bei der Auswahl eines Betriebsystems herausgearbeitet. Die vier sich ergebenden Kategorien sind: Kosten, Sicherheit, Flexibilität und Performance. Diesen zugeordnet sind die Kriterien: Open Source, Commmunity Projekt, professioneller Support, innere und äußere Sicherheit, Skalierbarkeit, Hardwarekonnektivität, Entwicklungswerkzeuge, Softwareentwicklung und Echtzeitverhalten.

Da es sich beim untersuchten Linux um freie Software handelt, wurde im Kapitel 3 die dahinter steckende Philosophie dargestellt. Die Lizenzmodelle fanden dabei eine besondere Beachtung, da diese im Umfeld der Kosten von entscheidender Bedeutung sind.

In Kapitel 4 wird eine Einführung in das umfangreiche Gebiet des Entwurfs und der Implementierung von Betriebssystemen gegeben. Es wurden nur die Konzepte betrachtet, die in direktem Zusammenhang mit dem Thema Echtzeit stehen, dies sind z.B. das Scheduling und die Mechanismen zu Synchronisation.

Kapitel 5 gibt einen Überblick über die unter GNU/Linux verfügbaren Entwicklungswerkzeuge. In diesem Zusammenhang wird auch der POSIX-Standard mit seinen Erweiterungen für die Echtzeit-Programmierung vorgestellt.

Die Darstellung des unmodifizierten Linux Kernels, sowie die Untersuchung der echtzeitfähigen Varianten bezüglich ihres Lösungsansatzes für das Problem der Echtzeitfähigkeit, geschah in Kapitel 7.

Das Kapitel 8 umfasst die Analyse und Darstellung der sich daraus ergebenden Folgerungen bezüglich des Einsatzes echtzeitfähiger Linuxvarianten in der Automatisierungstechnik.

Hierbei ergeben sich folgende allgemeine Kernaussagen:

- Gegenüber proprietären echtzeitfähigen Betriebsystemen ergibt sich eine große Unabhängigkeit von Marktinteressen und der Geschäftsentwicklung auf Seiten des Herstellers.
- GPL-lizensierte Varianten bieten einen hohen Investitionsschutz, da durch die offenen Quellen jederzeit eine Weiterentwicklung der Software erfolgen kann, sei es durch den Hersteller oder durch Dritte.
- Für Hersteller ist es trotz GPL möglich, für Linux proprietäre Produkte zu entwickeln und somit die aus ihrer Sicht vorteilhafte Lösung zu wählen.
- Echtzeitfähige Linuxvarianten bieten eine ausgezeichnete Basis für Projekte im Bereich der Automatisierungstechnik mit ihren beschriebenen Anforderungen nach Performance, Sicherheit, Flexibilität.
- Die Problematik des RTLinux-Patents zeigt deutlich, welche Folgen Softwarepatente im Bereich freier Software nach sich ziehen, unter Umständen wirken sie sich nachteilig auf die weitere Verbreitung von echtzeitfähigen Linuxvarianten aus.

Abschnitt 8.3 stellt die Charakteristika der einzelnen Varianten dar. Diese können bei einem Auswahlverfahren als Basis zur Entscheidungsfindung benutzt werden. Wie ein solcher Vorgang aussehen kann, zeigt der Abschnitt 8.4.

9.2 Ausblick

Betrachtet man die Varianten fällt eine Tatsache ins Auge: es gibt keine verbindlich *garantierten* Antwortzeiten. Wünschenswert wäre eine Aussage wie:

> Bei Verwendung unseres Linuxkernels auf dieser Hardwareplattform mit diesen Treibern usw. garantieren wir für eine Anwendung mit der höchsten Priorität und nicht ausgelagert... dass sie innerhalb von X Microsekunden beim Auftreten des entsprechenden Interrupts aufgeweckt wird. Wenn sie dies nicht erreichen, behandeln wir dies als Fehler.

Wie bereits beschrieben ist dies ein sehr umfangreiche Aufgabe. Es gibt bereits verschiedene Messungen [157] und auch die Hersteller haben ihren Code analysiert, aber ein umfassender Vergleich - am besten mit einem standardisierten Test, um vergleichbare Ergebnisse zu erhalten - wäre von großem Interesse.

Weiterhin ist der sich Entwicklung befindliche 2.5er Kernel ebenfalls präemptiv [106]. Hierdurch könnten die Varianten, die dies bereits jetzt - jede auf ihre Weise - realisieren, in Zukunft zu einer zusammengefasst werden. Dies würde auch die Zahl der "Mutationen" reduzieren. Ein beliebter Kritikpunkt von Linux-Gegnern.

Eine umfassendere POSIX-Unterstützung könnte vielen Entwicklern den Umstieg auf Linux erleichtern. Hier ist die Entwicklung bereits im Gang und die Zukunft wird hier einiges bringen. Vielleicht findet sich mit wachsender Popularität dann auch eine oder mehrere Firmen, die eine POSIX-Zertifizierung finanzieren.

Wie die Situation bezüglich des RT-Linux Patents in Zukunft aussieht, wird vielleicht von entscheidender Bedeutung für den Einsatz echtzeitfähiger Linuxvarianten in der Automatisierungstechnik sein. Hier herrscht für viele eine zu große Unsicherheit, besonders für Firmen, die auch in den USA aktiv sind. Eine Klärung der Situation ist also wünschenswert.

Mit den analysierten echtzeitfähigen Linuxvariante existiert für jeden Bereich in der Automatisierungstechnik ein geeignetes Betriebssystem. Dabei bietet gerade das Modell der Freien Software in der Automatisierungstechnik viele Vorteile, die bei proprietären Betriebssystem nicht vorhanden sind.

Anhang A

Ein Prozeßbeispiel

In diesem Abschnitt erfolgt eine genauere Vorstellung des in der Einleitung erwähnten Prozessbeispiels.

A.1 Aufgaben der Prüfstände

Die Prüfung erfolgt an mechanischen Druckreglern (siehe Abb. A.1).

Die Prüflinge haben einen quaderförmigen Grundkörper, der an drei Seiten Druckluftanschlüsse hat. Senkrecht zu der dadurch aufgespannten Ebene kann über einen Spindeltrieb der Druck eingestellt werden.

Es werden folgende Parameter durch die Software überprüft [151]:

- Einstellbarkeit des Druckes
- Dichtigkeit der Hochdruckseite
- Dichtigkeit der Niederdruckseite

Abbildung A.1: Ein typischer Druckregler

- Durchgängigkeit des Gußkörpers

A.2 Hardware der Prüfstände

Das Einspannen der Regler erfolgt - mit dem Antrieb nach unten - von vier Seiten mit je einem Druckluftzylinder (siehe Abb. A.2).

Drei der Spannstücke tragen mit O-Ringen gedichtet durchbohrte Zapfen, in die die Regleranschlüsse gleiten. Die Verbindung zum Testsystem erfolgt über Druckluftschläuche. Die Spindel des Reglers wird von einer Asynchronmaschine hinter einem Frequenzumrichter betätigt. Weitere Aspekte der Hardware sind für die anschließenden Betrachtungen nicht von Interesse, da sie keinen direkten Einfluß auf die Steuerungssoftware haben. Es müssen die folgenden elektrischen Signale verarbeitet werden:

- 11 Digitale Ausgänge
- 5 Digitale Eingänge
- 3 Analog-Eingänge mit 12 Bit Breite
- 2 (bzw. 3) Analog-Ausgänge mit mindestens 7 Bit Breite

A.3 Die Ausgangssituation bei der Steuerungssoftware

Die eigentliche Intelligenz der Prüfstände liegt in der Steuerungssoftware. Sie steuert alle im Zusammenhang mit der Prüfung stehenden Abläufe. Ausgehend von der bestehenden Softwarelösung wird das neue Konzept vorgestellt, auf das sich auch die anschließenden Betrachtungen zum Einsatz der echtzeitfähigen Linuxvarianten beziehen.

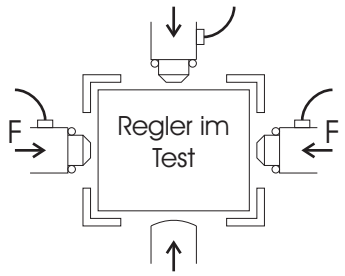


Abbildung A.2: Aufbau des Prüftisches

A.3.1 Die eingesetzte Software

Die Software erledigt in der zum Ausgangzeitpunkt (Anfang 2000) eingesetzten Version die folgenden Aufgaben:

- Steuerung des Prüfablaufs für die verschiedenen Reglertypen (z.Zt. 30).
- Verwaltung der Prüfparameter (lokal auf dem Prüfstand).
- Animierte Bildschirmdarstellung des Prüfablaufs.
- Prüfergebnisspeicherung (lokal auf dem Prüfstand).
- Benutzerverwaltung mit Paßwörtern und unterschiedlichen Rechten.
- Als Option: Weitergabe der Ergebnisse an einen FTP-Server im Netz.

A.3.2 Eingesetztes Steuerungskonzept

Die grundsätzliche Idee des realisierten Konzeptes ist es, so wenig wie möglich der Software in der SPS-Sprache Step7 [97] zu realisieren. Ziel war es, den komplexen Teil der Steuerung/Regelung und der Verwaltungsaufgaben in einer Hochsprache zu programmieren. Hierbei kam Turbo Pascal der Firma Borland in der Version 6.0 zum Einsatz [17].

Die SPS übernimmt also nur die Aufgabe, die Eingangs- und Ausgangs-Signale in ein PC-konformes Format zu konvertieren. Auf dem Steuerungs-PC wurde als Betriebssystem MU-DOS 5.1 aus den bereits in der Einleitung angegebenen Gründen eingesetzt. Die Architektur der Prüfungs-Software sieht für die einzelnen Aufgaben wie Visualisierung, Prüfablauf, Motorsteuerung, Druckeinstellung, Meßwerterfassung und Datenhaltung eigene Programme vor.

Aus der ursprünglich eingesetzten Lösung mit einer ET100 als "intelligenter Klemme" und der Anbindung des PCs als Steuerungsrechners über eine Koppelkarte ET100-PC ging schließlich die Lösung mit einer S7-200 hervor [95] [96] [97].

Die Ankopplung des PCs geschieht hierbei über die RS485-Schnittstelle und einem PPI-Kabel, welches die Konvertierung der Schnittstelle auf den RS232-Standard übernimmt (Abb. A.3).

So ist es möglich, jeden einzelnen Sensor oder Aktor direkt aus der Software heraus anzusprechen. Ein "zentrales" Programm (MESS) hält das Prozeßabbild und alle Programme synchronisieren sich über Queues, die als Betriebssystem-Service zur Verfügung stehen.

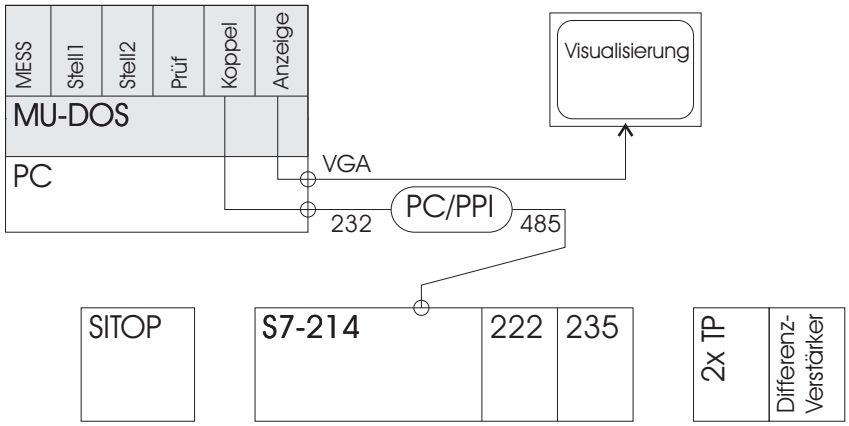


Abbildung A.3: Konzept mit direkter Kommunikation PC $\Longleftrightarrow$ S7-200

Ein weiterer Vorteil dieses Konzeptes ist, dass am Programm, welches auf der S7-200 läuft, nie etwas geändert werden muss, da sie nur dafür sorgt, das Prozeßabbild auf den PC zu übertragen. Weiterhin ist dies unabhängig vom Betriebssystem auf dem Steuerungs-PC, diese Tatsache zeigte ihre Vorteile besonders im weiteren Verlauf der Untersuchungen, da ein testweiser Wechsel des Betriebssystems keine Änderungen an der Programmierung der SPS erforderlich machte.

A.3.3 Das Koppelprogramm auf der S7

Wegen der zentralen Bedeutung für die Steuerungssoftware erfolgt hier ein genauerer Blick auf das realisierte Koppelprogramm auf der S7.

Um das Koppelprogramm auch für andere Einsatzfälle geeignet zu halten, wurde der Datenumfang für jede Richtung auf vier Digital- und vier Analogwerte festgelegt. Das entspricht dann je 32 Digitaleingabe bzw. Ausgabebits und vier 16-Bit Analogwerten. Nach dem gleichen Konzept wurde auch ein Demonstrationsmodell für eine Fahrstuhlsteuerung entwickelt.

Als Protokoll wurden diesen 12 Nettobytes noch ein Längenbyte vorangestellt, eine 16-Bit Checksumme und mit einem Carriage-Return / Line-Feed abgeschlossen. Hierdurch ist es möglich, die gesendeten Daten mit einfachen Mitteln zu empfangen und gegebenenfalls zu Testzwecken darzustellen. Die Checksumme ermöglicht eine einfache Kontrolle, ob die Daten fehlerfrei übertragen wurden. Die 17 Byte Sendedaten an die S7-200 ergeben sich dann wie folgt:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
17	A0	A1	A2	A3	AA0		AA1		AA2		PWP		CS		oa	od

AnalogAugang0 geht an das Analogmodul 235, AA1 und AA2 setzen die beiden Pulsweitenmodulatoren der S7-214 und PWP ist die für die beiden PWM's

gemeinsame Periodendauer.

Analog dazu sehen die von der S7-200 an den PC gesendeten Datenbyte aus:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
17	E0	E1	E2	E3	AE0		AE1		AE2		ZAE		CS		oa	od

Neben den vier Bytes für den Digitaleingang sind das die drei Analogmesswerte des 235 und der Zählerstand eines High-Speed-Counter. Die Übertragung der 17 Bytes erfolgt mit 19200 Baud.

Der Zeitbedarf errechnet sich wie folgt:

$$\frac{17Byte * 10Bit/Byte}{19200Bits/s} = 8,85ms$$

Um eine gesicherte Übertragung mit 19200 Baud zu ermöglichen, ist der Einsatz einer seriellen Schnittstelle mit FIFO-Baustein im PC notwendig. Bei jedem modernen PC ist diese Bedingung aber bereits erfüllt, bzw. lässt sich eine solche Schnittstelle leicht nachrüsten. Der PC ermöglicht eine "Abtastrate" des Prozeßabbildes von 100Hz, hier setzt allerdings die Schnittstelle der S7-200 Grenzen.

Für die Durchführung der Prüfprozesse ist es ausreichend, alle 100ms ein Prozeßabbild zu übertragen, daraus ergibt sich eine Abtastrate von 10Hz. Die Kommunikation erfolgt im Halbduplex-Betrieb, da die S7-200 nicht voll duplex kommunizieren kann.

Das Koppelprogramm als Anweisungsliste (AWL) in Step 7 hat einen Umfang von 250 Zeilen (siehe Anhang).

A.4 Geplante Erweiterungen

In der Abbildung A.4) sind die geplanten Erweiterungen der Prüfstandssoftware dargestellt.

Die Anbindungen an die Datenbank des PPS-Systems sorgt dafür, dass die Prüfvorschriften zentral und normgerecht verwaltet werden können. Im PPS-System erfolgt auch die Archivierung der Prozeßdaten. Somit ist eine lückenlose Dokumentation des Fertigungs- und des abschliessenden Prüfvorgangs möglich.

Die Visualisierung und Fernwartung dient dazu, den Prüfablauf zu kontrollieren und im Fehlerfall eine Analyse des Problems durchführen zu können.

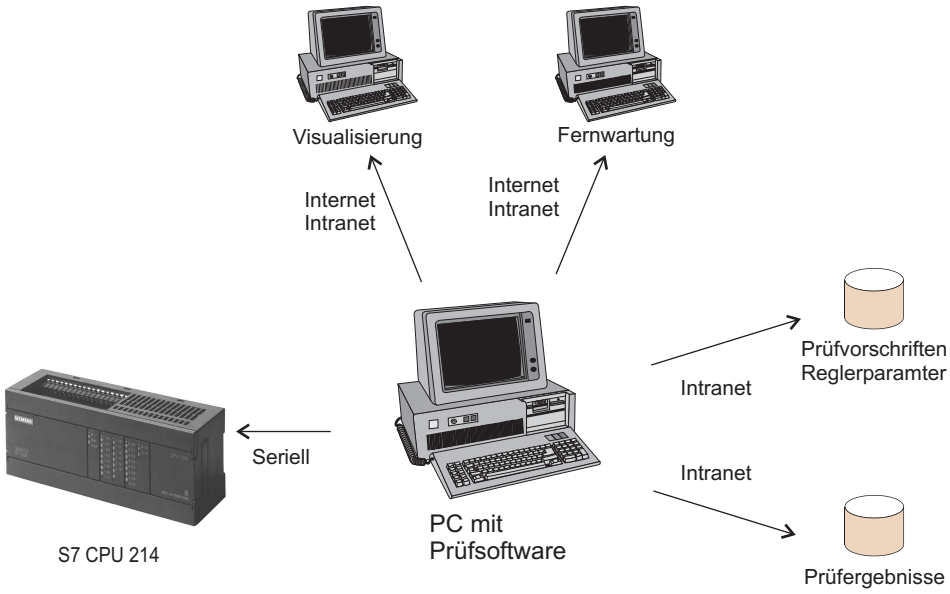


Abbildung A.4: Integration des Prüfstandes in das Netzwerk

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Tigran Aivazian. Linux Kernel 2.4 Internals, Okt. 2001. http://www.moses.uklinux.net/patches/lki.html.

[2] Albers u. Fromberger Multiusersysteme, Dortmund. *A + F TEAM 4.0 - Software für Logistik - Produktion - Rechnungswesen*, 1998.

[3] N. Allahwerdi, J. Baudin, P. Gaffney, W. Grimson, T. Groth, A. Laugier, and L. Schilders. Remote instrument telemaintenance. *Computer Methods an Programs in Biomedicine*, 50:187–194, 1996.

[4] Furquan Ansari, Robert Hill, Douglas Niehaus, Shyam Pather, and Balaji Srinivasan. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. Techn. Report, ITTC / University of Kansas, 1998.

[5] George Anzinger. The real-time scheduler project, 2000. http://rtsched.sourceforge.net.

[6] Heide Balzert. *Methoden der objektorientierten Systemanalyse*, volume 14 of *Angewandte Informatik*. BI-Wiss.Verl., Mannheim; Leipzig; Wien; Zürich, 1995.

[7] Michael Barabanov. *A Linux-based Real-Time Operating System*. Dissertation, New Mexico Institute of Mining and Technology, Socorro, New Mexico, 1997.

[8] Michael Barabanov and Victor Yodaiken. ELJonline: Real-Time Applications with RTLinux, 2001. http://www.linuxdevices.com/articles/AT8948080759.html.

[9] Dieter Barelmann. Offenes Fernwirkkonzept. *ETZ Elekrtotechnik + Automation*, 13:51–54, 1999.

[10] Joseph S. Barrera III, Alessandro Forin, Michal B. Jones, Paul J. Leach, Daniela Rosu, and Marcel-Catalin Rosu. An Overview of the Rialto Real-time Architekture. Techn. Report, Microsoft Research, Redmond, 1996.

[11] D. L. Bayer and H. Lycklama. The MERT Operating System. *Bell System Technical Journal*, 1978.

[12] R. Beauvais. Software-Entwicklung als Kernkompetenz. *Computerwoche*, (16):49–50, 1998.

[13] L. Belady and M. Lehman. The characteristics of large systems, Program Evolution - Process of sofware change. *APIC studies in data processing*, (27):289–329, 1985. academic press.

[14] Rudy Belliardi, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The real-time specifikation for java*. Addison-Wesley, Boston, San Francisco, New York, Toronto, 1 edition, 2000.

[15] Tim Bird. Comparing two approaches to real-time Linux, Dez. 2000. http://www.linuxdevice.com.

[16] Josef Börcsök. *Prozeßrechner und -automation*. Heinz Heise GmbH Co. KG, Hannover, 1996.

[17] GmbH Borland, editor. *Turbo PAscal 6.0 Benutzerhandbuch*. Schoder Druck GmbH, München, Okt. 1990.

[18] Daniel P. Bovert and Marco Cesati. *Understanding the linux kernel*. O'Reilly & Associates Inc., Bonn, Cambridge, Paris, Sebastopol, Tokyo, 1 edition, Jan. 2001.

[19] Ivan Bowman. Conceptual architecture of the linux kernel, 1998. http://plg.uwaterloo.ca/ itbowmann/CS746G/a1/.

[20] Ivan Bowman, Saheem Siddiqi, and Meyer c. Tanuan. Concrete architecture of the linux kernel, 1998.
http://plg.uwaterloo.ca/ itbowmann/CS746G/a2/.

[21] Nichols Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates Inc., BoN, Cambridge, Paris, Sebastopol, Tokyo, 1996.

[22] Peter Brich, Gerhard Hinsken, and Karl-Heinz Krause. *Echtzeitprogrammierung in Java*. Publicis MCD Verlag, 2000.

[23] Alan Burns and Andy Wellings. *Realtime systems and their programming languages*. Addison-Wesley, 1 edition, 1989.

[24] Alan Burns and Andy Wellings. *Realtime systems and their programminglanguages*. Addison-Wesley, 3 edition, 2001.

[25] Ava Chen. *Windows CE Prgrammers Guide*. Microsoft Press, 1998.

[26] J. Bradley Chen and Daniel Stodolsky. Fast interrupt priority managment in operating system kernels. Techn. Report, Carnegie Mellon University, Pittsburgh PA, 1993. http://citeseer.nj.nec.com/article/stodolsky93fast.html.

[27] Julie E. Cohen and Mark A. Lemley. Patent Scope and Innovation in the Software Industry, Jan. 2001. http://www.law.georgetown.edu/faculty/jec/.

[28] J. E. Cooling. *Software design for real-time systems*. Chapman and Hall, London, New York, Tokyo, Mebourne, Madras, 1991.

[29] Jonathan Corbet and Allesandro Rubini. *Linux device drivers*. O'Reilly & Associates Inc., Bonn, Cambridge, Paris, Sebastopol, Tokyo, 2 edition, 2001.

[30] Horst Czichos, editor. *Die Grundlagen der Ingenieurwissenschaften*. Springer, Berlin Heidelberg New York, 30 edition, 1996.

[31] Björn Dähn. Auf dem Vormarsch. *iX*, (4):S. 60ff, April 2001.

[32] Kevin Dankwardt. Comparing real-time Linux alternatives, Okt. 2000. http://www.linuxdevices.com.

[33] David Diamond and Linus Torvalds. *Just For Fun*. Hanser, München, Wien, 2001.

[34] Werner Dietrich and Thomas Gebhardt. *CDOS CCP/M-86*. Hüthig, Heidelberg, 1988.

[35] Werner Dietrich and Thomas Gebhardt. Volle Power. *ElektronikPraxis*, (5):38–41, März 1998.

[36] Digital Research Inc., Monterey, California. *Multiuser DOS - Benutzerhandbuch*, März 1991.

[37] Digital Research Inc., Monterey, California. *Multiuser DOS - Programmers Guide*, Jan. 1991.

[38] J.-E. Dubois and N. Gershorn. *Industrial Information and Design Issues*. Springer, Berlin Heidelberg, 1996.

[39] Jerry Epplin. Using GPL software in embedded applications, März 2001. http://www.linuxdevices.com.

[40] A. Ferro De Beca, B. Iung, J.B. Leger, and J. Pinoteau. An innovative approach for new distributed maintenance system. *Computers in Industry*, 38:131–148, 1999.

[41] Inc. Free Software Foundation. GNU general public license, 1991. http://www.gnu.org/license/.

[42] Inc. Free Software Foundation. GNU lesser general public license, 1999. http://www.gnu.org/license/.

[43] Kevin Fu. RT-Linux: An Interview with Victor Yodaiken, 1999. http://www.acm.org/crossroads/xrds6-1/yodaiken.html.

[44] Berczi Gabor. Free Sockets Interface, 2000. http://www.phekda.freeserve.co.uk/richdawe/dossock/.

[45] Richard P. Gabriel and William N. Joy. Sun community source license principles, 2002.
http://www.sun.com/software/communitysource.

[46] Richard Gail and L. Kleinrock. *Queueing Systems*. John Wiley & Sons, Inc., New York, Chichester, Brisbane, Toronto, Singapore, 1996.

[47] Daniel D. Gajski, Jie Gong, Sanjiv Narayan, and Frank Vahid. *Specification and design of embedded systems*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1994.

[48] Bill O. Gallmeister. *POSIX.4 Programming For The Real World*. O'Reilly & Associates Inc., 1995.

[49] Jack Ganssle. Is embedded Linux a Bust, Nov. 2001.
http://www.embedded.com.

[50] Simson Garfinkel. *Database nation: the death of privacy in the 21st century*. O'Reilly & Associates Inc., Beijing, Cambridge, Köln, Paris, Sebastopol, Taipei, Tokyo, 2000.

[51] David Garland and Mary Shaw. *Advances in software engineering and knowledge engineering*, volume 2. World Scientific, 1993.

[52] Robert Gehring, Axel H. Horns, and Bernd Lutterbeck. Sicherheit in der Informationstechnologie und Patentschutz für Software-Produkte - Ein Widerspruch? Kurzgutachten, Berlin, Dez. 2000.

[53] Rubert Ralf Geiger. Echtzeitverarbeitung, 2001.
http://encarta.msn.com.

[54] Rainer Glatz, Tuomo Honkanen, and Ismo Niemela. Lebenszyklus-Management von Ventilen. *chemieanlagen + verfahren*, (4):126–128, 1999.

[55] Library Steering Committee GNU C. GNU C Library, 2002. http://www.gnu.org/projekts/.

[56] James Gosling and Henry McGilton. The Java Language Environment. Technical report, Sun Microsystems, Mai 1996.

[57] Johann Haas. PC an der Leine. *Elektronik*, (10):92–94, 1995.

[58] Greg Hawley. Selecting a real-time operating system. *Embedded Systems Programming*, (3), März 1999.

[59] Heinz-Gerd Hegering and Alfred Läpple. *Ethernet: Basis für Kommunikationsstrukturen*. DATACOM-VERLAG, Bergheim, 1992.

[60] G.T. Heng. Microcomputer-based remote unit for a SCADA system. *Microprocessors and Microsystems*, 20, 1996.

[61] Sebastian Hetze, Dirk Hohndel, and Martin Müller. *Linux Anwenderhandbuch*. Linuxland München, München, 7 edition, 1997.

[62] Sven Heursch. Latenzzeitmessungen des Betriebssystems Linux. Diplomarbeit, Universität der Bundeswehr München, 1999.

[63] R.C. Hicks. Minimizing maintenance anomalies in expert systems. *Information & Managment*, 28:177–184, 1995.

[64] Robert Hill, Douglas Niehaus, Shyam Pather, and Balaji Srinivasan. Temporal Resolution and Real-Time Extension to Linux. Techn. Report, ITTC / University of Kansas, Juni 1998.

[65] Peter Holleczek, editor. *PEARL 2000 - Echtzeitbetriebsysteme und Linux*. Springer, Berlin Heidelberg, 2000.

[66] Jordan K. Hubbard. Übersicht über die BSD-Projektziele, 2001. http://www.freebsd.org/handbook/handbook4.html.

[67] Valerie Illingworth. *Oxford Dictionary of Computing*. Oxford University Press, 3 edition, 1991.

[68] Reinhold Keck. Neue Wege in der Fernwirktechnik. *ETZ Elekrtotechnik + Automation*, (18):40–41, 1997.

[69] Helmut Kerner. *Rechnernetze nach OSI*. Addison-Wesley (Deutschland), Bonn; Paris; Reading Mass., 2 edition, 1993.

[70] Wooju Kim, B.Y. Lee, and J.K. Lee. A Knowledge-Based Maintenace of Legacy Systems: METASOFT. *Expert Systems With Applications*, 12:483–496, 1997.

[71] L. Kleinrock. *Queueing Systems*, volume 1. Wiley Intersience Publication, New York, 1975.

[72] Peter Knopf and K. Neubeck. Fernüberwachung von Systemen zu Ver- und Entsorgung. *ETZ Elekrtotechnik + Automation*, (22):6–8, 1998.

[73] Jochen Küpper. rt_com: real-time serial port device driver, 1999. http://rt-com.sourceforge.net/.

[74] Douglas Kramer. The Java Platform. Techn. Report, Sun Microsystems, 1996.

[75] Monica S. Lam and Jason Nieh, editors. *The Design, Implementaiton an Evaluation of SMART*, Proceedings of the Sixteenth ACM Symposium on Operating System Principles, St.Malo, France, Oktober 1997.

[76] Phillip Laplante. *Real-time systems design and analysis*. IEEE Press, 2 edition, 1997.

[77] Andrew Leonard. Name that movement: Open source or free software?, Feb. 1999. http://www.salon.com/21st/log/1999/02/17log.html.

[78] Donald Lewine. *POSIX Programmers Guide*. O'Reilly & Associates Inc., 5 edition, 1994.

[79] Wolfgang Lex. Fernwirken per Funk mit Modacom - sicher und komfortabel. *ETZ Elekrtotechnik + Automation*, (13):38–41, 1998.

[80] Wolfgang Lex. Anlagen überwachen mit Handy und SPS. *ETZ Elekrtotechnik + Automation*, (13):24–25, 1999.

[81] Kwei-Jay Lin and Yu-Chung Wang. Enhancing the Real-Time Capability of the Linux Kernel. Techn. Report, Department of Electrical and Computer Engineering (University of California), Irvine, 1998.

[82] Kwei-Jay Lin and Yu-Chung Wang. Providing Rel-Time Support in the Linux Kernel. Technical report, Department of Electrical and Computer Engineering (University of California), Irvine, 1998.

[83] Robert Love. The Linux kernel preemption project, 2000. http://kpreemt.sourceforge.net.

[84] Claes Lundhom and Bill Weinberg. *White Paper: Embedded Linux - Ready for Real-Time*. MontaVista Software, 2001.
http://www.mvista.com.

[85] Arnold Mayr, Joachim Kroll, and Bernd Reichert. Fernwirktechnik für Ortsnetzstationen. *ETZ Elekrtotechnik + Automation*, (5):238–243, 1994.

[86] Nicholas McGuire. Identifying the top requirements for embedded Linux systems, März 2002.
http://www.linuxdevices.com.

[87] Stefan Meretz. GNU/Linux ist nichts wert - und das ist gut so!, April 2000.
http://www.kritische-informatik.de/lxwert.htm.

[88] H. J. Meyer. Einsatzpotential von Teleservicesystemen in der Landwirtschaft. In *Landtechnik 1997*, number 1356 in VDI Berichte. 1997.

[89] H. J. Meyer. Funk-Fernübertragung und Meßwertübertragung im 70cm Band. *ETZ Elekrtotechnik + Automation*, (13):26–27, 1999.

[90] Paolo Montegazza and Steve Papacharalambous. DIAPM-RTAI Position Paper, 2000.
http://www.rtai.org.

[91] Kevin Morgan. Echtzeit zwischen hart und weich. *Linux Magazin*, pages 108,109, April 2002.

[92] G. Müller. EDV und Servicegrad. In *Rechnergestützter Kundendienst 1988*, number 633 in VDI Berichte. 1988.

[93] Martin Müller. *Open Source*. O'Reilly & Associates Inc., Beijing, Cambridge, Köln, Paris, Sebastopol, Taipei, Tokyo, 1999.

[94] Thilo Müller. Entwicklung und Programmierung einer Prüfstandsteuerung. Studienarbeit, Universität Dortmund / MB MT, Dortmund, Juni 1989.

[95] N N. SIMATIC S5 - Speicherprogrammierbare Steuerungen. Katalog, Siemens AG, Nürnberg, Okt. 1993.

[96] N N. Automatisierungsystem S7-200 - Aufbau, CPU Daten. Hanbuch, Siemens AG, Nürnberg, 1995.

[97] N N. Automatisierungssystem S7-200. Systemhandbuch, Siemens AG, Nürnberg, 1998.

[98] N N. Safeguarding your Technologie - Practical Guidlines for Electronic Education Information Security, 1998. http://nces.ed.gov/pubs98/safetech/chapter7.html.

[99] N N. The Single UNIX Specification, Version 2 and UNIX 98. Techn. Report, The Open Group, 1998.

[100] N N. *INTERNATIONAL STANDARD ISO/IEC 9899:1999 (E)*. Internatinal Organisation for Standardization, 1999.

[101] N N. Apache Web Server, 2000.
http://www.apache.org.

[102] N N. Boa Web Server, 2000.
http://www.boa.org.

[103] N N. KDE, 2000.
http://www.kde.org.

[104] N N. RTAI Mailingliste, 2000.
http://www.rtai.org.

[105] N N. SICOMP RMOS, 2000.
http://www.ad.siemens.de/sicomp/html_76/sw_rmos.htm.

[106] N N. Changelog Linux-Kernel V2.5, 2001.
http://www.linuxhq.com/kernel/v2.5/changes.

[107] N N. Free Software Foundation, 2001.
http://www.fgf.org.

[108] N N. GNU-Projekt, 2001.
http://www.gnu.org.

[109] N N. IEEE Standards Association Home Page, 2001.
http://standards.ieee.org.

[110] N N. iRMX for Windows, 2001.
http://www.tenasys.com/irfw/default.htm.

[111] N N. MontaVista Software Inc., 2001.
http://www.mvista.com.

[112] N N. Mozilla NPL, 2001.
http://www.mozilla.org.

[113] N N. The XFree Project, 2001.
http://www.xfree.org.

[114] N N. Unifix Software GmbH, 2001.
www.unifix.de.

[115] N N. About the java technology, 2002. http://java.sun.com/doc/books/tutorial/getStarted/intro/definition.html.

[116] N N. GPL: Categories of Free and Non-Free Software, 2002. http://www.gnu.org/philosphy/categories.html.

[117] N N. GPL Frequently asked Questions, 2002. http://www.gnu.org/licenses/gpl-faq.html.

[118] N N. Information and Telekommunication Technology Center, 2002. http://www.ittc.ukans.edu.

[119] N N. Java Community Process (JCP), 2002. http://www.jcp.org.

[120] N N. KURT: THE KU Real-Time Linux, 2002. http://www.ittc.ukans.edu/kurt/.

[121] N N. Linux-Automation, 2002. http://www.linux-automation.de.

[122] N N. Microsoft.com, 2002. http://www.microsoft.com.

[123] N N. Open Secure Shell, 2002. http://www.openssh.org.

[124] N N. Open Source Initiative, 2002. http://www.opensource.org.

[125] N N. Realtimelinux.org, 2002. http://www.realtimelinux.org.

[126] N N. REDSonic Inc., 2002. http://www.redsonic.com.

[127] Stefanie Prozny. Freie Software - Philosophie und Organisationsmodell. Diplomarbeit, Universität Dortmund / Empirische Wirtschafts und Sozialforschung, Dortmund, 1999.

[128] Software Ltd QNX, editor. *QNX System Architektur*. QNX Software Systems Ltd, Ontario, 2 edition, 1997.

[129] C. Ritchie. *Operating Systems*. DP Publications Ltd, London, 1 edition, 1992.

[130] Klaus-Dieter Scheffer and K. Neubeck. Fernüberwachung von Gasversorgungssystemen. *ETZ Elekrtotechnik + Automation*, (22):36–38, 1997.

[131] Siegmar Schmidt. http://www.waage.de. *Elektronik*, (14), Juli 1999.

[132] Uwe Schmidt. Einheitlichkeit macht stark. *Elektrizitätswirtschaft*, (14):32–35, Juni 1999.

[133] Goran Schröder. Pipline-Leitzentrale in Bygnes am Ende des Weges zur Jahr-2000-Konformität. *ABB Technik*, März 1999.

[134] Peter Schulze. PC an der Strippe. *Elektronik*, (4):18, 1994.

[135] Peter Schulze. Echtzeit Betriebssysteme. *Design & Elektronik*, (7):50–52, Juli 1999.

[136] Peter Schulze. Marktübersicht USV-Anlagen. *ETZ Elekrtotechnik + Automation*, (10):24–29, 1999.

[137] Rudolf Schulze. Maschinenbauer gehen ins Netz. *VDI nachrichten*, (41), Okt. 1998.

[138] Robert Schwebel. *Embedded Linux*. mitp-Verlag, Bonn, 2001.

[139] Robert Schwebel. News from the License front. Newsgroup Artikel, Okt. 2001. http://www.realtimelinux.org/archives/rtai/200110/0095.html.

[140] Jürgen Scriba. Frische Bits vom Basar, März 1999. www.spiegel.de/netzwelt/technologie/0,1518,14103,00.html.

[141] Markku Simula. RTOS und Embedded Webserver. *Design & Elektronik*, (7):26, Juli 1999.

[142] Richard Stallman. The Free Software Definition, 2001. http://www.gnu.org/philosophy/free-sw.html.

[143] Richard Stallman. What is copyleft?, 2001. http://www.gnu.org/copyleft/copyleft.html.

[144] Claus Stellwag. Kostenaspekte entscheiden. *Elektronik*, (22):112–115, 1997.

[145] W. Richard Stevens. *UNIX network programming*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1990.

[146] Ralph Steyer. *Java2*. Markt und Technik, München, 2000.

[147] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1992.

[148] Robin Tawab. Wizzard Watchdog. Techn. Report, Apptime AG, München, 1999.

[149] J. Tegethoff and Andre Hergenhan. Zustandsorientierte Instandhaltung von Dampfturbinen. *Allianz Report*, (2):120–125, 1999.

[150] Mathias Uhle. Universität Dortmund, Maschinenbau, Messtechnik. http://www.mt.mb.uni-dortmund.de.

[151] Mathias Uhle. Siemens SIMATIC S7-200 als intelligente Klemme. In W. Ebster, editor, *SAK Tagungsband*. Proleit AG, Mai 1998.

[152] Das deutsche Urheberrechtsgesetz - UrhG, Sep. 2000.

[153] Helge Weber and Bernd Reichert. Modulares Mini-Fernwirkgerät intergiert Automatisierungsfunktion. *ETZ Elekrtotechnik + Automation*, (13):46–47, 1998.

[154] Christoph Weiler, N N, and Wolfgang Rosenstiel. Internet-basierte eingebettete Systeme in der industriellen Automation. *at Automatisierungstechnik*, (7):305–312, Juli 1999.

[155] Bill Weinberg. Moving from a proprietary RTOS to embedded Linux. Technical report, MontaVista Inc., 2001.

[156] Engelbert Westkämper. Zukünftig bleiben Maschinen lebenslang im Infonetz des Herstellers. *VDI nachrichten*, (41), Okt. 1998.

[157] Clark Williams. Linux Scheduler Latenca. Technical report, Red Hat Inc., März 2002.

[158] Karim Yaghmour. Re: A reply on the RTLinux discussion, Mai 2002. http://www.uwsg.iu.edu/hypermail/linux/kernel/0205.3/1016.html.

[159] Victor Yodaiken. *Adding real-time support to general purpose operating systems*. US Patent and Trademark Office, Nov. 1997.

[160] S. Young. *Real time languages: design and development*. Ellis Horwood Publishers, Chichster, 1982.

Lebenslauf

Name und Anschrift:	Dirk Düding Wipfelweg 34 44265 Dortmund
Geburtsdatum:	30. Juni 1969
Geburtsort:	Dortmund
Staatsangehörigkeit:	deutsch
Familienstand:	ledig

Daten	Tätigkeit	Arbeitgeber / Organisation
1975 - 1979	Schulbesuch	Wichlinghofer Grundschule Dortmund
1979 - 1988	Schulbesuch	Goethe-Gymnasium Dortmund
10/88 - 12/89	Wehrdienst	Bundeswehr
10/89 - 06/96	Studium des Maschinenbaus Vertiefungsrichtung: Materialfluß und Logistik	Universität Dortmund
10/96 - 04/02	Wissenschaftlicher Angestellter	Universität Dortmund Fakultät Maschinenbau Fachgebiet Meßtechnik
05/02 -	Systemanalytiker	Krupp Edelstahlprofile GmbH
01/93 - 06/96	Studentische Hilfskraft	Universität Dortmund, Fakultät Maschinenbau Fachbereich Meßtechnik